Introduction to
Graph Data Management
with neo4j

Tobias . Lindaaker@neo4j.com
@thobe #neo4j
Why graphs?
The world is a graph – everything is connected

• people, places, events
• companies, markets
• countries, history, politics
• life sciences, bioinformatics, clinical data
• art, teaching
• technology, networks, machines, applications, users
• software, code, dependencies, architecture, deployments
• criminals, fraudsters and their behavior
The topology of the data is at least as important as the data itself
Humans think in graphs

• We understand and learn by
  • how something new is similar to what we already know
  • how it differs
  • i.e. by relating things
    • in a graph!
What are people using graph databases for?
Use Cases

**Internal Applications**
- Master Data Management
- Network and IT Operations
- Fraud Detection

**Customer-Facing Applications**
- Real-Time Recommendations
- Graph-Based Search
- Identity and Access Management
Social Network
Logistics & Routing
Recommendations
Fraud Analysis
Modelling translations and complex access rules

EN: house
ES: casa
DE: Hause
SE: hus

EN: building
ES: edificio
DE: Gebäude
SE: byggnad
Whiteboard Friendliness

Easy to design and model, direct representation of the model
Whiteboard friendliness

- Tom Hanks
  - Acted in
    - Cloud Atlas
  - Directed
    - Lana Wachowskis

- Hugo Weaving
  - Acted in
    - The Matrix
Whiteboard friendliness

Tom Hanks

ACTED_IN

Cloud Atlas

ACTED_IN

The Matrix

DIRECTED

Hugo Weaving

DIRECTED

Lana Wachowski

ACTED_IN
Whiteboard friendliness

Person  Actor
name: Tom Hanks
born: 1956

ACTED_IN
roles: Zachry

ACTED_IN
roles: Bill Smoke

Movie

Title: Cloud Atlas
released: 2012

Person  Actor
name: Hugo Weaving
born: 1960

ACTED_IN
roles: Agent Smith

Movie

Title: The Matrix
released: 1999

Person  Director
name: Lana Wachowski
born: 1965
Whiteboard friendliness
Intro to the property graph model
Neo4j Fundamentals

- Nodes
- Relationships
- Properties
- Labels
Property Graph Model Components

**Nodes**
- Represent the objects in the graph
- Can be *labeled*
Property Graph Model Components

**Nodes**
- Represent the objects in the graph
- Can be *labeled*

**Relationships**
- Relate nodes by *type* and *direction*
Property Graph Model Components

Nodes
• Represent the objects in the graph
• Can be labeled

Relationships
• Relate nodes by type and direction

Properties
• Name-value pairs that can go on nodes and relationships.

- **Dan**
  - Name: “Dan”
  - Born: May 29, 1970
  - Twitter: “@dan”
  - Since: Jan 10, 2011
- **Ann**
  - Name: “Ann”
  - Born: Dec 5, 1975
  - Brand: “Volvo”
  - Model: “V70”

- **Car**
  - Since: Jan 10, 2011
  - Owns

- **LOVES**
- **LIVES WITH**
- **DRIVES**
- **OWNS**
Summary of the graph building blocks

- **Nodes** - Entities and complex value types
- **Relationships** - Connect entities and structure domain
- **Properties** - Entity attributes, relationship qualities, metadata
- **Labels** - Group nodes by role
Graph Querying
Why not SQL?

• SQL is inefficient in expressing graph pattern queries

  In particular for queries that
  • are recursive
  • can accept paths of multiple different lengths

• Graph patterns are more intuitive and declarative than joins

• SQL cannot handle path values
Cypher

A pattern matching query language made for graphs

- Declarative
- Expressive
- Pattern Matching
Pattern in our Graph Model

Dan \(\xrightarrow{LOVES}\) Ann

NODE \(\xrightarrow{Relationship}\) NODE
Cypher: Express Graph Patterns

(:Person { name:"Dan"} ) -[:LOVES]-> (:Person { name:"Ann"} )
Cypher: CREATE Graph Patterns

```
CREATE (
  :Person {
    name: "Dan"
  }
)-[:LOVES]->
  (:Person {
    name: "Ann"
  })
```

**NODE**  **Relationship**  **NODE**

**LABEL**  **PROPERTY**  **LABEL**  **PROPERTY**
MATCH ( :Person { name: "Dan" } ) -[:LOVES]-> ( whom ) RETURN whom
A graph query example
A social recommendation
MATCH (person:Person)-[:IS_FRIEND_OF]->(friend),
  (friend)-[:LIKES]->(restaurant),
  (restaurant)-[:LOCATED_IN]->(loc:Location),
  (restaurant)-[:SERVES]->(type:Cuisine)
WHERE person.name = 'Philip'
AND loc.location='New York'
AND type.cuisine='Sushi'
RETURN restaurant.name
The Syntax
Nodes

Nodes are drawn with parentheses.

