Introduction to Data Management

Graph Databases

Alyssa Pittman
Based on slides by Jonathan Leang, Dan Suciu, et al

Paul G. Allen School of Computer Science and Engineering
University of Washington, Seattle
Graph Database
- Persists nodes and edges
- Data model
- Architecture
- Cypher query language
NoSQL Data Models

Key-Value Database

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>K1</td>
<td>AAA,BBB,CCC</td>
</tr>
<tr>
<td>K2</td>
<td>AAA,BBB</td>
</tr>
<tr>
<td>K3</td>
<td>AAA,DDD</td>
</tr>
<tr>
<td>K4</td>
<td>AAA,2,01/01/2015</td>
</tr>
<tr>
<td>K5</td>
<td>3,ZZZ,5623</td>
</tr>
</tbody>
</table>

Wide-Column Store (Extensible Record Store)

Graph Database

Document Store

XML

```
<empinfo>
  <employees>
    <employee>
      <name>James Kirk</name>
      <age>40</age>
    </employee>
    <employee>
      <name>Jean-Luc Picard</name>
      <age>45</age>
    </employee>
    <employee>
      <name>Wesley Crusher</name>
      <age>27</age>
    </employee>
  </employees>
</empinfo>
```

JSON

```
{  
  "empinfo" : {  
    "employees" : [  
      {  
        "name" : "James Kirk",  
        "age" : 40,  
      },  
      {  
        "name" : "Jean-Luc Picard",  
        "age" : 45,  
      },  
      {  
        "name" : "Wesley Crusher",  
        "age" : 27,  
      }  
    ] 
  }
}
```
## NoSQL Data Models

### Key-Value Database

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>K1</td>
<td>AAA,BBB,CCC</td>
</tr>
<tr>
<td>K2</td>
<td>AAA,BBB</td>
</tr>
<tr>
<td>K3</td>
<td>AAA,DDD</td>
</tr>
<tr>
<td>K4</td>
<td>AAA,2,01/01/2015</td>
</tr>
<tr>
<td>K5</td>
<td>3,ZZZ,5623</td>
</tr>
</tbody>
</table>

### Wide-Column Store (Extensible Record Store)

### Graph Database
- Entities and relationships
- “Unstructured graph”

### Document Store

#### XML
```
<empinfo>
    <employees>
        <employee>
            <name>James Kirk</name>
        </employee>
        <employee>
            <name>Jean-Luc Picard</name>
        </employee>
        <employee>
            <name>Wesley Crusher</name>
        </employee>
    </employees>
</empinfo>
```

#### JSON
```
{   "empinfo" : {     "employees" : [       {           "name" : "James Kirk",           "age" : 40,       },       {           "name" : "Jean-Luc Picard",           "age" : 45,       },       {           "name" : "Wesley Crusher",           "age" : 27,       }     ]   } }
```
# NoSQL Data Models

## Key-Value Database

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>K1</td>
<td>AAA,BBB,CCC</td>
</tr>
<tr>
<td>K2</td>
<td>AAA,BBB</td>
</tr>
<tr>
<td>K3</td>
<td>AAA,DDD</td>
</tr>
<tr>
<td>K4</td>
<td>AAA,2,01/01/2015</td>
</tr>
<tr>
<td>K5</td>
<td>3,ZZZ,5623</td>
</tr>
</tbody>
</table>

## Wide-Column Store (Extensible Record Store)

**UserProfile**

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key</td>
<td>Userprofile</td>
</tr>
<tr>
<td>First</td>
<td>John</td>
</tr>
<tr>
<td>Last</td>
<td>Smith</td>
</tr>
<tr>
<td>Age</td>
<td>30</td>
</tr>
<tr>
<td>Email</td>
<td><a href="mailto:john@smith.com">john@smith.com</a></td>
</tr>
<tr>
<td>Phone</td>
<td>1234567890</td>
</tr>
</tbody>
</table>

## Graph Database

- **neo4j**
- **Apache TinkerPop**

## Document Store

**XML**

```xml
<employees>
  <employee>
    <name>James Kirk</name>
    <age>40</age>
  </employee>
  <employee>
    <name>Jean-Luc Picard</name>
    <age>45</age>
  </employee>
  <employee>
    <name>Wesley Crusher</name>
    <age>27</age>
  </employee>
</employees>
```

**JSON**

```json
{   "employees": [   {     "name": "James Kirk",     "age": 40,   },   {     "name": "Jean-Luc Picard",     "age": 45,   },   {     "name": "Wesley Crusher",     "age": 27,   }   ]}
```
Graph Databases

- **Data model**
  - Entities (nodes)
  - Relationships (directed edges)
  - Both can have properties

![Graph Databases Diagram]

