TDT4225
Chapter 2 – Data Models and Query Languages

Svein Erik Bratsberg
Department of Computer Science (IDI), NTNU

Kunnskap for en bedre verden

Relational model vs. document model

- SQL is based on Ted Codd's relational model (1970)
- Relations and tuples and the relational model
- Tables, columns and rows in SQL
- Business data processing, transaction processing, and batch processing
- The big idea with SQL was to hide implementation detail
- Older alternatives: hierarchical model, network model
- Alternative models: object-oriented databases, XML databases, JSON databases

NoSQL

- Not Only SQL
  - Key/value stores
  - Document-oriented databases (JSON)
  - Graph databases
  - Extended (nested) relational databases
- Performance for simple operations
- Scalability (sharding)
- Free and open source?
- Specialized query operations
- No schema?
- Frustration with SQL model. More dynamic and expressive model.

The object-relational mismatch

- Mismatch between application code and database model (objects vs rows)
- Object-Relational mapping (ORM, SQLAlchemy, Prisma, ActiveRecord and Hibernate) try to hide the difference
- Example LinkedIn resume (3 solutions in SQL):
  - Normalized: position, education, and contact information, using foreign keys
  - Structured datatypes/XML data/JSON inside rows. Querying and indexing these.
  - JSON/XML *documents* inside text columns containing Position, Education and Contact Information. Let application query and decode

Tables and foreign keys

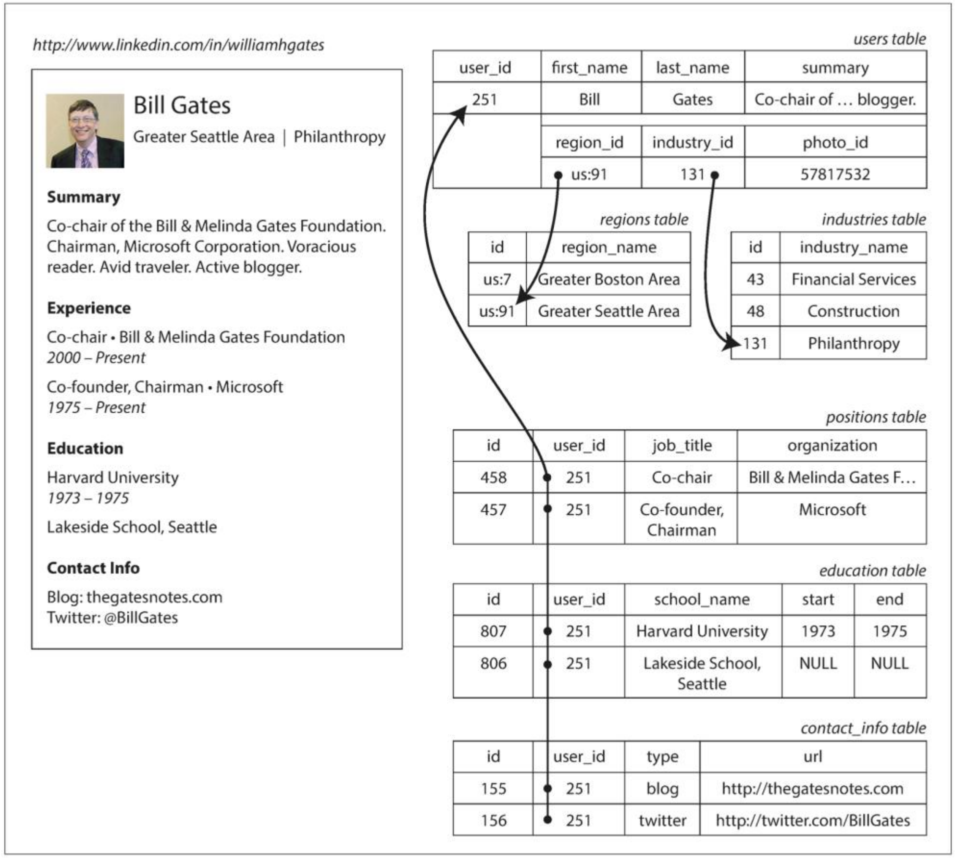

Figure 2-1. Representing a LinkedIn profile using a relational schema. Photo of Bill Gates courtesy of Wikimedia Commons, Ricardo Stuckert, Agência Brasil.