()
Relationships

Relationships are drawn as arrows, with additional detail in brackets.

\[ \rightarrow \]

\[-[\text{:DIRECTED}]\rightarrow\]
Patterns are drawn by connecting nodes and relationships with hyphens, optionally specifying a direction with > and < signs.

\[
(\text{()}) - [\text{[]}] - (\text{()})
\]

\[
(\text{()}) - [\text{[]}] - > (\text{()})
\]

\[
(\text{()}) <- [\text{[]}] - (\text{()})
\]
The components of a Cypher query

MATCH (m:Movie)
RETURN m

MATCH and RETURN are Cypher keywords
m is a variable
:Movie is a node label
The components of a Cypher query

MATCH (p:Person)-[r:ACTED_IN]->(m:Movie)
RETURN p, r, m

MATCH and RETURN are Cypher keywords
p, r, and m are variables
:Movie is a node label
:ACTED_IN is a relationship type
The components of a Cypher query

MATCH  path = (:Person)-[:ACTED_IN]->(:Movie)
RETURN  path

MATCH and RETURN are Cypher keywords
path is a variable
:Movie is a node label
:ACTED_IN is a relationship type
Graph versus Tabular results

MATCH (m:Movie)
RETURN m
MATCH (m:Movie)
RETURN m.title, m.released

Properties are accessed with `{variable}.{property_key}`
Case sensitivity

Case sensitive

Node labels
Relationship types
Property keys

Case insensitive

Cypher keywords
Case sensitivity

Case sensitive

:Person

:ACTED_IN

name

Case insensitive

MaTcH

return
Write queries
The CREATE Clause

CREATE (m:Movie {title:'Mystic River', released:2003})
RETURN m
The SET Clause

MATCH (m:Movie {title: 'Mystic River'})
SET m.tagline = 'We bury our sins here, Dave. We wash them clean.'
RETURN m
The CREATE Clause

MATCH (m:Movie {title: 'Mystic River'})
MATCH (p:Person {name: 'Kevin Bacon'})
CREATE (p)-[r:ACTED_IN {roles: ['Sean']}]-(m)
RETURN p, r, m
The MERGE Clause

MERGE (p:Person {name: 'Tom Hanks'})
RETURN p
The MERGE Clause

MERGE (p:Person {name: 'Tom Hanks', oscar: true})
RETURN p
The MERGE Clause

```mermaid
MERGE (p:Person {name: 'Tom Hanks', oscar: true})
RETURN p
```

There is not a :Person node with name:'Tom Hanks' and oscar:true in the graph, but there is a :Person node with name:'Tom Hanks'.

What do you think will happen here?
The MERGE Clause

MERGE (p:Person {name: 'Tom Hanks'})
SET p.oscar = true
RETURN p
The MERGE Clause

MERGE (p:Person {name: 'Tom Hanks'})-[[:ACTED_IN]]->(m:Movie {title: 'The Terminal'})

RETURN p, m
The MERGE Clause

MERGE (p:Person {name: 'Tom Hanks'})-[:ACTED_IN]-(m:Movie {title: 'The Terminal'})
RETURN p, m

There is not a :Movie node with title:"The Terminal" in the graph, but there is a :Person node with name:"Tom Hanks".

What do you think will happen here?
The MERGE Clause

MERGE (p:Person {name: 'Tom Hanks'})
MERGE (m:Movie {title: 'The Terminal'})
MERGE (p)-[r:ACTED_IN]->(m)
RETURN p, r, m
ON CREATE and ON MATCH

MERGE (p:Person {name: 'Your Name'})

ON CREATE SET p.created = timestamp(), p.updated = 0
ON MATCH SET p.updated = p.updated + 1
RETURN p.created, p.updated;
Indexes

We create indexes to:

- allow fast lookup of nodes which match label-property pairs.

```sql
CREATE INDEX ON :Label(property)
```
What are these fast lookups?

