NoSQL DBMS and queries

Databases

Ira Assent

ira@cs.au.dk

Data-Intensive Systems group, Department of Computer Science, Aarhus University, DK

Intended learning outcomes

- Be able to
  - Describe the data model and query language in NoSQL databases
  - Discuss differences in common NoSQL systems

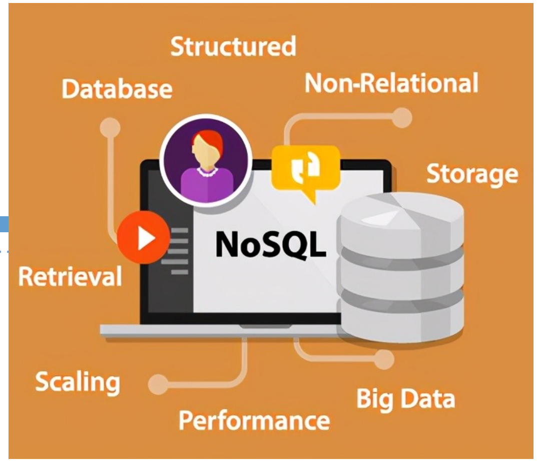
ira@cs.au.dk

Recap: Security and Authorization

- SQL injection: passing user input directly to DBMS is risky
  - Instead of expected values, may contain SQL code (parts)
  - Use variable binding and sanitizing

- Authorization to restrict data access
  - `GRANT` *`privileges`* `ON` *`object`* `TO` *`identity`*
  - `SELECT` (retrieval or read), modify privileges (`INSERT`, `DELETE`, `UPDATE`), `REFERENCES`, `WITH GRANT OPTION`
  - `CREATE ROLE` to group access privileges into profiles
  - Use views for fine-grained authorization for `SELECT`s
  - `REVOKE` to remove access rights again

Recap: NoSQL data model



- NoSQL (Not Only SQL)
  - High performance, Availability, Replication, Scalability
- Willing to sacrifice
  - some (immediate) data consistency
  - powerful query languages
  - structured data storage
- Emphasize performance and flexibility over modeling power and complex querying
- Typically, **no schema** required
  - Allow **semi-structured**, **self-describing** data
    - JSON (JavaScript Object Notation), XML (Extensible Markup Language),…
    - Idea: description, such as tags, part of the data objects

```
<document>
   <type>Student</type>
   <name>An</name>
   <study><stname>CS</stname>
      <level>BSc</level>
   </study>
</document>
```


ira@cs.au.dk

Why self-describing formats in NoSQL?

A. They are easier to read.
B. They are more flexible.
C. They are easier to normalize.
D. They allow for better compression.

ira@cs.au.dk

NoSQL systems: query languages

- Less powerful query languages
- Many NoSQL based systems rely on simple search/read access
  - Find particular object using id
  - No need for complex query languages that express powerful conditions and combinations across tables
- Many NoSQL systems provide functions and operations via an API (application programming interface)
  - Read/write done via function/operation calls
  - CRUD (Create, Read, Update, Delete) operations
  - SCRUD (plus Search)
  - Some provide query language that corresponds to subset of SQL capabilities
    - Typically join not available
    - If needed: must be handled by application program
- Versioning
  - Some NoSQL systems provide storage of multiple versions
  - Timestamps of version creation

ira@cs.au.dk

Categories of NoSQL systems

- Four major categories of systems
  - Document-based
    - Store data as documents using e.g. JSON
    - Access using document id, other indexes
  - Key-value stores
    - Simple data model of key-value, where value can be record, document, some other (complex) data structure
  - Column-based / wide-column
    - Partition tables by column families (vertical partitioning)
    - Each column family stored in separate file
    - Typically versioning
  - Graph-based
    - Graphs as data model
    - Access data by traversing edges using path expressions

ira@cs.au.dk

MongoDB: document-based NoSQL

- **Document-based** also called **document store**
- Data model: collection of documents
  - Documents in a collection are typically similar, but can have different data elements (attributes)
    - Great flexibility: if you feel like adding an attribute, just put in a new element in the document in question
  - Documents are self-describing, typically BSON based (binary JSON) format

```
"document": {
        "type": "Student Record",
        "name": "Anne Christensen",
        "study": "Computer Science",
        {
              "level": "Bachelor"
        }
    }
```


ira@cs.au.dk

MongoDB: document-based NoSQL

COMPANY database example

- **Create** a collection called project to hold PROJECT objects

```
db.createCollection(”project”, {capped:
true, size: 1310720, max: 500})
```

  - Capped: limit on storage (size) and number of documents (max)

```
db.createCollection(”worker”, {capped:
true, size: 5242880 max: 2000})
```