JSON

Example 2-1. Representing a LinkedIn profile as a JSON document

```json
{
  "user_id":     251,
  "first_name":  "Bill",
  "last_name":   "Gates",
  "summary":     "Co-chair of the Bill & Melinda Gates... Active blogger.",
  "region_id":   "us:91",
  "industry_id": 131,
  "photo_url":   "/p/7/000/253/05b/308dd6e.jpg",

  "positions": [
    {"job_title": "Co-chair", "organization": "Bill & Melinda Gates Foundation"},
    {"job_title": "Co-founder, Chairman", "organization": "Microsoft"}
  ],
  "education": [
    {"school_name": "Harvard University",       "start": 1973, "end": 1975},
    {"school_name": "Lakeside School, Seattle", "start": null, "end": null}
  ],
  "contact_info": {
    "blog":    "http://thegatesnotes.com",
    "twitter": "http://twitter.com/BillGates"
  }
}
```

JSON viewed as trees

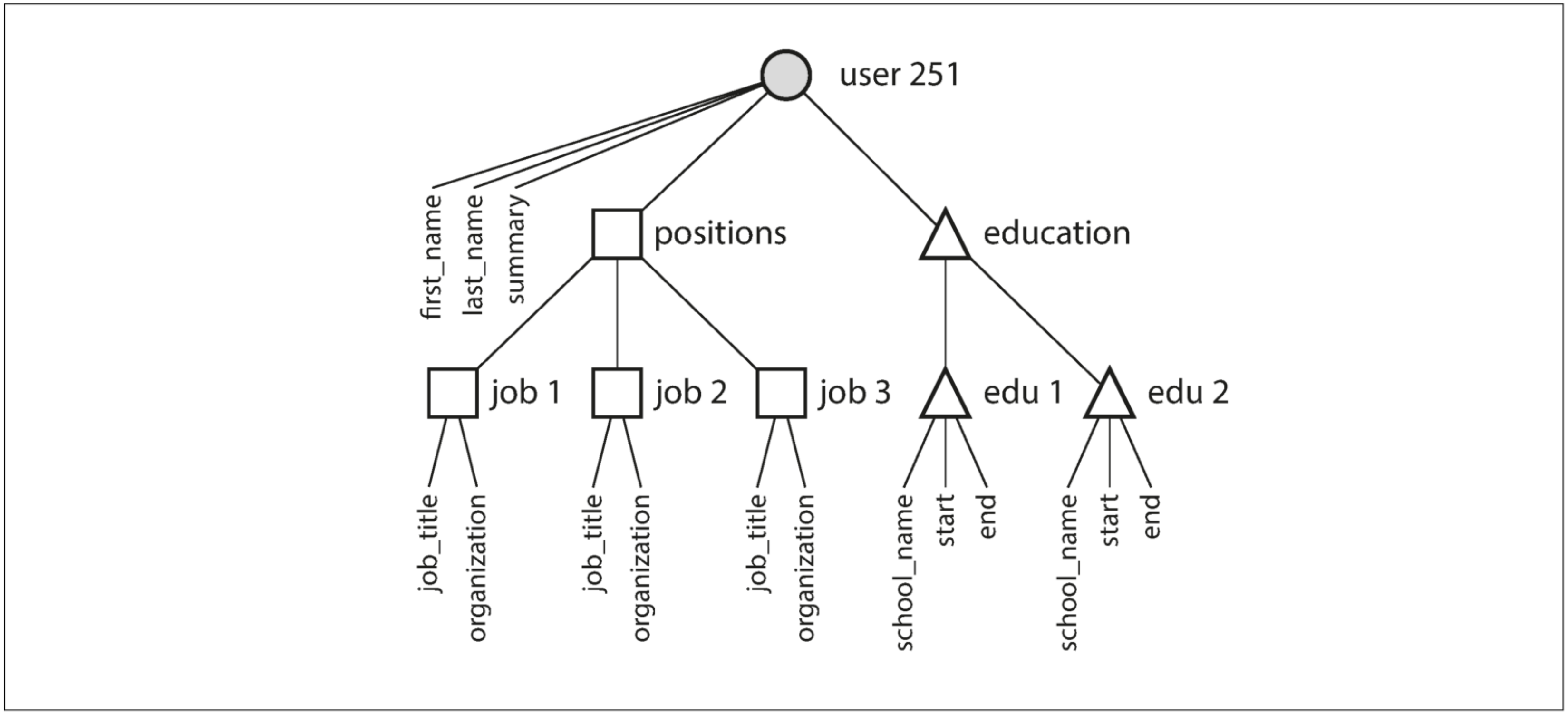

Figure 2-2. One-to-many relationships forming a tree structure.

NTNU

Many-to-one and many-to-many relationships

- Standardized lists of geo regions, industries, etc.
- Use IDs
  - Consistent style and spelling
  - Avoiding ambiguity
  - Ease of updating
  - Localization support (when translating between languages)
  - Better search
- Ids has no meaning to humans and don't need to change
- Relationships supported by joins in SQL databases
- Weakly supported in document databases (MongoDB/CouchDB)