The following predicates use indexes:

- Equality
- \textsc{starts with}
- \textsc{contains}
- \textsc{ends with}
- Range searches
- (Non-) existence checks
How are indexes used in neo4j?

Indexes are **only** used to find the **starting points** for queries.

Use index scans to look up rows in tables and join them with rows from other tables.

Use indexes to find the starting points for a query.
Graph Modeling
Models
The modeling workflow

1. Derive the question
2. Obtain the data
3. Develop a model
4. Ingest the data
5. Query/Prove our model
Developing the model and the query

1. Identify application/end-user goals
2. Figure out what questions to ask of the domain
3. Identify entities in each question
4. Identify relationships between entities in each question
5. Convert entities and relationships to paths
   - These become the basis of the data model
6. Express questions as graph patterns
   - These become the basis for queries
1. Application/End-User Goals

As an employee
I want to know who in the company has similar skills to me
So that we can exchange knowledge
2. Questions to ask of the Domain

As an employee
I want to know who in the company has similar skills to me
So that we can exchange knowledge

Which people, who work for the same company as me, have similar skills to me?
3. Identify Entities

Which *people*, who work for the same *company* as me, have similar *skills* to me?

- Person
- Company
- Skill
4. Identify Relationships Between Entities

Which people, who work for the same company as me, have similar skills to me?

- Person WORKS FOR Company
- Person HAS SKILL Skill
5. Convert to Cypher Paths

- Person WORKS FOR Company
- Person HAS SKILL Skill
5. Convert to Cypher Paths

- **Person WORKS FOR Company**
- **Person HAS SKILL Skill**
5. Convert to Cypher Paths

- **Person** WORKS FOR **Company**
- **Person** HAS SKILL **Skill**

- (:Person)-[:WORKS_FOR]->(:Company),
- (:Person)-[:HAS_SKILL]->(:Skill)
5. Convert to Cypher Paths

- **Person** WORKS FOR **Company**

- **Person** HAS **SKILL** **Skill**

- (:Person)-[:WORKS_FOR]->(:Company),
- (:Person)-[:HAS_SKILL]->(:Skill)
Consolidate Pattern

(:Person)-[:WORKS_FOR]->(:Company),
(:Person)-[:HAS_SKILL]->(:Skill)

(:Company)<-[::WORKS_FOR]-(:Person)-[:HAS_SKILL]->(:Skill)
Candidate Data Model

(:Company)<-[:WORKS_FOR]-(:Person)-[:HAS_SKILL]->(:Skill)
Which people, who work for the same company as me, have similar skills to me?
Cypher Query

Which people, who work for the same company as me, have similar skills to me?

MATCH (company)<-[[:WORKS_FOR]]-(me:Person)-[:HAS_SKILL]->(skill)
  (company)<-[[:WORKS_FOR]]-(colleague)-[:HAS_SKILL]->(skill)
WHERE me.name = $name
RETURN colleague.name AS name,
  count(skill) AS score,
  collect(skill.name) AS skills
ORDER BY score DESC
Cypher Query

Which people, who work for the same company as me, have similar skills to me?

```
MATCH (company)<-[::WORKS_FOR]-(me:Person)-[::HAS_SKILL]->(skill)
  (company)<-[::WORKS_FOR]-(colleague)-[::HAS_SKILL]->(skill)
WHERE me.name = $name
RETURN colleague.name AS name,
    count(skill) AS score,
    collect(skill.name) AS skills
ORDER BY score DESC
```

I. Graph pattern
Cypher Query

Which people, who work for the same company as me, have similar skills to me?

MATCH (company)<-[[:WORKS_FOR]]-(me:Person)-[:HAS_SKILL]->(skill)
  (company)<-[[:WORKS_FOR]]-(colleague)-[:HAS_SKILL]->(skill)
WHERE me.name = $name
RETURN colleague.name AS name,
    count(skill) AS score,
    collect(skill.name) AS skills
ORDER BY score DESC

1. Graph pattern
2. Filter, using index if available
Cypher Query

Which people, who work for the same company as me, have similar skills to me?