- **User name: Al**
  - KNOWS since: 1/1/1970
- **User name: Bob**
  - KNOWS since: 2/18/2018
- **User name: Carl**
  - KNOWS
Graph Databases

- Data model
  - Entities (nodes)
  - Relationships (directed edges)
    - Both can have types
    - Both can have properties

- Query language
  - CRUD API for data manipulation
  - No standard language for querying
    - GraphQL, Cypher, Gremlin, SPARQL...
Motivation

- Graphs in relational model
  - Foreign keys to represent edge, tables to represent “many” relationships
  - Compute joins on the fly to retrieve connected entities
  - Indexes can help, but for each edge followed, still need to look in index, then on disk
  - No recursive queries

CREATE TABLE Person(id INT PRIMARY KEY,
                     name TEXT);

CREATE TABLE PersonKnows(
    person INT REFERENCES(Person),
    knows INT REFERENCES(Person));
Motivation

- Graphs in relational model
  - Foreign keys to represent edge, tables to represent “many” relationships
  - Compute joins on the fly to retrieve connected entities
  - Indexes can help, but for each edge followed, still need to look in index, then on disk
  - No recursive queries

```sql
SELECT p3.name
FROM Person p1, PersonKnows pk1,
     PersonKnows pk2,
     PersonKnows pk3, Person p3
WHERE p1.name = "Bob" AND
  pk1.knows = pk2.person AND
  pk2.knows = pk3.person AND
  pk3.person = p3.id;
```
Motivation

- **Good use cases**
  - Heavily connected entities
  - \( n \) entities \( \rightarrow n^2 \) relationships
  - Number of relationships grows faster than entities
Use Cases

- Recommendation engines (Walmart)
  - Who else bought the same products as you? What else did they buy?
- Biological data (NIH)
  - How do genes, diseases, drug treatments, responses relate?
- Knowledge Graphs (NASA)
- ...
- Many problems can be modeled as graphs
5 W’s of Neo4j

▪ Who?
  • Neo company founded by Emil Eifrem, Johan Svensson

▪ What?
  • A native graph database
  • Commercial or open sourced versions

▪ When?
  • First version released in 2010

▪ Where?
  • Started in Sweden, headquartered in California

▪ Why?
  • Ran into performance problems with RDBMS
5 W’s of Cypher

- **Who?**
  - Andrés Taylor at the Neo company

- **What?**
  - A query language over the property graph data model

- **When?**
  - First version released in 2011
  - openCypher released in 2015

- **Where?**
  - Neo

- **Why?**
  - Want to easily, expressively query over graph data
Implementation

- **Native Implementation**
- **Storage:**
  - Nodes and relationships stored close together
- **Processing:**
  - Index-free adjacency
    - It stores the *physical* address of adjacent nodes
    - Instead of using a foreign key or looking in an index to see where the adjacent node is.
    - Query time doesn’t increase with the amount of data/relations you have!

![Diagram of relationships between persons](diagram.png)
Transactions

- Integrity constraints important for graph dbs
  - What happens if a relationship is updated but the nodes aren’t?
  - Graph is corrupted!
- Most modern graph databases fully ACID-compliant
- Basic Graph DBs are single nodes
- Neo4j has “causal consistency” mode
  - writes go to core servers
    - must be accepted by a majority of them
    - synced to replica servers
  - reads go to replica servers
• Basic Graph DBs are single nodes
• Neo4j has “causal consistency” mode
  • writes go to core servers
    • must be accepted by a majority of them
    • synced to replica servers
  • reads go to replica servers
  • some consistency:
    • when a client writes, gets a “bookmark” back
- Basic Graph DBs are single nodes
- Neo4j has “causal consistency” mode
  - writes go to core servers
    - must be accepted by a majority of them
    - synced to replica servers
  - reads go to replica servers
  - some consistency:
    - when a client writes, gets a “bookmark” back
    - client provides that bookmark for subsequent reads/writes
    - servers only respond if they already have the data from that bookmarked transaction

```
write v2
if bookmark1
  write!
else
  error!
```
Neo4j uses the Cypher query language

```cypher
MATCH (tom {name: "Tom Hanks"})
RETURN tom
```
MATCH patterns

- Match clause searches for *patterns* in the graph
  - Kind of like FROM/WHERE
  - It finds the subgraph of elements that match

Node pattern:

\[
(variable:Label \{propertyKey: 'propertyValue'\})
\]

Relationship pattern:

\[-[variable:TYPE]->\]

Put it all together:

\[(node1:LabelA) -[rel1:TYPE]->(node2:LabelB)\]
MATCH (tom:Person {name: "Tom Hanks"})
  -[:ACTED_IN]-(movie:Movie)
RETURN movie
MATCH patterns

