COMP5338 – Advanced Data Models

Week 8: Neo4j Internal and Data Modeling

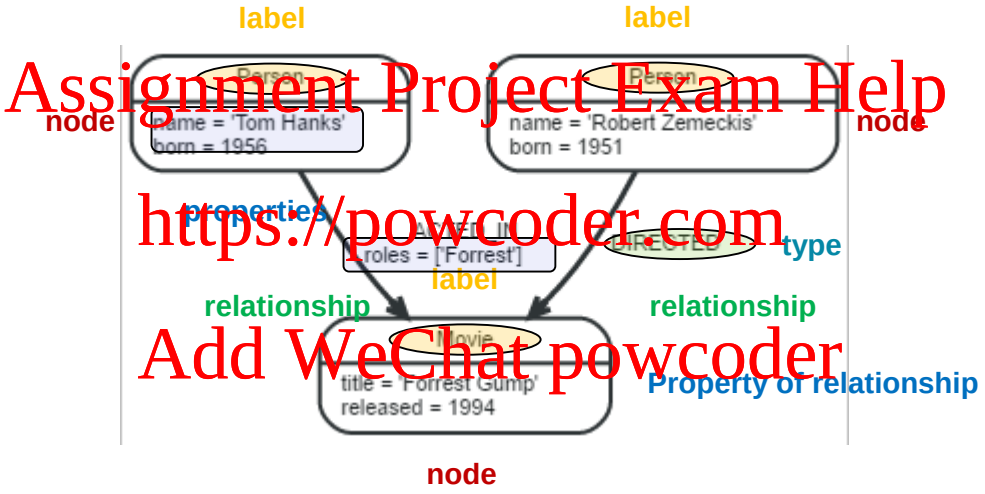
Dr. Ying Zhou
School of Information Technologies

Outline

- **Neo4j Storage**
- Neo4j Query Plan and Indexing
- Neo4j – Data Modeling
- Neo4j – Graph Algorithms

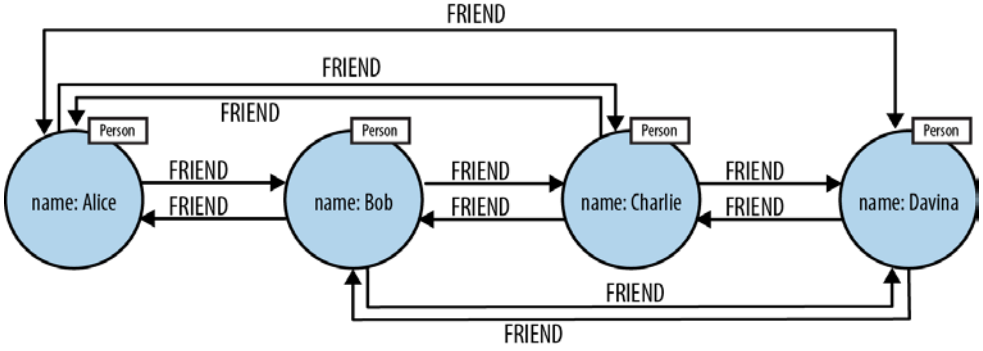
Materials adapted by permission from *Graph Databases (2nd Edition)* by Ian Robinson et al (O'Reilly Media Inc.). Copyright 2015 Neo Technology, Inc