MATCH (company)<-[:WORKS_FOR]-(me:Person)-[:HAS_SKILL]->(skill)
  (company)<-[:WORKS_FOR]-(colleague)-[:HAS_SKILL]->(skill)
WHERE me.name = $name
RETURN colleague.name AS name,
  count(skill) AS score,
  collect(skill.name) AS skills
ORDER BY score DESC

1. Graph pattern
2. Filter, using index if available
3. Create projection of result
First Match

Person
  name: Tobias
  WORKS_FOR
  Person
  name: Ian
  WORKS_FOR
  Company
  name: ACME
  HAS_SKILL
  Skill
  name: Scala
  HAS_SKILL
  Person
  name: Jacob
  WORKS_FOR
  Person
  name: Tobias
  HAS_SKILL
  Skill
  name: C#
  HAS_SKILL
  Person
  name: Tobias
  HAS_SKILL
  Skill
  name: Neo4j
  HAS_SKILL
  Person
  name: Tobias
  HAS_SKILL
  Skill
  name: Python
  HAS_SKILL
  Person
  name: Tobias
  HAS_SKILL
  Skill
  name: Scala
  HAS_SKILL
  Person
  name: Tobias
  HAS_SKILL
  Skill
  name: C#
  HAS_SKILL
  Person
  name: Tobias
  HAS_SKILL
  Skill
  name: Neo4j
  HAS_SKILL
  Person
  name: Tobias
  HAS_SKILL
  Skill
  name: Python
  HAS_SKILL
  Person
  name: Tobias
  HAS_SKILL
  Skill
  name: Scala
  HAS_SKILL
  Person
  name: Tobias
  HAS_SKILL
  Skill
  name: C#
  HAS_SKILL
  Person
  name: Tobias
  HAS_SKILL
  Skill
  name: Neo4j
  HAS_SKILL
  Person
  name: Tobias
  HAS_SKILL
  Skill
  name: Python
  HAS_SKILL
  Person
  name: Tobias
  HAS_SKILL
  Skill
  name: Scala
  HAS_SKILL
  Person
  name: Tobias
  HAS_SKILL
  Skill
  name: C#
  HAS_SKILL
  Person
  name: Tobias
  HAS_SKILL
  Skill
  name: Neo4j
  HAS_SKILL
  Person
  name: Tobias
  HAS_SKILL
  Skill
  name: Python
  HAS_SKILL
  Person
  name: Tobias
  HAS_SKILL
  Skill
  name: Scala
  HAS_SKILL
  Person
  name: Tobias
  HAS_SKILL
  Skill
  name: C#
  HAS_SKILL
  Person
  name: Tobias
  HAS_SKILL
  Skill
  name: Neo4j
  HAS_SKILL
  Person
  name: Tobias
  HAS_SKILL
  Skill
  name: Python
  HAS_SKILL
  Person
  name: Tobias
  HAS_SKILL
  Skill
  name: Scala
  HAS_SKILL
  Person
  name: Tobias
  HAS_SKILL
  Skill
  name: C#
  HAS_SKILL
  Person
  name: Tobias
  HAS_SKILL
  Skill
  name: Neo4j
  HAS_SKILL
  Person
  name: Tobias
  HAS_SKILL
  Skill
  name: Python
  HAS_SKILL
  Person
  name: Tobias
  HAS_SKILL
  Skill
  name: Scala
  HAS_SKILL
  Person
  name: Tobias
  HAS_SKILL
  Skill
  name: C#
  HAS_SKILL
  Person
  name: Tobias
  HAS_SKILL
  Skill
  name: Neo4j
  HAS_SKILL
  Person
  name: Tobias
  HAS_SKILL
  Skill
  name: Python
  HAS_SKILL
  Person
  name: Tobias
  HAS_SKILL
  Skill
  name: Scala
  HAS_SKILL
  Person
  name: Tobias
  HAS_SKILL
  Skill
  name: C#
  HAS_SKILL
  Person
  name: Tobias
  HAS_SKILL
  Skill
  name: Neo4j
  HAS_SKILL
  Person
  name: Tobias
  HAS_SKILL
  Skill
  name: Python
  HAS_SKILL
  Person
  name: Tobias
  HAS_SKILL
  Skill
  name: Scala
  HAS_SKILL
  Person
  name: Tobias
  HAS_SKILL
  Skill
  name: C#
  HAS_SKILL
  Person
  name: Tobias
  HAS_SKILL
  Skill
  name: Neo4j
  HAS_SKILL
  Person
  name: Tobias
  HAS_SKILL
  Skill
  name: Python
  HAS_SKILL
  Person
  name: Tobias
  HAS_SKILL
  Skill
Second Match