Moving Organization to separate entity

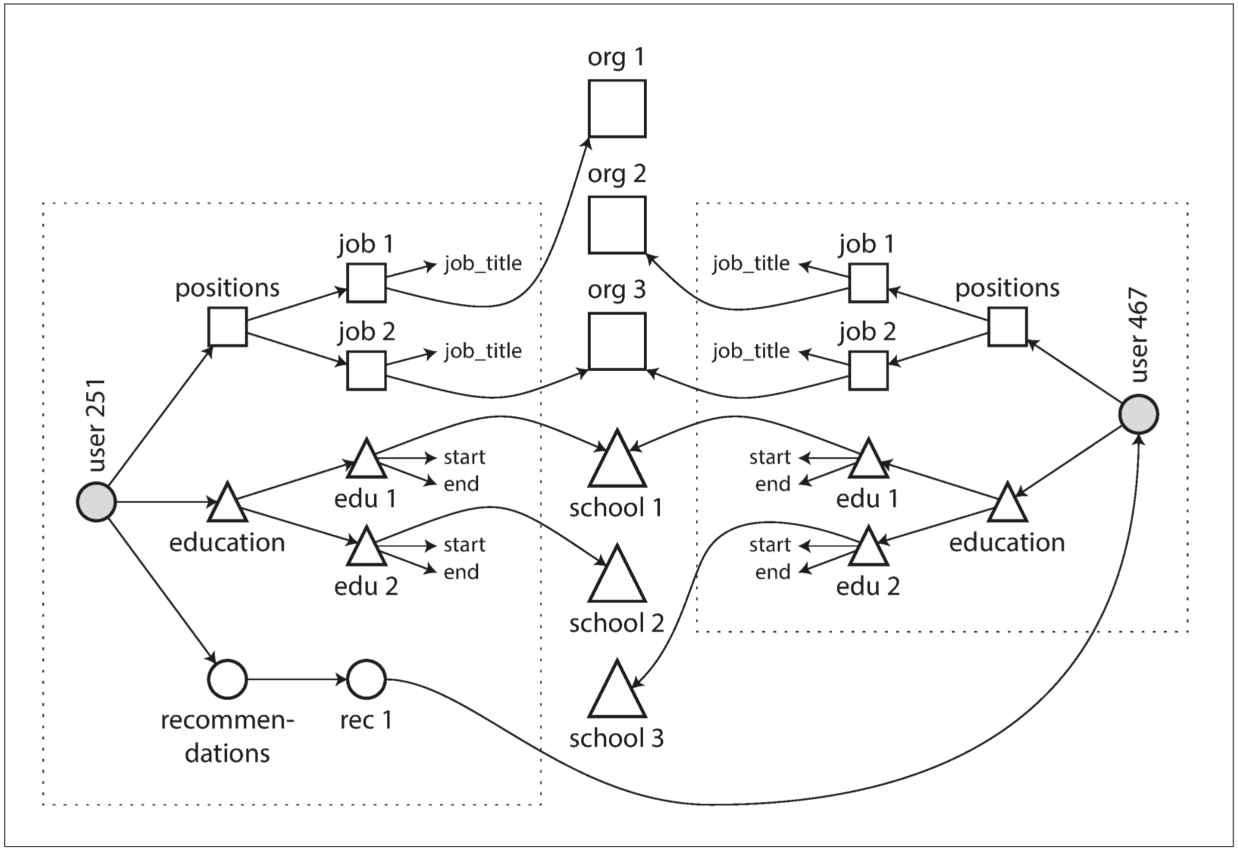

Figure 2-4. Extending résumés with many-to-many relationships.

NTNU

Hierarchical vs. codasyl vs. SQL

- IBM's IMS system was hierarchical and had the same problems as the document model
- CODASYL, the network model, allowed pointers between records. Access paths and cursors iterated over collections and pointers.
- SQL uses a query compiler and optimizer to automatically decide how to execute a query.
- SQL allows arbitrary relationships using joins

Document model vs. SQL

- Schema flexibility in document model
  - Schema-on-read, not schemaless
  - May support certain schema changes easily

```
if (user && user.name && !user.first_name) {
    // Documents written before Dec 8, 2013 don't have first_name
    user.first_name = user.name.split(" ")[0];
}
```

```
ALTER TABLE users ADD COLUMN first_name text;
UPDATE users SET first_name = split_part(name, ' ', 1);      -- PostgreSQL
UPDATE users SET first_name = substring_index(name, ' ', 1);      -- MySQL
```

- Suits document-like structures
- Bad support for joins
- Bad support for many-to-many relationships (Check links to MongoDB in exercise 3)

Schema-on-read and storage locality

- Good when there are many different types of objects
- Structure of objects determined by external systems (you have no control)
- When all objects are expected to have the same format, schema-on-read is not advantageous
- A document is usually stored as a single continuous string (JSON, BSON, XML).
- Gives locality when needing to access the whole document
- SQL databases have recently acquired support for XML and JSON. Making SQL DBs and document DBs similar.

Query languages for data

- Imperative query

```
function getSharks() {
    var sharks = [];
    for (var i = 0; i < animals.length; i++) {