Property Graph Model



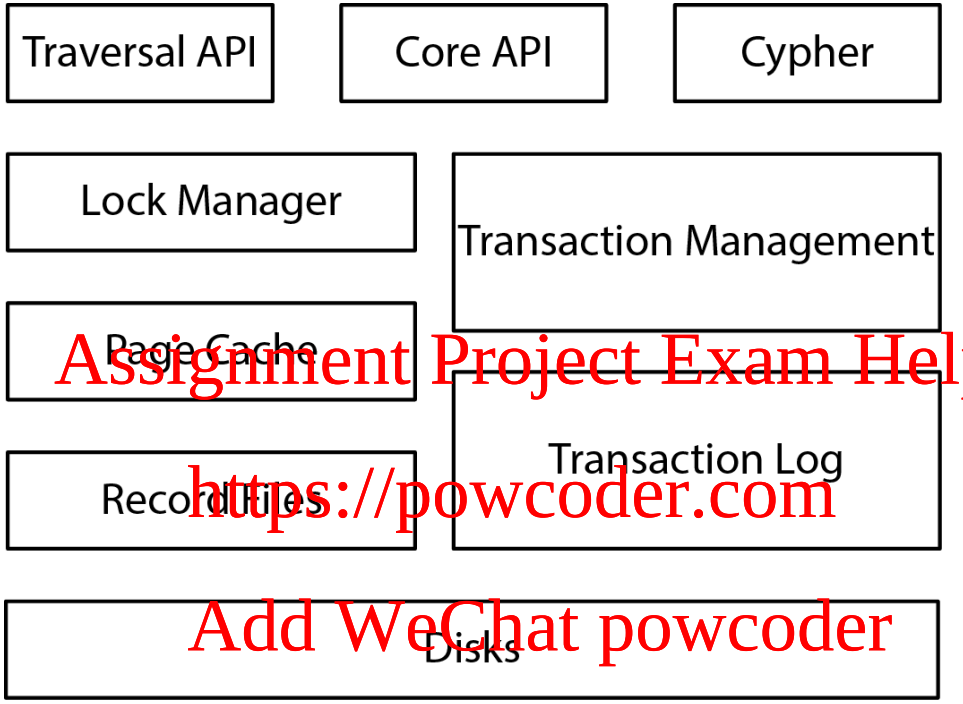
Index-free Adjacency

- Native storage of relationships between nodes
  - Effectively a pre-computed bidirectional join
- Traversal is like pointer dereferencing
  - Almost as fast as well
- Index-free Adjacency
  - Each node maintains direct link to its adjacent nodes
  - Each node is effectively a micro-index to the adjacent nodes
- Cheaper than global indexes
  - Query are faster, do not depends on the total size of the graph



Slides 3-10 are based on Graph Database chapter 6.1 and 6.2

Neo4j Architecture

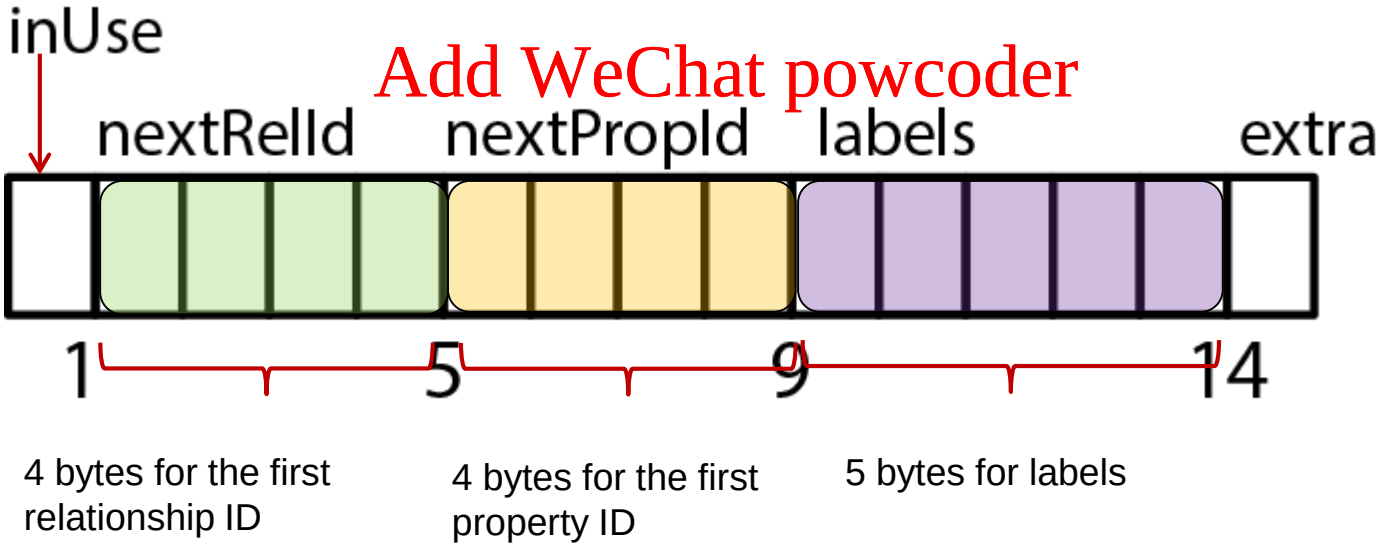


Page 163 of Graph Database

- Graph data is stored in *store files* on disk
  - Nodes, relationships, properties and labels all have their own store files.
  - Separating graph and property data promotes fast traversal
- user's view of their graph and the actual records on disk are structurally dissimilar