- Directionality matters
  MATCH (a:Person)<-[[:ACTED_IN]]-(b:Movie)

- Sometimes you don’t care about direction
  MATCH (a:Person)--(b:Movie)

- Sometimes you don’t care about the relationship type
  MATCH (a:Person)<--(b:Movie)

- ...or the node type
  MATCH (a)<--(b:Movie)

- Can also match relationship properties
  MATCH (a:Person)<-[:ACTED_IN {role:"Santa Claus"}]- (b:Movie)

No matches: the edge goes the other direction
MATCH patterns

- If you need the matched element later, give it a variable:
  ```cypher
  MATCH (a:Person)<-[r:ACTED_IN]-(b:Movie)
  RETURN a, r, b
  ```

- VS
  ```cypher
  MATCH (a:Person)<-[[:ACTED_IN]]-(:Movie)
  RETURN a
  ```
MATCH patterns

- Can chain patterns:
  MATCH (:Person {name:"Tom Hanks"})
      -[:ACTED_IN]->(:Movie)<-[[:ACTED_IN]]-(co:Person)
  RETURN co

- or have multiple match clauses:
  MATCH (tom:Person {name:"Tom Hanks"})
  MATCH (tom)-[:ACTED_IN]->(m:Movie)
  MATCH (co:Person)-[:ACTED_IN]->(m)
  RETURN co
WHERE constraints

- Adds constraints to the MATCH pattern
  - Similar to a SQL WHERE
  - It can filter results
  - But adjusts the MATCH pattern instead of filtering after matching executes

```
MATCH (p:Person)-[:ACTED_IN]-> (:Movie {title:"Apollo 13" })
WHERE p.born < 1960
RETURN p
```

Can use:
- node or relationship properties
- boolean operators
- IN [...], EXISTS, IS NULL ...

And others you don’t need to know, such as subqueries
WHERE constraints

- Put the WHERE clause with the MATCH it adjusts
  - Can affect performance, and for some other clauses, correctness

- This is preferred:
  ```
  MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
  WHERE p.born < 1960
  MATCH (m {title:"Apollo 13" })
  RETURN p
  ```

- over this:
  ```
  MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
  MATCH (m {title:"Apollo 13" })
  WHERE p.born < 1960
  RETURN p
  ```
Neo4j uses the Cypher query language
  • Kind of like SELECT

Return nodes
  MATCH (a:Person)-[:ACTED_IN]->(b:Movie)
  RETURN a, b

Return relationships
  MATCH (:Person)-[r:ACTED_IN]->(:Movie)
  RETURN r

Return properties
  MATCH (a:Person)-[:ACTED_IN]->(:Movie)
  RETURN a.name
Some of our old friends

- Many familiar SQL clauses:

MATCH (:Person {name: "Tom Hanks"})
  -[:ACTED_IN]->(:Movie)
  <-[:ACTED_IN]-(co:Person)
RETURN DISTINCT co.name
ORDER BY co.name
LIMIT 10
- Can create a blank sandbox at https://neo4j.com/sandbox/
- Today we’ll play with data from their movie database walkthrough
MATCH with variable length

- Neo4j uses the Cypher query language
  - Kind of like SELECT

- Variable length relationship match pattern:
  - `[:TYPE*minHops..maxHops]`-

  minHops, maxHops are optional
  default to 1 and infinity, respectively
  dots can be omitted if want a single length
MATCH with variable length

- **Within three hops**
  
  ```
  MATCH (tom { name: 'Tom Hanks' })
      -[:ACTED_IN*1..3]-(movie:Movie)
  RETURN movie
  ```

- **Exactly two hops (like a SQL self-join!)**
  
  ```
  MATCH (tom { name: 'Tom Hanks' })
      -[:ACTED_IN*2]-(movie:Movie)
  RETURN movie
  ```

- **Any number of hops (warning: huge! why?)**
  
  ```
  MATCH (tom { name: 'Tom Hanks' })
      -[:ACTED_IN*1..]-(_)-[:ACTED_IN*1..]-(movie:Movie)
  RETURN movie
  ```
What if we want to see the paths themselves?

Give the match a variable:

```sql
MATCH p=(tom { name: 'Tom Hanks' })
    -[:ACTED_IN*1..3]-(movie:Movie)
RETURN p
```
Another nifty feature: find the shortest path between two nodes

Give the path a variable:
```
MATCH (tom:Person { name: 'Tom Hanks' }),
(kevin:Person { name: 'Kevin Bacon' }),
p = shortestPath((tom)-[*..15]-(kevin))
RETURN p
```
Takeaways

▪ Graphs can model relational data in intuitive ways
▪ Tradeoffs to prioritize following relationships lead to a different architecture and access pattern from relational databases