        if (animals[i].family === "Sharks") {
            sharks.push(animals[i]);
        }
    }
    return sharks;
}
```

- Declarative query

```
SELECT * FROM animals WHERE family = 'Sharks';
```

- SQL gives room for optimizations and parallel execution, e.g. using indexes

Declarative queries on the web

- CSS

```css
li.selected > p {
    background-color: blue;
}
```

- XSL

```xml
<xsl:template match="li[@class='selected']/p">
    <fo:block background-color="blue">
        <xsl:apply-templates/>
    </fo:block>
</xsl:template>
```

- Javascript (DOM)

```javascript
var liElements = document.getElementsByTagName("li");
for (var i = 0; i < liElements.length; i++) {
    if (liElements[i].className === "selected") {
        var children = liElements[i].childNodes;
        for (var j = 0; j < children.length; j++) {
            var child = children[j];
            if (child.nodeType === Node.ELEMENT_NODE && child.tagName === "P") {
                child.setAttribute("style", "background-color: blue");
            }
        }
    }
}
```

MapReduce querying

- MapReduce is a query concept popularized by Google
- MongoDB also supports a form of MapReduce
- In SQL

```sql
SELECT date_trunc('month', observation_timestamp) AS observation_month,
       sum(num_animals) AS total_animals
FROM observations
WHERE family = 'Sharks'
GROUP BY observation_month;
```

- MongoDB

```javascript
db.observations.mapReduce(
    function map() { ❷
        var year  = this.observationTimestamp.getFullYear();
        var month = this.observationTimestamp.getMonth() + 1;
        emit(year + "-" + month, this.numAnimals); ❸
    },
    function reduce(key, values) { ❹
        return Array.sum(values); ❺
    },
    {
        query: { family: "Sharks" }, ❶
        out: "monthlySharkReport" ❻
    }
);
```

MapReduce querying (2)