- Each document in a collection has **automatically indexed unique identifier** called `_id`

ira@cs.au.dk

MongoDB CRUD operations

- CRUD: create, read, update, delete
  - **Create and insert** documents into collection using

    db.<collection_name>.insert(<document_name(s)>)
- Textbook example

```
db.project.insert( { _id: "P1", Pname: "ProductX", Plocation: "Bellaire" } )
db.worker.insert( [ { _id: "W1", Ename: "John Smith", ProjectId: "P1", Hours: 32.5 },
                    { _id: "W2", Ename: "Joyce English", ProjectId: "P1",
                      Hours: 20.0 } ] )
```

  - **Delete**: db.<collection_name>.remove(<condition>)
  - **Read**: db.<collection_name>.find(<condition>)
  - General Boolean conditions evaluating to true are selected

mongoDB

Document-based NoSQL and MongoDB

- Collection **does not have schema**
  - Structure of data fields in documents
    - **Normalized**
      - Decompose into documents of similar structure and content
        - References between documents
          - Store ids of other documents in the document
    - **Denormalized**
      - All information in one document

ira@cs.au.dk

Denormalized pattern

- Textbook example: store workers John, Joyce in ProductX document
- Information about workers and projects stored together
  - In relation model: corresponds to storing also the workers in the project table
  - **denormalized**
  - […] notation: array
- information about workers **embedded** in project document, no need for separate worker collection

project document with an array of embedded workers:

```
{
    _id:            "P1",
    Pname:          "ProductX",
    Plocation:      "Bellaire",
    Workers: [
                  { Ename: "John Smith",
                    Hours: 32.5
                  },
                  { Ename: "Joyce English",
                    Hours: 20.0
                  }
             ]
);
```


ira@cs.au.dk

Embedded array of references

- Alternative: **embed only references,** here to workers via their id

project document with an embedded array of worker ids:

```
{
    _id:          "P1",
    Pname:        "ProductX",
    Plocation:    "Bellaire",
    WorkerIds:    [ "W1", "W2" ]
}
    { _id:        "W1",
    Ename:        "John Smith",
    Hours:        32.5
}
    { _id:        "W2",
    Ename:        "Joyce English",
    Hours:        20.0
}
```


ira@cs.au.dk

What’s this?

A. Fully normalized representation of projects and workers
B. Normalized representation of projects and workers, but not for M:N relationships
C. Denormalized representation of projects and workers
D. Denormalized representation of workers embedded in projects

```
{
    _id:         "P1",
    Pname:       "ProductX",
    Plocation:   "Bellaire"
}
{   _id:         "W1",
    Ename:       "John Smith",
    ProjectId:   "P1",
    Hours:       32.5
}
{   _id:         "W2",
    Ename:       "Joyce English",
    ProjectId:   "P1",
    Hours:       20.0
}
```


ira@cs.au.dk

Normalized documents

- Fully **normalized** for many-to-many requires three collections project, employee, works_on (as seen in relational model before)
  - Here: employee working on several projects represented by several worker documents with different ids
    - **redundancy**

normalized project and worker documents (not a fully normalized design for M:N relationships):

```
{
    _id:          "P1",
    Pname:        "ProductX",
    Plocation:    "Bellaire"
}
{   _id:          "W1",
    Ename:        "John Smith",
    ProjectId:    "P1",
    Hours:        32.5
}
{   _id:          "W2",
    Ename:        "Joyce English",
    ProjectId:    "P1",
    Hours:        20.0
}
```


ira@cs.au.dk

Voldemort

- Voldemort open source project builds on Amazon DynamoDB
- Data model:
  - Collections of self-describing items
    - NOT like relational model, without schema
  - Items are attribute-value pairs
    - Item can also be JSON file

- **Read** the entry with key 1234
  - `get(1234) => {"name":"ann", "email":"ann.jensen@cs.au.dk"}`
- **Write** the entry with key 1234
  - `put(1234, {"name":"ann jensen", "email":"ann.jensen@cs.au.dk"})`
- **Delete** the entry with key 1234
  - `Delete(1234)`