Person
name: Tobias

Person
name: Ian

Person
name: Jacob

Person
name: Tobias

Person
name: Ian

Person
name: Jacob

Company
name: ACME

Skill
name: Scala

Skill
name: C#

Company
name: ACME

Skill
name: Python

Skill
name: Neo4j

Skill
name: C#

Skill
name: Scala
Third Match
### Result of the Query

<table>
<thead>
<tr>
<th>name</th>
<th>score</th>
<th>skills</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Ian&quot;</td>
<td>2</td>
<td>[&quot;Scala&quot;,&quot;Neo4j&quot;]</td>
</tr>
<tr>
<td>&quot;Jacob&quot;</td>
<td>1</td>
<td>[&quot;Neo4j&quot;]</td>
</tr>
</tbody>
</table>

2 rows
Modeling exercise: Movie genres
Adding movie genres

The question: should we model them as properties or as nodes?
MATCH (m:Movie {title: 'The Matrix'})
SET m.genre = ['Action', 'Sci-Fi']
RETURN m
MATCH (m:Movie {title: 'Mystic River'})
SET m.genre = ['Action', 'Mystery']
RETURN m
The good side of properties

Accessing a movie’s genres is quick and easy.

MATCH (m:Movie {title:"The Matrix"})
RETURN m.genre;
Finding movies that share genres is painful and we have a disconnected pattern in the MATCH clause - a sure sign you have a modeling issue.

MATCH (m1:Movie), (m2:Movie)
WHERE any(x IN m1.genre WHERE x IN m2.genre)
AND m1 <> m2
RETURN m1, m2;
Genres as nodes

MATCH (m:Movie {title:"The Matrix"})
MERGE (action:Genre {name:"Action"})
MERGE (scifi:Genre {name:"Sci-Fi"})
MERGE (m)-[:IN_GENRE]->(action)
MERGE (m)-[:IN_GENRE]->(scifi)
Genres as nodes

MATCH (m:Movie {title: "Mystic River"})
MERGE (action:Genre {name: "Action"})
MERGE (mystery:Genre {name: "Mystery"})
MERGE (m)-[:IN_GENRE]->(action)
MERGE (m)-[:IN_GENRE]->(mystery)
The good side of nodes

Finding movies that share genres is a natural graph pattern.

MATCH (m1:Movie)-[:IN_GENRE]->(g:Genre),
    (m2:Movie)-[:IN_GENRE]->(g)
RETURN m1, m2, g
The (not too) bad side of nodes

Accessing the genres of movies requires a bit more typing.

```
MATCH (m:Movie {title: "The Matrix"}),
     (m)-[:IN_GENRE]-(g:Genre)
RETURN g.name;
```
Generic vs Specific Relationship types
Relationship Granularity

```
code: LAS
Airport

CONNECTED TO

number: 335
Flight

CONNECTED TO

code: LAX
Airport

code: LAS
Airport

FLIGHT_FROM_LAS

number: 335
Flight

FLIGHT_TO_LAX

code: LAX
Airport
```
Symmetric Relationships
Symmetric relationships

\[
\text{name: Sarah} \quad \text{Person} \quad \text{PARENT_OF} \quad \text{CHILD_OF} \quad \text{Person} \quad \text{name: Eric}
\]

\[
\text{OR}
\]

\[
\text{name: Sarah} \quad \text{Person} \quad \text{PARENT_OF} \quad \text{Person} \quad \text{name: Eric}
\]

\[
\text{OR}
\]

\[
\text{name: Sarah} \quad \text{Person} \quad \text{CHILD_OF} \quad \text{Person} \quad \text{name: Eric}
\]
Bidirectional Relationships
Use single relationship and ignore direction in queries

MATCH (:Person {name:'Eric'})-[[:MARRIED_TO]]-(p2)
RETURN p2
openCypher

the standard

graph query language
Becoming an industry standard for Graph Querying

• Cypher originally designed by Neo4j

• Implemented and supported by
  • SAP HANA
  • Redis
  • Bitnine AgensGraph
  • Gradoop

• More systems coming
  • Spark
openCypher

- Open language design process
- Implementations for multiple systems
- Compatibility test suite
- Grammar specification
- Reference implementation
- Defining the next version of Cypher

https://github.com/openCypher/openCypher
http://opencypher.org
Formal semantics

- Nadime Francis, Paolo Guagliardo, Leonid Libkin
- Formally defines a (large) core of Cypher
Future improvements
Regular Path Queries