- Map and reduce may only use data that is passed to them as input
- They cannot have side-effects
- «Low-level programming model» for distributed execution on a cluster of computers
- SQL may be run distributed and in parallel and may be optimized

Graph-like data models

- When many-to-many relationships are common, a graph model is appropriate
- Vertices and edges
  - Social graph
  - The web graph
  - Road and rail networks
- Multiple types of edge and nodes

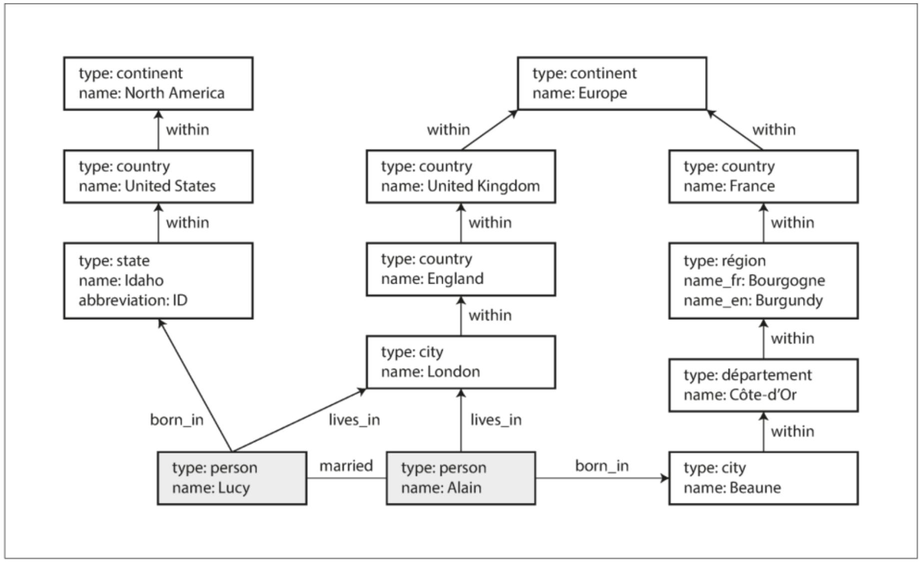

Figure 2-5. Example of graph-structured data (boxes represent vertices, arrows represent edges).

Graph-like data models (2)

- Property graph: Neo4j, Titan and InfiniteGraph
- Triple store: Datomic, AllegroGraph
- Query languages: Cypher, SPARQL and Datalog

Property graphs

- Vertex (id, outgoing edges, incoming edges, properties)
- Edge (id, tail vertex, head vertex, label, properties)
- Edges between any type of vertex (no restrictions)
- Easy to traverse the graph
- Labels give a rich modeling framework

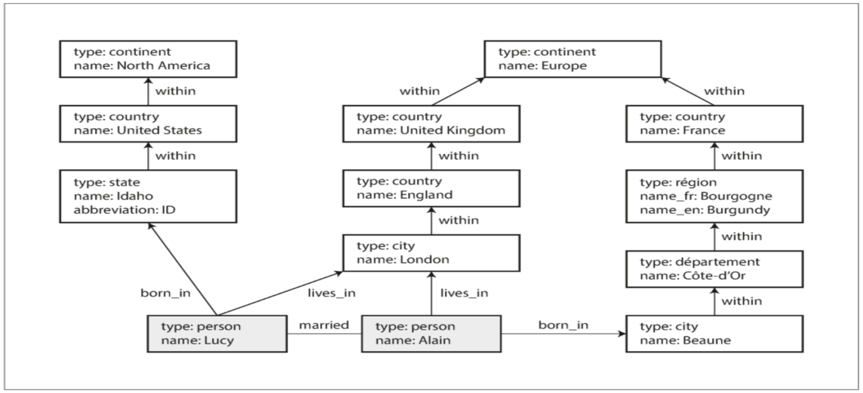

Figure 2-5. Example of graph-structured data (boxes represent vertices, arrows represent edges).