https://www.project-voldemort.com/voldemort/design.html

ira@cs.au.dk

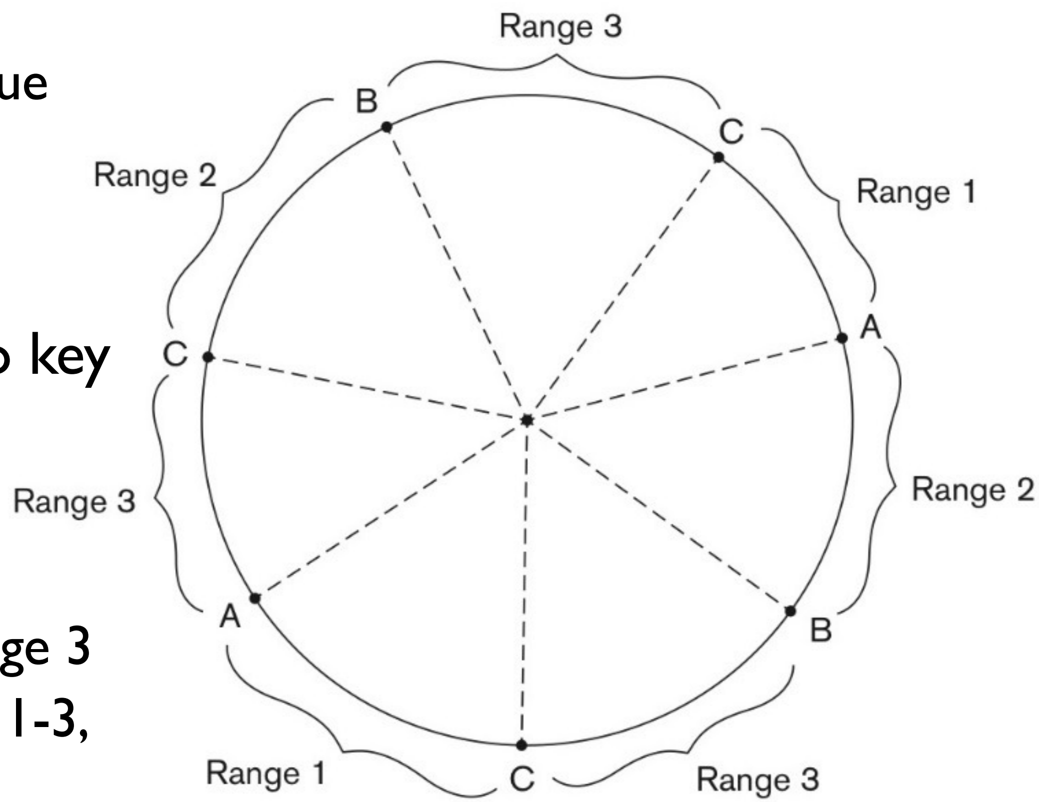
Key-Value Storage

- Only very simple data model: key-value data access
  - keys and values can be complex compound objects (including JSON)
- High performance, availability and scalability via distributed storage system
- No expressive query language, but set of operations to be used in application programs
- Pros
  - efficient queries, very predictable performance
  - easy to distribute across a number of sites (distributed database)
  - no object-relational impedance mismatch
- Cons
  - no complex queries
  - all joins must be done in program code
  - no foreign key constraints
  - no triggers

ira@cs.au.dk

Scaling Voldemort storage

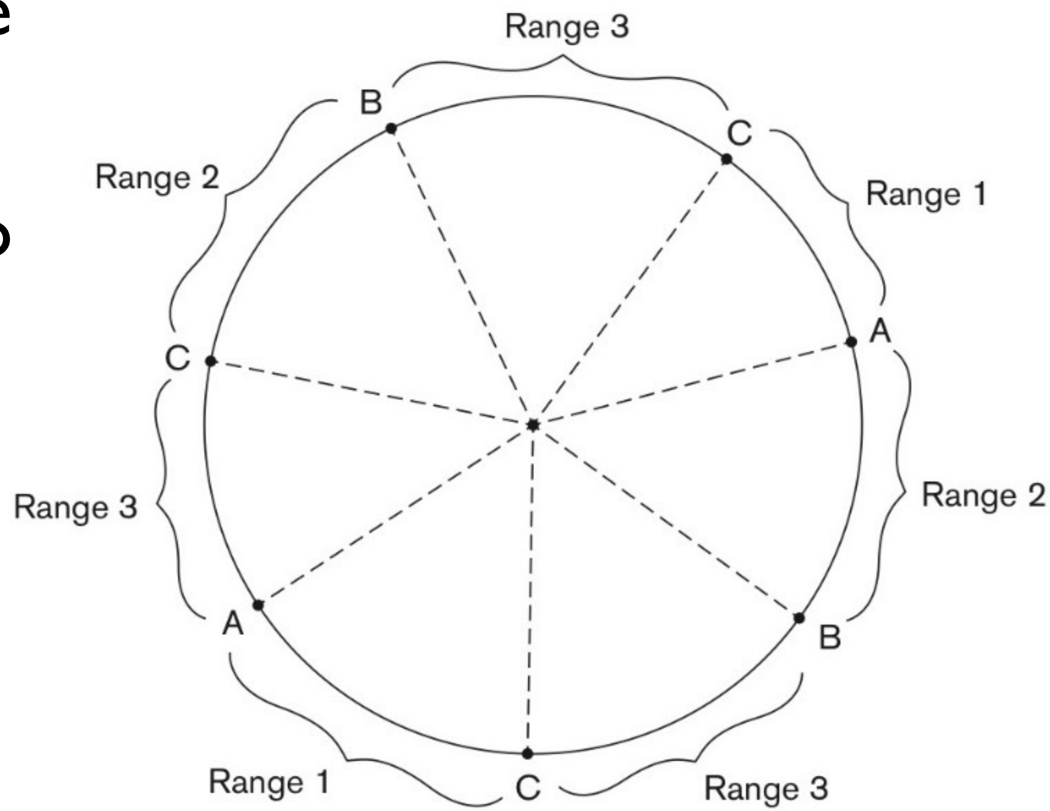
- Key-value data access across distributed database supported by consistent hashing
  - Map the key value to a hash value
  - Range of hash values stored by some site (or several for replication)
- Hash function h(key) applied to key of (key, value) pair
  - E.g. h(key)= value mod 5
    - h(3)=3, h(7)=2, h(10)=0
  - E.g. A: range 1, B: range 2, C: range 3
    - Ranges could be hash values 1-3, 4, 5-0
    - Ideally, each server gets the same workload and amount of data