• Conjunctive bi-directional Regular Path Queries with Data
  • Plus specification of a cost function for paths, which allows for more interesting notions of *shortest path*

```
PATH PATTERN coauth=()-[:WROTE]->(b)<-[:WROTE]-(),
    (b)<-[sale:SELLS]-()
COST min(sale.price)
```

```
MATCH (a)-/~coauth* COST x/->(b)
ORDER BY x LIMIT 10
```
Further improvements

• Support for querying from multiple graphs
• Support for returning graphs
• Support for defining views

• Integration with SQL
Relational to Graph
Relational to Graph

Relational is simple...until it gets complicated...
Relational to Graph

You know relational...now consider relationships...
Normalized Relational Model to Graph Model

- Entity-Tables become Nodes
- Foreign Keys become Relationships
- Link Tables become Relationships
- Remove artificial Primary Keys and Foreign Keys
RDBMS can’t handle relationships well

- Cannot model or store data and relationships without complexity
- Performance degrades with number and levels of relationships, and database size
- Query complexity grows with need for JOINs
- Adding new types of data and relationships requires schema redesign, increasing time to market
Express Complex Queries Easily with Cypher

Find all managers and how many people they manage, up to 3 levels down.

**Cypher**

```cypher
MATCH (boss)-[:MANAGES*0..3]->(mgr)
WHERE boss.name = "John Doe"
AND (mgr)-[:MANAGES]->()
RETURN mgr.name AS Manager,
    size((mgr)-[:MANAGES*1..3]->()) AS Total
```

**SQL**

```
SELECT Manager, COUNT(*) AS Total
FROM employees
WHERE boss_name = 'John Doe'
    AND manager_name IS NOT NULL
GROUP BY manager_name
```
Unlocking Value from Your Data Relationships

- Model your data **naturally** as a graph of data and relationships
- Drive graph model **from domain and use-cases**
- Use relationship information in **real-time** to transform your business
- Add new relationships **on the fly** to adapt to your changing requirements
High Query Performance with a Native Graph DB

- Relationships are **first class citizen**
- No need for joins, just follow pre-materialized relationships of nodes
- Query & Data-**locality** – navigate out from your starting points
- Only load what’s needed
- Aggregate and project results as you go
- Optimized disk and memory model for graphs
The performance advantage of materialised relationships

- a sample social graph with ~1,000 persons
- average 50 friends per person
- pathExists(a,b) limited to depth 4
- caches warmed up to eliminate disk I/O

<table>
<thead>
<tr>
<th></th>
<th># persons</th>
<th>query time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relational database</td>
<td>1,000</td>
<td>2000ms</td>
</tr>
<tr>
<td>Neo4j</td>
<td>1,000</td>
<td>2ms</td>
</tr>
<tr>
<td>Neo4j</td>
<td>1,000,000</td>
<td>2ms</td>
</tr>
</tbody>
</table>
Native Graphs
Neo4j preserves relationships

• The relationships in your model exist as first-class entities in the database

• First-class entities in memory

• First-class entities on disk
Index-free adjacency

**Neo4j**
- Index → Node [O(\log N)]
- Node → Properties [O(1)]
- Node → Edges (by type) [O(1)]
  - this includes degree information
- Edges by type → Edges [O(1)]
- Edge → Properties [O(1)]
- Edge → source /target Node [O(1)]

**Relational Databases**
- Everything is an index
  - *(sometimes a table scan)*
- typically O(\log n)
  - *(sometimes hash: O(1))*
  - *Still indirection through index*
Graphs at scale
Queries scale with dataset size

- Index-free adjacency means query execution time never depends on total data size

- Query execution only depend on the number of edges actually related to the nodes actually found

- Expected execution time based on average degree
Graphs scale up easier than scaling out

• Neo4j trivially handles several billion nodes and edges in a single graph

• RAM size the main factor for performance

• Big RAM is not expensive
  • Terabyte RAM is readily available (even on AWS)

• RAM size growing faster than dataset sizes
Neo4j scales out queries

- Replicated cluster
- Fault tolerance
- Query processing capacity scale out
Neo4j scales in reality

• Real customers, such as Albumprinter have graphs in production that are larger than
  • 1 billion nodes
  • 2.5 billion edges
• Other customers have had >90% of the social graph of Facebook replicated in Neo4j (before Facebook restricted their data use policies)
Learn more about Neo4j
Getting started with Neo4j