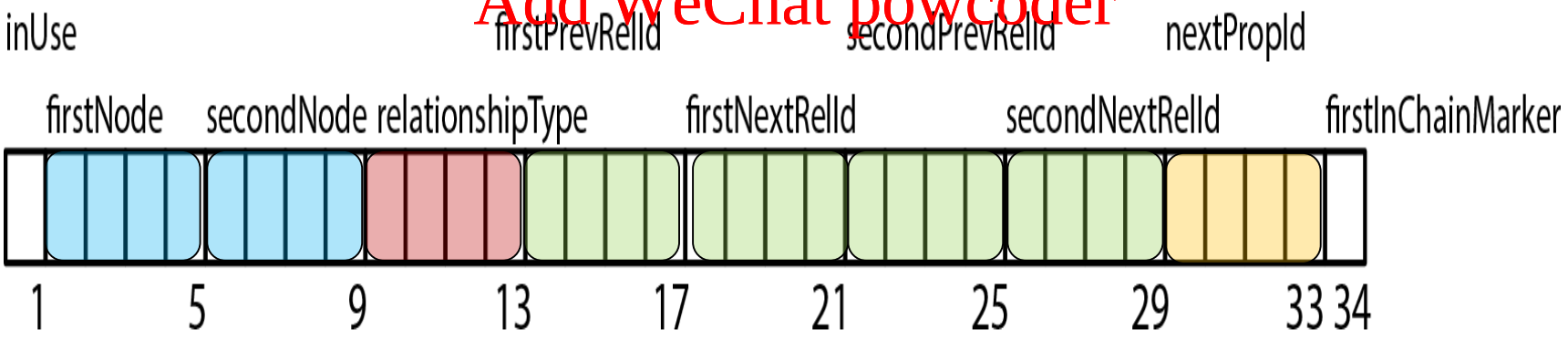
Node store file

- All node data is stored in one node store file
- Physically stored in file named *neostore.nodestore.db*
- Each record is of a **fixed size** – 15 bytes (*was 9 bytes in earlier version*)
- Offset of stored node = node id * 15 (node id = 100, offset = 1500)
- Deleted IDs in *.id* file and can be reused



Relationship store file

- All relationship data is stored in one relationship store file
- Physically stored in file named *neostore.relationshipstore.db*
- Each record is of a fixed size – 34 bytes
- Offset of stored relationship = relationship id * 34
  - So, relationship id = 10, offset = 340