NTNU

The Cypher query language

- Neo4j's query language

Example 2-3. A subset of the data in Figure 2-5, represented as a Cypher query

```
CREATE
  (NAmerica:Location {name:'North America', type:'continent'}),
  (USA:Location      {name:'United States', type:'country'  }),
  (Idaho:Location    {name:'Idaho',         type:'state'    }),
  (Lucy:Person       {name:'Lucy' }),
  (Idaho) -[:WITHIN]->  (USA)  -[:WITHIN]-> (NAmerica),
  (Lucy)  -[:BORN_IN]-> (Idaho)
```

Example 2-4. Cypher query to find people who emigrated from the US to Europe

```
MATCH
  (person) -[:BORN_IN]->  () -[:WITHIN*0..]-> (us:Location {name:'United States'}),
  (person) -[:LIVES_IN]-> () -[:WITHIN*0..]-> (eu:Location {name:'Europe'})
RETURN person.name
```

- Query may be executed in several ways, starting at Person or Location?

Graph queries in SQL

- Variable number of joins to traverse a graph?
- WITH RECURSIVE introduced in SQL:1999

Example 2-5. The same query as Example 2-4, expressed in SQL using recursive common table expressions

```sql
WITH RECURSIVE

  -- in_usa is the set of vertex IDs of all locations within the United States
  in_usa(vertex_id) AS (
      SELECT vertex_id FROM vertices WHERE properties->>'name' = 'United States' ❶
    UNION
      SELECT edges.tail_vertex FROM edges ❷
        JOIN in_usa ON edges.head_vertex = in_usa.vertex_id
        WHERE edges.label = 'within'
  ),

  -- in_europe is the set of vertex IDs of all locations within Europe
  in_europe(vertex_id) AS (
      SELECT vertex_id FROM vertices WHERE properties->>'name' = 'Europe' ❸
    UNION
      SELECT edges.tail_vertex FROM edges
        JOIN in_europe ON edges.head_vertex = in_europe.vertex_id
        WHERE edges.label = 'within'
  ),

  -- born_in_usa is the set of vertex IDs of all people born in the US
  born_in_usa(vertex_id) AS ( ❹
    SELECT edges.tail_vertex FROM edges
      JOIN in_usa ON edges.head_vertex = in_usa.vertex_id
      WHERE edges.label = 'born_in'
  ),

  -- lives_in_europe is the set of vertex IDs of all people living in Europe
  lives_in_europe(vertex_id) AS ( ❺
    SELECT edges.tail_vertex FROM edges
      JOIN in_europe ON edges.head_vertex = in_europe.vertex_id
      WHERE edges.label = 'lives_in'
  )

SELECT vertices.properties->>'name'
FROM vertices
-- join to find those people who were both born in the US *and* live in Europe
JOIN born_in_usa     ON vertices.vertex_id = born_in_usa.vertex_id ❻
JOIN lives_in_europe ON vertices.vertex_id = lives_in_europe.vertex_id;
```


NTNU

Triple-Stores and SPARQL

- (subject, predicate, object), e.g. (Jim, likes, bananas)
- Subject corresponds to a vertex
- Object corresponds to
  - A primitive datavalue or
  - Another vertex

Example 2-6. A subset of the data in Figure 2-5,

```
@prefix : <urn:example:>.
_:lucy     a       :Person.
_:lucy     :name   "Lucy".
_:lucy     :bornIn _:idaho.
_:idaho    a       :Location.
_:idaho    :name   "Idaho".
_:idaho    :type   "state".
_:idaho    :within _:usa.
_:usa      a       :Location.
_:usa      :name   "United States".
_:usa      :type   "country".
_:usa      :within _:namerica.
_:namerica a       :Location.
_:namerica :name   "North America".
 :namerica :type   "continent".
```

Semantic web and RDF

- Semantic web describes machine readable data of the web
- RDF – Resource Description Framework

Example 2-8. The data of Example 2-7, expressed using RDF/XML syntax

```
<rdf:RDF xmlns="urn:example:"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">

  <Location rdf:nodeID="idaho">
    <name>Idaho</name>
    <type>state</type>
    <within>
      <Location rdf:nodeID="usa">
        <name>United States</name>
        <type>country</type>
        <within>
          <Location rdf:nodeID="namerica">
            <name>North America</name>
            <type>continent</type>
          </Location>
        </within>
      </Location>
    </within>
  </Location>

  <Person rdf:nodeID="lucy">
    <name>Lucy</name>
    <bornIn rdf:nodeID="idaho"/>
  </Person>
</rdf:RDF>
```