1. Download Neo4j: [http://neo4j.com/download/](http://neo4j.com/download/)
2. Start the server.
3. It should be running on: [http://localhost:7474](http://localhost:7474)
4. Log-in with default credentials
   user: neo4j
   password: neo4j
5. Choose a new password

We’re good to go!
Tutorials for learning Neo4j

• :play intro - to learn the Neo4j query interface
• :play movies - simple examples on a simple dataset
• :play northwind-graph - the classic northwind example as a graph
• :play http://guides.neo4j.com/fundamentals/cypher_the_basics.html introduction to Cypher
• :play http://guides.neo4j.com/modeling_airports a tutorial on graph data modeling
• :play http://guides.neo4j.com/reco/file a tutorial on how to use Neo4j for recommendations
Developer Pages

neo4j.com/developer

- Intro to Graphs
- RDBMS to Graphs
- Data Import & Data Modeling
- Language Guides & Drivers
- Visualization
- Neo4j Ecosystem
Graph Academy

neo4j.com/graphacademy

- Online & Classroom Training
- University Program
- Certification
- Webinars
- Graph Days, Graph Talks
Become a Neo4j Certified Professional

neo4j.com/graphacademy/neo4j-certification/

1 hour certification exam covers:

- Cypher
- Data Modeling
- Clustering
- and more...
Cypher Reference Card

neo4j.com/docs/cypher-refcard/current/
Developer Documentation

neo4j.com/docs
neo4j.com/docs/developer-manual/current
neo4j.com/docs/operations-manual/current
Knowledge Base

neo4j.com/developer/kb

• Frequently Asked Questions and Answers
• Canonical Source
• Maintained by our Support and Field team
• Tagged
• Version specific
Example Applications

github.com/neo4j-examples
Meetup Groups

neo4j.meetup.com

Find out what's happening in Neo4j Meetup groups around the world and start meeting up with the ones near you.

Related topics: Graph Databases - NoSQL - Big Data - Data Visualization - Graph Theory - Database Development - Data Analytics - Database Professionals - Open Source - Neo4j-Social
Slack

neo4j.com/slack
GitHub

github.com/neo4j
github.com/neo4j-contrib
github.com/neo4j-examples
Stack Overflow

stackoverflow.com/questions/tagged/neo4j
Books

neo4j.com/books
Free, high value e-books
Third party publications
Talk to us

- Have an interesting project, use-case?
  - We can help you promote it with articles, talks, webinars
- Wrote a tool, library, connector?
  - We’d love to make it known in our community
- Have questions about licensing & pricing?
  - Reach out with your questions, we can help clarify
- Want to work on Neo4j?
  - We have many cool jobs, please apply
  - We also have open research topics

- Email me
  - tobias@neo4j.com
Research topics

that we are interested in collaborating on
Graph Indexes

- Topology-based indexes
- Path indexes
- Indexing of nodes based on related nodes

- How to ensure complex indexes are consistent when the data changes

- <your idea here>
  - anything that can be used to speed up queries
Graph Partitioning

• How to split a graph into multiple partitions in order to handle larger graphs

• Automatic partitioning

• Schema-guided partitioning

• <your idea here>
Distributed Query Execution

• Dividing the work of one query over multiple workers
  • Both in a partitioned environment
  • and in a replicated environment
Efficient Data Structures

• Better ways of representing graphs and indexes

• On persistent storage

• In memory / cache
Semantic/Intentional locking

• and/or other ways of ensuring consistent updates in a concurrent system

• Both when the data is partitioned,
• and when the data is replicated,
• as well as on a single machine with many threads
Graph Metrics and Statistics

- that we can compute and maintain up to date while the graph is being updated with low cost overhead
- that can either be used to directly answer user queries
- or to better optimise queries
Query optimisation

• more efficient ways to evaluate queries

• better algorithms for finding an “optimal” plan
Graph Query Languages

- Evolution of Cypher
- New ways of querying graphs
- Languages and models for expressing algorithms on graphs
- *<your idea here>*
  - anything that make people more successful and efficient in making sense of data
Fault Tolerance

• Resilient fault tolerant systems

• with as low overhead as possible
Thank you!
Questions?

Tobias . Lindaaker@neo4j.com
@thobe #neo4j