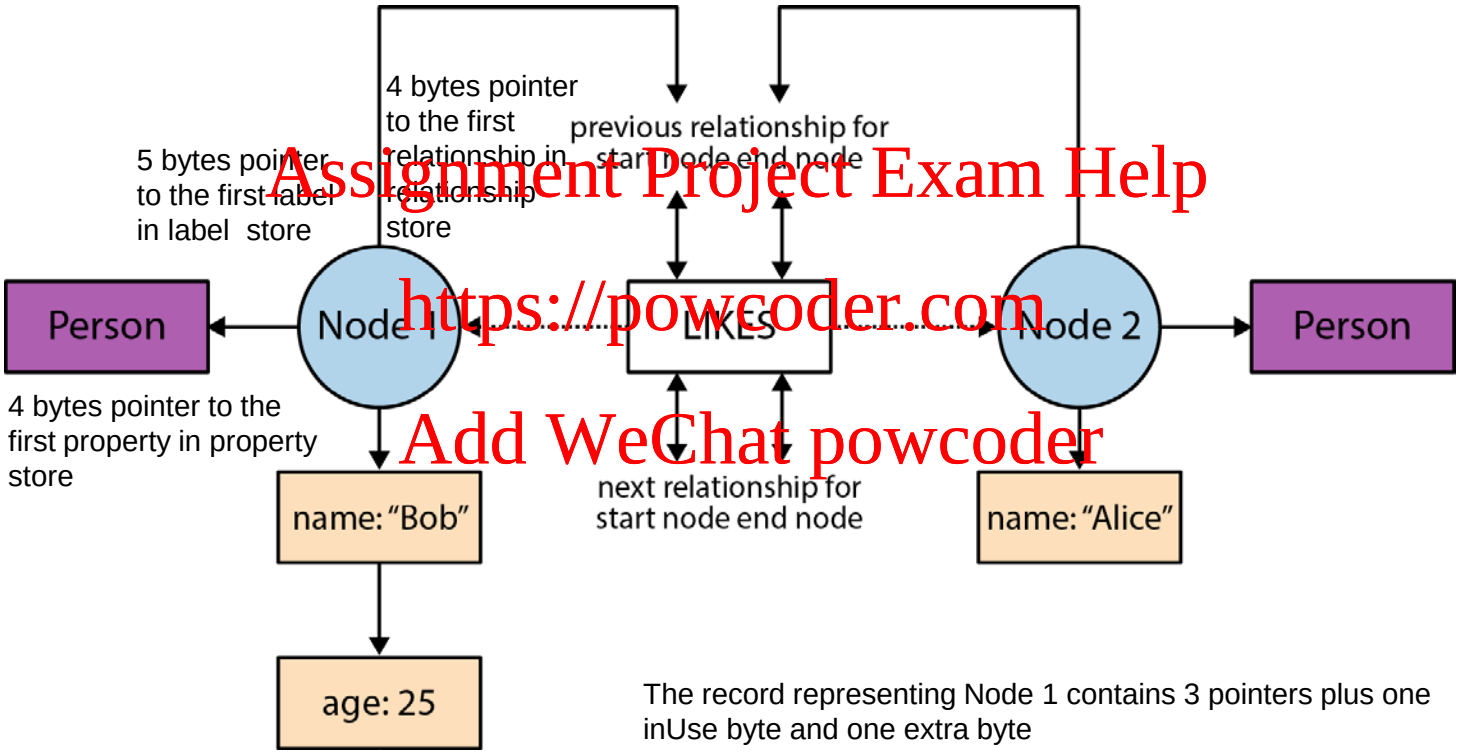


Other Files

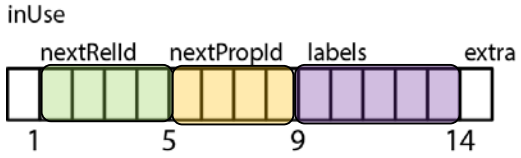
- **Property store** contains fixed size records to store properties for nodes and relationships
  - Simple properties are stored inline
  - Complex ones such as long string or array property are stored else where
- Node label in node records references data in **label store**
- Relationship type in relationship record references data in **relationship type store**
- Both Node ID and Property ID are of 4 bytes
  - The maximum ID value is $2^{32}-1$
  - ID is assigned and managed by the system
    - The corresponding record will be stored in the computed offset
  - The IDs of deleted nodes/relationships will be reused