ira@cs.au.dk

How do we scale to more data?

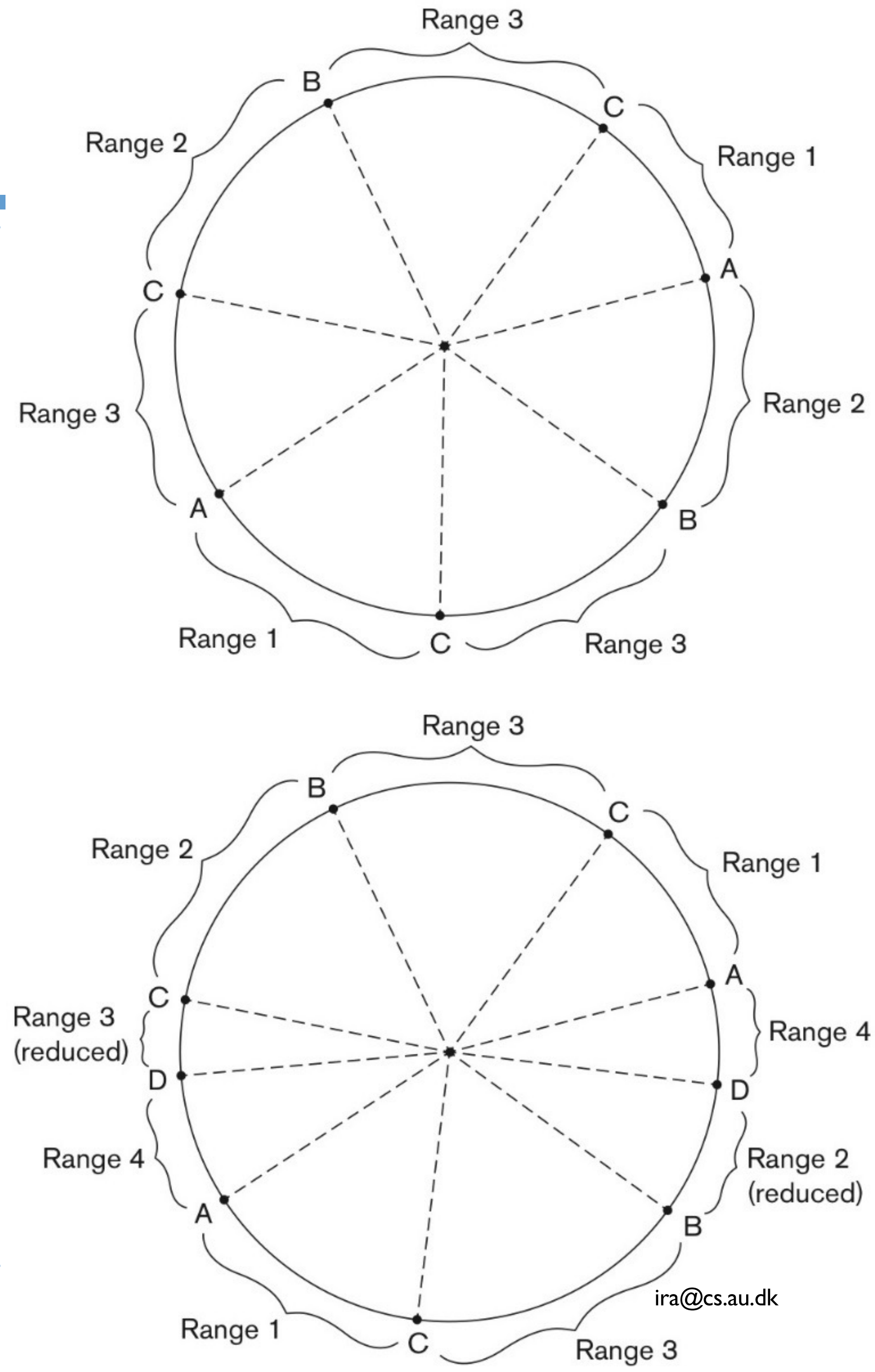
A. Add a range and a site
B. Add a site
C. Redistribute ranges to sites
D. Add rings