The SPARQL query language

- SPARQL is a query language for triple-stores using RDF

Example 2-9. The same query as Example 2-4, expressed in SPARQL

```
PREFIX : <urn:example:>

SELECT ?personName WHERE {
  ?person :name ?personName.
  ?person :bornIn  / :within* / :name "United States".
  ?person :livesIn / :within* / :name "Europe".
}
```

```
(person) -[:BORN_IN]-> () -[:WITHIN*0..]-> (location)   # Cypher

?person :bornIn / :within* ?location.                    # SPARQL
```

Datalog

- Old model based on predicate logic
- Predicate(subject, object)

Example 2-10. A subset of the data in Figure 2-5

```
name(namerica, 'North America').
type(namerica, continent).

name(usa, 'United States').
type(usa, country).
within(usa, namerica).

name(idaho, 'Idaho').
type(idaho, state).
within(idaho, usa).

name(lucy, 'Lucy').
born_in(lucy, idaho).
```

Example 2-11. The same query as Example 2-4, expressed in Datalog

```
within_recursive(Location, Name) :- name(Location, Name).     /* Rule 1 */

within_recursive(Location, Name) :- within(Location, Via),    /* Rule 2 */
                                    within_recursive(Via, Name).

migrated(Name, BornIn, LivingIn) :- name(Person, Name),       /* Rule 3 */
                                    born_in(Person, BornLoc),
                                    within_recursive(BornLoc, BornIn),
                                    lives_in(Person, LivingLoc),
                                    within_recursive(LivingLoc, LivingIn).

?- migrated(Who, 'United States', 'Europe').
/* Who = 'Lucy'. */
```

Datalog (2)

- A rule applies if the system can find a match for all predicates on the righthand side

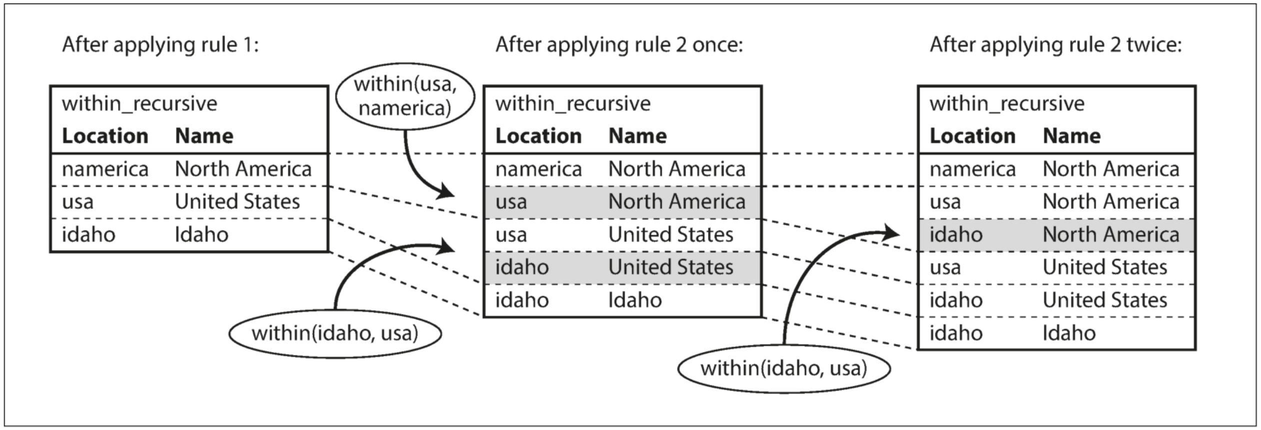

Figure 2-6. Determining that Idaho is in North America, using the Datalog rules from Example 2-11.

Document vs graph databases

- Document databases target use cases where data comes in self-contained documents and relationships between one document and another are rare.
- Graph databases go in the opposite direction, targeting use cases where anything is potentially related to everything.