Node structure

- Bob LIKES Alice

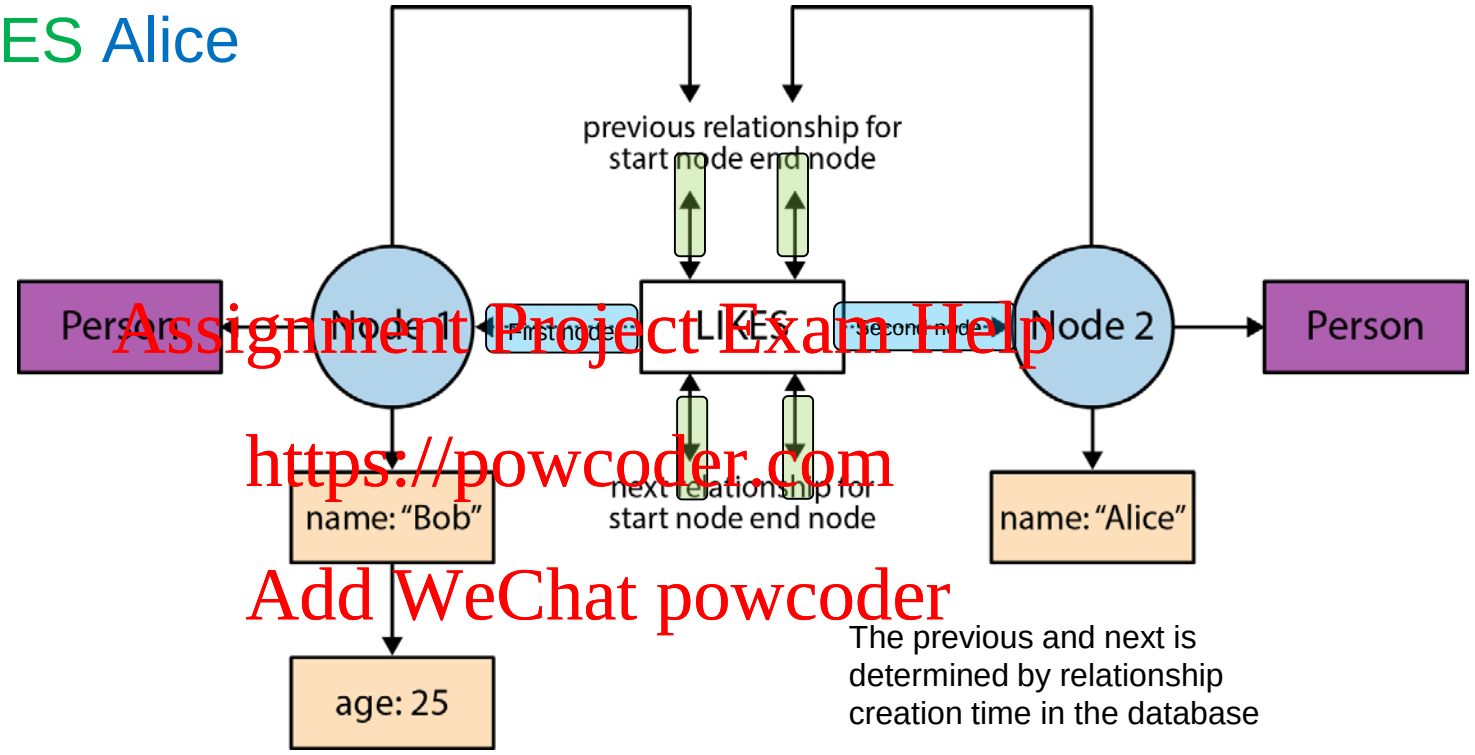


The record representing Node 1 contains 3 pointers plus one inUse byte and one extra byte

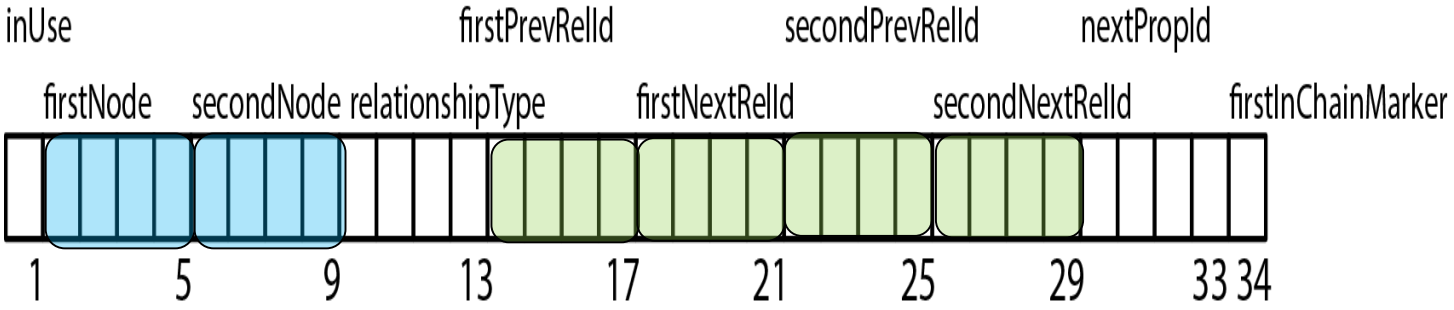


Relationship structure

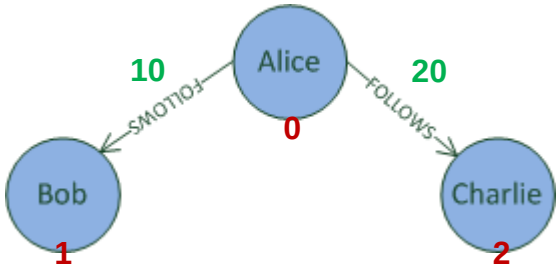
- Bob LIKES Alice



The previous and next is determined by relationship creation time in the database

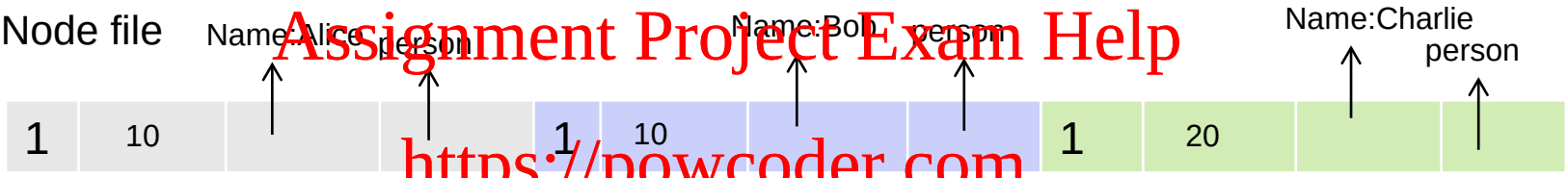


Doubly linked list



Creation order:
- node Alice
- node Bob
- node Charlie
- relationship Alice --> Bob
- relationship Alice --> Charlie