ira@cs.au.dk

Horizontal scalability

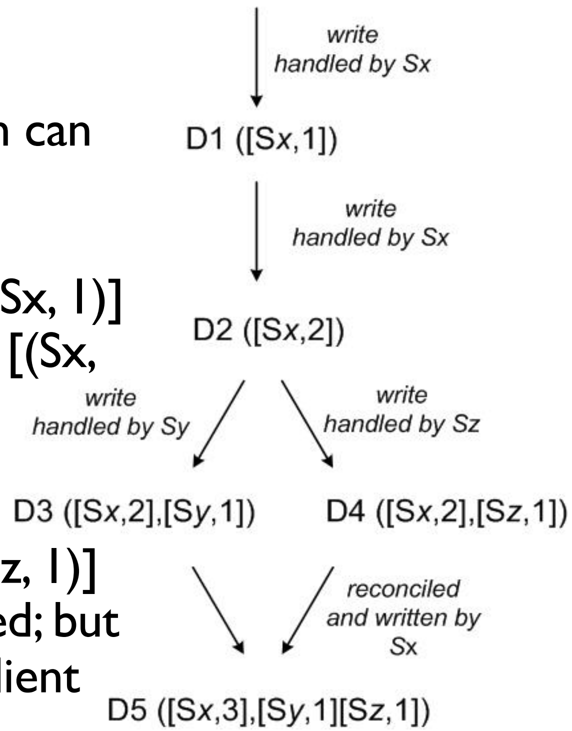
- Easy **horizontal scalability**
  - Add node to ring
    - E.g. add D
      - Items in range 4 are moved to D from B and C (ranges 2 and 3 reduced)



ira@cs.au.dk

Consistency and versioning

- Voldemort allows concurrent writes
  - Different values might be associated with the same key at different nodes when items replicated
  - When reading, achieve consistency by "versioning and read repair"
    - Each write associated with vector clock value
      - If system can reconcile different read values, do so
      - Otherwise, pass on several values to application which can reconcile based on application semantics
- Example:
  - Node Sx writes object D1 and produces associated clock [(Sx, 1)]
  - Sx then handles it again and produces version D2 and clock [(Sx, 2)]
  - D2 overwrites D1, but D1 copies may exist still
  - Node Sy updates D2 to D3 with clocks [(Sx, 2), (Sy, 1)]
  - Sz updates D2 to D4 (unaware of D3) with clock [(Sx, 2), (Sz, 1)]
  - Sx finds D1 and D2 are overwritten, can be garbage collected; but for D3 and D4 finds no relation, so both must be kept for client (upon a read) for semantic reconciliation



ira@cs.au.dk

Hbase

- Column-based / wide column system, similar to Google's BigTable
- Apache Hbase typically uses HDFS (Hadoop Distributed File System) for storage
- Sparse multidimensional distributed persistent sorted map
  - Map is a collection of (key, value) pairs, key is mapped to value
  - Main difference is the key: here, multidimensional, typically a combination of table name, row key, column, timestamp
  - Column typically consists of column family and column qualifier

https://hbase.apache.org/

ira@cs.au.dk

Hbase examples

- Table name followed by the names of the column families
  - E.g. Name
- Column qualifiers may group related columns for storage purposes when data is created (self-describing; not defined at creation time)
  - E.g. Fname

creating a table:

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

inserting some 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’

Some Hbase basic CRUD operations:

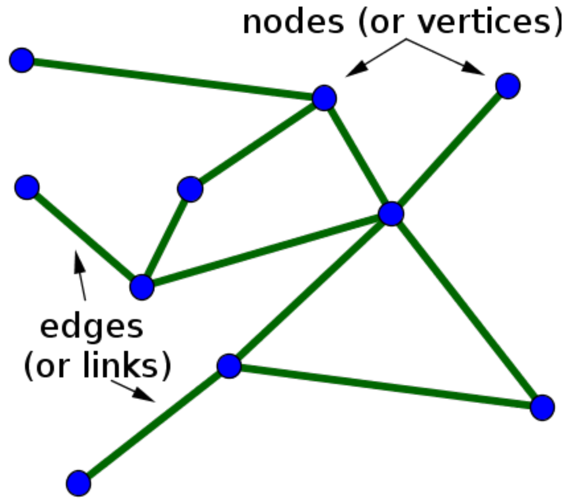
Creating a table: create <tablename>, <column family>, <column family>, …
Inserting Data: put <tablename>, <rowid>, <column family>:<column qualifier>, <value>
Reading Data (all data in a table): scan <tablename>
Retrieve Data (one item): get <tablename>,<rowid>

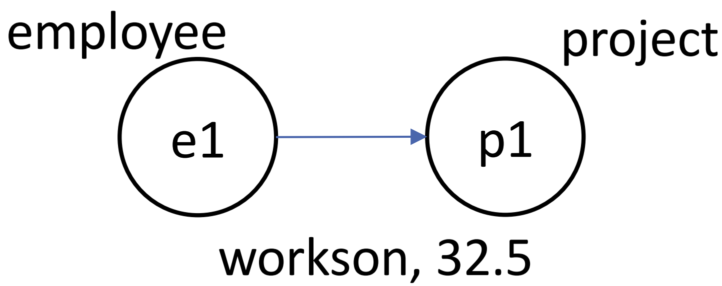
APACHE HBASE

ira@cs.au.dk

Neo4j

- Example graph oriented NoSQL system
- Data model: graph
  - Collection of vertices (nodes) and edges
    - Can be labeled to indicate the types of entities and relationships they represent
- Open source, Java
- High-level query language Cypher
  - Declarative commands for creating, modifying, and finding nodes and relationships