Node file

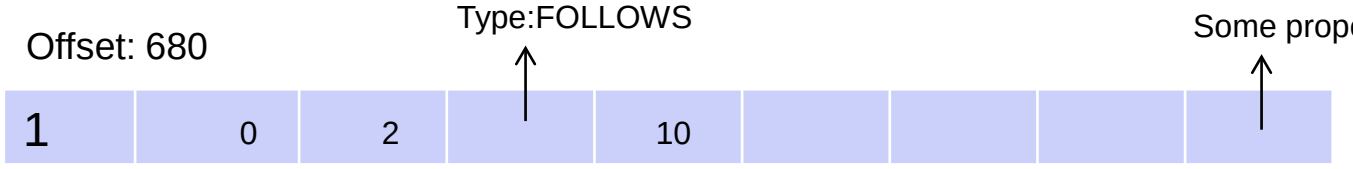


Relationship file

Offset: 340

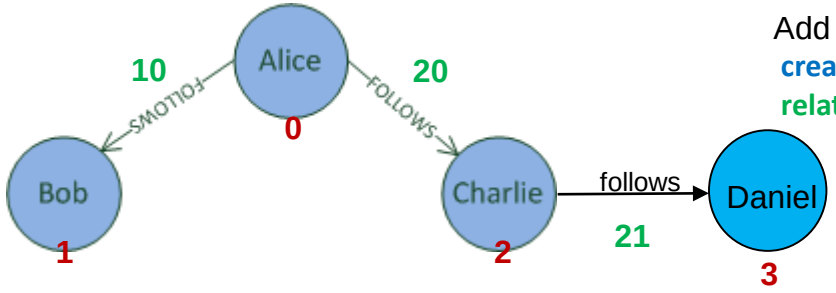
Alice's next relation

Offset: 680



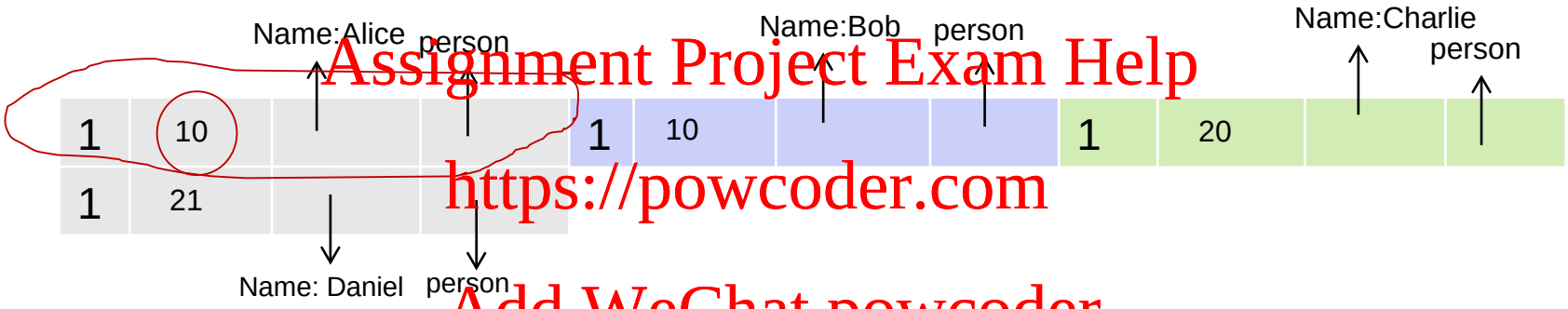
Alice's previous relation

Doubly linked list (cont'd)



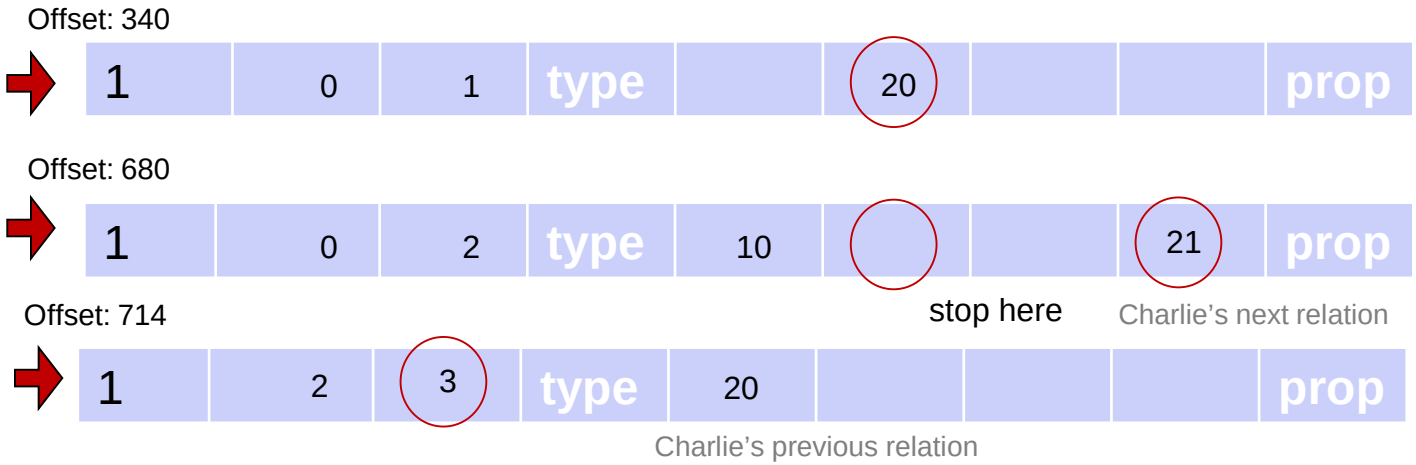
Add another node and relationship
create node Daniel
relationship Charlie --> Daniel

Node file



MATCH (Person {Name: "Alice}) –[*2]->(other) RETURN other

Relationship file



"The node and relationship stores are concerned **only** *with the* ***structure*** *of the graph, not its property data. Both stores use fixed-sized records so that any individual record's location within a store file can be rapidly computed given its ID. These are critical design decisions that underline Neo4j's commitment to high-performance traversals."*

-- Chapter 6, Graph Databases

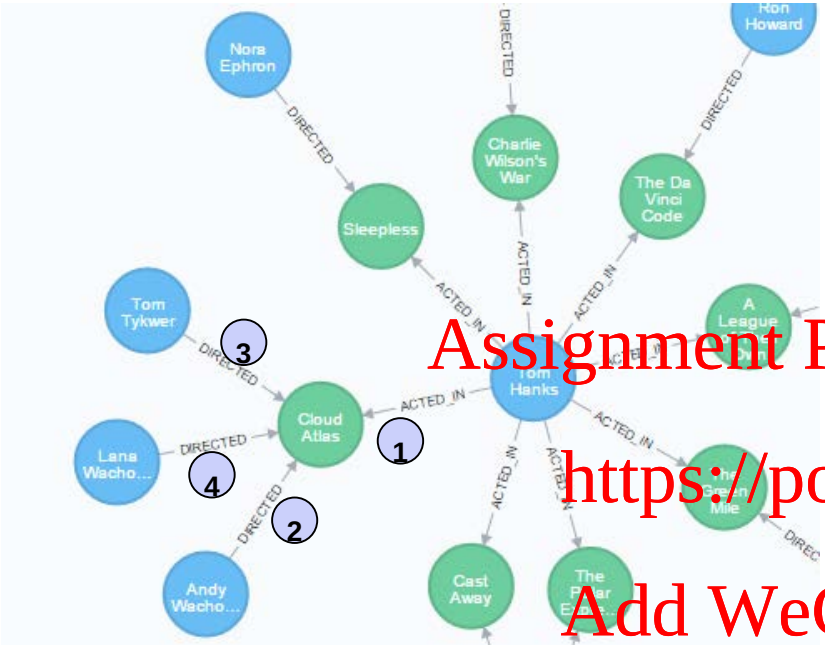
Outline

- Neo4j Storage
- **Neo4j Query Plan and Indexing**
- Neo4j – Data Modeling
- Neo4j – Graph Algorithms

Neo4j Query Execution

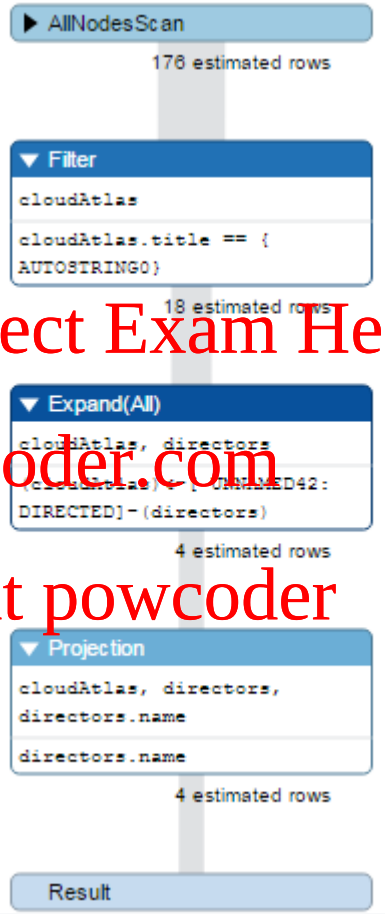
- Each Neo4j Query is turned into an execution plan by a **execution planner**
  - **Rule** Strategy Planner
    - Consider available indexes but does not use statistical information
  - **Cost** Strategy Planner (default and in development)
    - Use statistic information to evaluate a few alternative plans
    - E.g. If there are less Movie nodes than People nodes, a query involving both may get better performance if starting from a collection of Movie nodes
      - See example in lab
- Query plan stages
  - Starting point
  - Expansion by matching given path in the query statement
  - Row filtering, skipping, sorting, projection, etc…
  - Combining operations
  - Updating