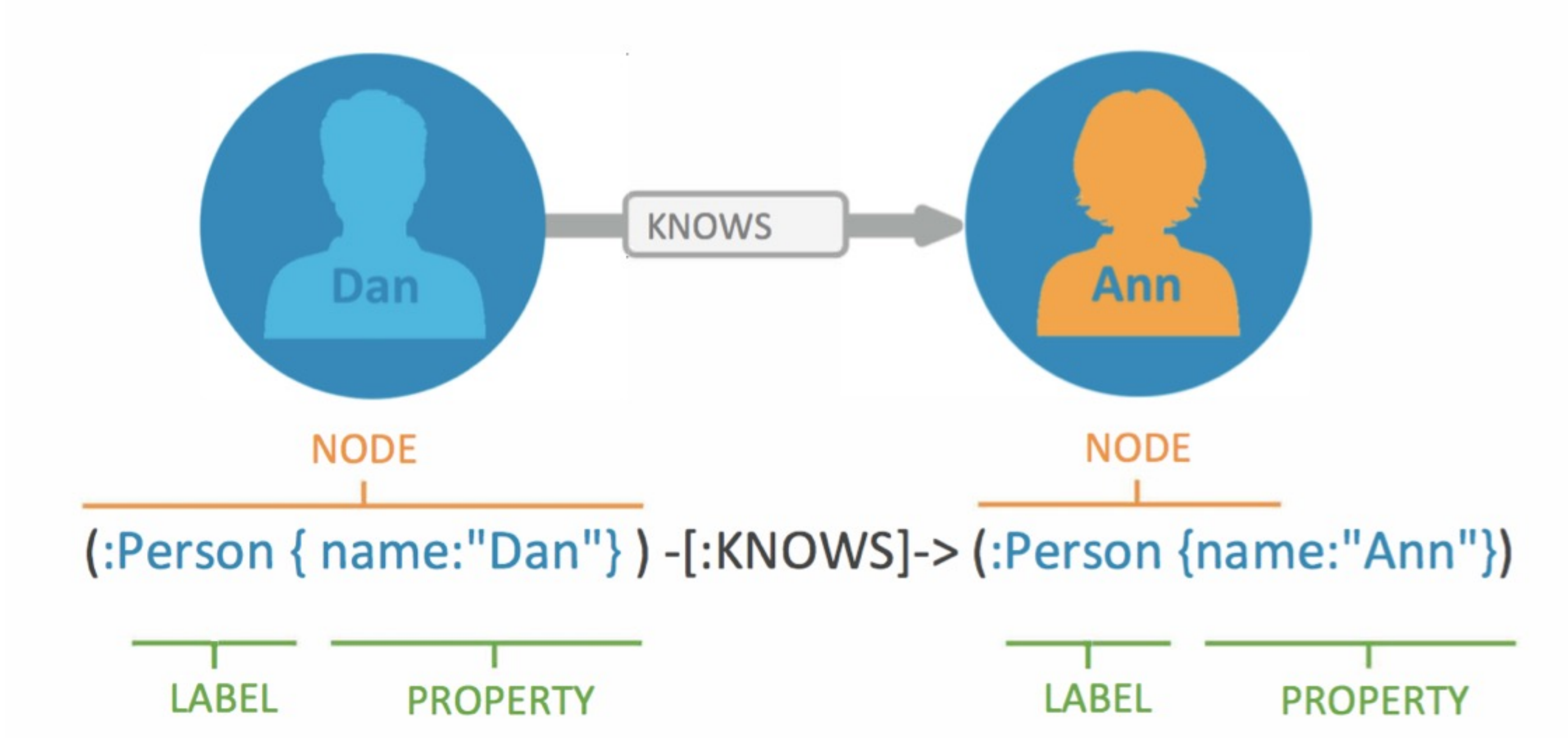


https://neo4j.com/docs/cypher-manual/current/
https://neo4j.com/docs/getting-started/current/cypher-intro/patterns/#cypher-intro-patterns

ira@cs.au.dk

Neo4j Graph Query Example

- Start from edge Knows between nodes with label Person and property Dan, Ann, resp.



ira@cs.au.dk

Neo4j Graph Query Example: Cypher

- Generalize by introducing Alias to second property of label Person
  - This generalized pattern is used in `MATCH` to define a query

ira@cs.au.dk

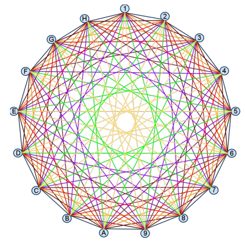
Graph Queries: CRUD

- **Create Read Update Delete** operations in Cypher query language
  - Similar to insert, select, update, delete in relational model
  - Examples:
    - Create new node with properties P
    - Add edge from v1 to v2 (plus some properties P)
    - Add property { Name :Value } to node v or to edge e
    - Add a new node, and then edges from it to other
    - Find nodes/edges with property P { Name :Value }
    - Find edges with a specific label

Operations on nodes and edges, and the patterns they are part of

ira@cs.au.dk

Example Cypher query



- Find nodes/edges with property P { Name :Value }
- Find edges with a specific label
- In Cypher:
  - MATCH pattern WHERE predicate RETURN expression

```
MATCH (node1:Person)-[:KNOWS]->
(node2:Person)
          (node1)-[:LIVES_IN]->(node3:City)
          (node2)-[:LIVES_IN]->(node3)
 WHERE node1.propertyA = {value1}
RETURN node2.propertyA, node2.propertyB
```


ira@cs.au.dk

Creating nodes for COMPANY

- EMPLOYEE, DEPARTMENT, etc. are node labels
  - Multiple labels possible
- {…} contain properties

```
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: 'Jennifer', 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'})
...
CREATE (loc1: LOCATION, {Lname: 'Houston'})
CREATE (loc2: LOCATION, {Lname: 'Stafford'})
CREATE (loc3: LOCATION, {Lname: 'Bellaire'})
CREATE (loc4: LOCATION, {Lname: 'Sugarland'})
...
```

Creating relationships

- CREATE syntax
  - → indicate direction of relationships
  - E.g. e1 works for d1
  - Relationship types are labels WorksFor, Manager,…

CREATE (e1) – [: WorksFor] –> (d1)
CREATE (e3) – [: WorksFor] –> (d2)
…
CREATE (d1) – [: Manager] –> (e2)
CREATE (d2) – [: Manager] –> (e4)
…
CREATE (d1) – [: LocatedIn] –> (loc1)
CREATE (d1) – [: LocatedIn] –> (loc3)
CREATE (d1) – [: LocatedIn] –> (loc4)
CREATE (d2) – [: LocatedIn] –> (loc2)
…
CREATE (e1) – [: WorksOn, {Hours: '32.5'}] –> (p1)
CREATE (e1) – [: WorksOn, {Hours: '7.5'}] –> (p2)
CREATE (e2) – [: WorksOn, {Hours: '10.0'}] –> (p1)
CREATE (e2) – [: WorksOn, {Hours: 10.0}] –> (p2)
CREATE (e2) – [: WorksOn, {Hours: '10.0'}] –> (p3)
CREATE (e2) – [: WorksOn, {Hours: 10.0}] –> (p4)

RELATIONSHIPS

(d1: DEPARTMENT, {Dno: '5', Dname: 'Research'})

↑ WorksFor

e1: EMPLOYEE, {Empid: '1', Lname: 'Smith', Fname: 'John', Minit: 'B'})

Queries

- Cypher queries made up of clauses
  - Result of one clause may be used as input to another
  - Similar to what we had seen for SQL (tables in, tables out)

Basic simplified syntax of some common Cypher clauses:

Finding nodes and relationships that match a pattern: MATCH <pattern>

Specifying aggregates and other query variables: WITH <specifications>

Specifying conditions on the data to be retrieved: WHERE <condition>

Specifying the data to be returned: RETURN <data>

Ordering the data to be returned: ORDER BY <data>

Limiting the number of returned data items: LIMIT <max number>

Creating nodes: CREATE <node, optional labels and properties>

Creating relationships: CREATE <relationship, relationship type and optional properties>

Deletion: DELETE <nodes or relationships>

Specifying property values and labels: SET <property values and labels>

Removing property values and labels: REMOVE <property values and labels>

ira@cs.au.dk

What could a relational equivalent be?

VOTE

neo4j

MATCH(e: EMPLOYEE {Empid:'2'}) – [w :WorksOn] → (p)

RETURN (e.Ename, w.Hours, p.Pname)

A. SELECT Ename,Hours,Pname FROM EMPLOYEE WHERE Empid='2';
B. SELECT Ename,Hours,Pname FROM EMPLOYEE, WORKSON WHERE Empid='2';
C. SELECT Ename,Hours,Pname FROM EMPLOYEE NATURAL JOIN WORKSON WHERE Empid='2';
D. SELECT Ename,Hours,Pname FROM EMPLOYEE NATURAL JOIN WORKSON NATURAL JOIN Projects WHERE Empid='2';

ira@cs.au.dk

Query examples

- MATCH specifies a pattern and query variables (d, loc)
- RETURN specifies query result

Examples of simple Cypher queries:

1. MATCH (d : DEPARTMENT {Dno: '5'}) – [: LocatedIn] → (loc)
   RETURN d.Dname , loc.Lname
2. MATCH (e: EMPLOYEE {Empid: '2'}) – [w: WorksOn] → (p)
   RETURN e.Ename , w.Hours, p.Pname
3. MATCH (e) – [w: WorksOn] → (p: PROJECT {Pno: 2})
   RETURN p.Pname, e.Ename , w.Hours
4. MATCH (e) – [w: WorksOn] → (p)
   RETURN e.Ename , w.Hours, p.Pname
   ORDER BY e.Ename
5. MATCH (e) – [w: WorksOn] → (p)
   RETURN e.Ename , w.Hours, p.Pname
   ORDER BY e.Ename
   LIMIT 10
6. MATCH (e) – [w: WorksOn] → (p)
   WITH e, COUNT(p) AS numOfprojs
   WHERE numOfprojs > 2
   RETURN e.Ename , numOfprojs
   ORDER BY numOfprojs
7. MATCH (e) – [w: WorksOn] → (p)
   RETURN e , w, p
   ORDER BY e.Ename
   LIMIT 10
8. MATCH (e: EMPLOYEE {Empid: '2'})
   SET e.Job = 'Engineer'

ira@cs.au.dk

Graph Query Example: OrientDB SQL

- OrientDB: Open source database written in Java (now owned by SAP)
  - multimodal, i.e., supports data models graph, document, key value and object

/* A and B are friends */

```
MATCH {class:Person, as:A} -FriendOf- {class:Person,
as:B},
```

/* C is a friend of both A ad B */

```
{as:A} -FriendOf- {as:C} -FriendOf- {as:B},
```

/* A has a cat */

```
{as:C} -HasA-> {class:Cat},
```

/* B works at AU */

```
{as:B} -WorksAt-> {class:Company,
where:(name=”AU")},
```

/* return the names of A and B with aliases name, friendName*/

```
RETURN A.name as name, B.name as friendName
```

https://orientdb.com/nosql/pattern-matching-with-orientdb/

ira@cs.au.dk

Summary

- Intended learning outcomes
- Be able to
  - Describe the data model and query language in NoSQL databases
  - Discuss differences in common NoSQL systems

ira@cs.au.dk

Where to go from here?

- We have seen NoSQL as an approach for high performance and availability of very large databases where immediate consistency and complex query languages are not the focus
- We will now turn to handling text documents, in particular from the web
  - Information Retrieval and Web

ira@cs.au.dk

What was this all about?

Guidelines for your own review of today's session

- NoSQL systems treat consistency as…
  - Which is different from relational DBMS where…
- NoSQL data models…
- Self-describing formats…
  - E.g. we have…
- NoSQL query languages are….
  - They typically provide operations: …
- Some types of NoSQL systems are…
- MongoDB uses the following….
- Neo4j instead models data as…
  - A query is expressed by…

ira@cs.au.dk