Query Plan: an example

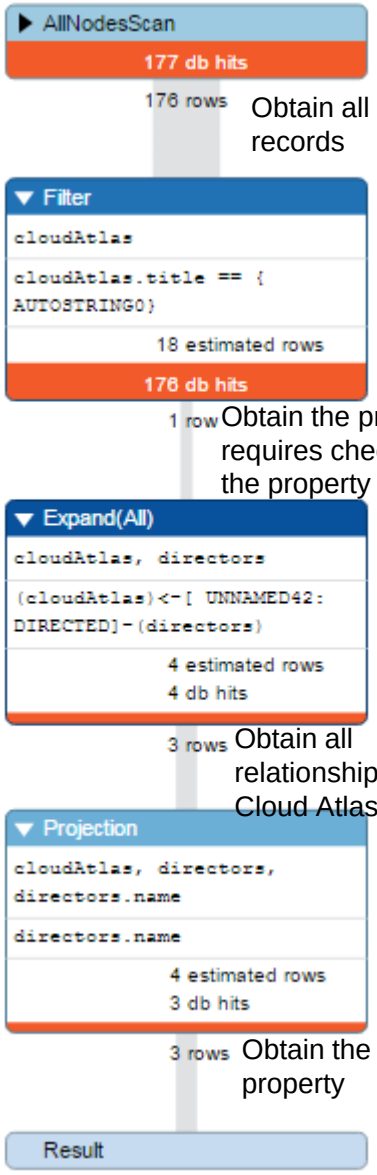


Query:

MATCH
(cloudAtlas {title: "Cloud Atlas"})<-[:DIRECTED]-(directors)
RETURN directors.name



explain



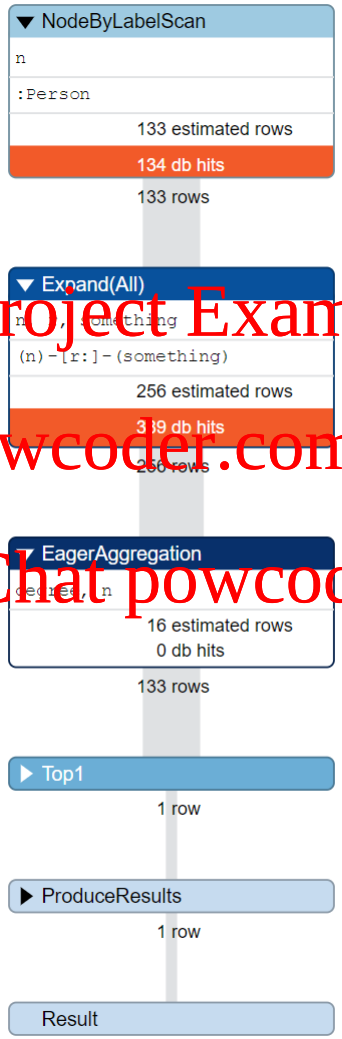
profile

Query Starting Points

- Most queries start with one or a set of **nodes** except if a relationship ID is specified
  - **MATCH (n1)-[r]->() WHERE id(r)= 0 RETURN r, n1**
  - This query will start from locating the first record in the relationship file
- Query may start by scanning all nodes
  - **MATCH(n) RETURN (n)**
  - **MATCH (cloudAtlas {title: "Cloud Atlas"})<-[:DIRECTED]-(directors) RETURN directors.name**
- Query may start by scanning all nodes belonging to a given label
  - **MATCH (p:Person{name:"Tom Hanks"}) return p**
  - Labels are implicitly indexed
- Query may start by using index

Query starting from labelled node

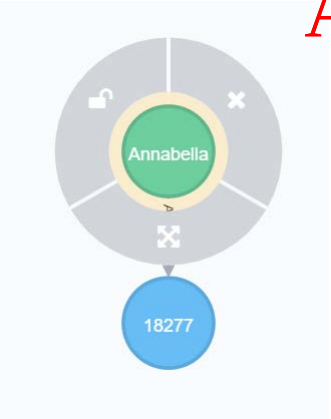
```
MATCH (n:Person) -[r]-(something)
with n, count(something) as degree
order by degree
limit 1
return n
```



Obtain all 133 Person nodes records

Obtain133 nodes + 256 relationships

Memory processing



Query Plan With Index

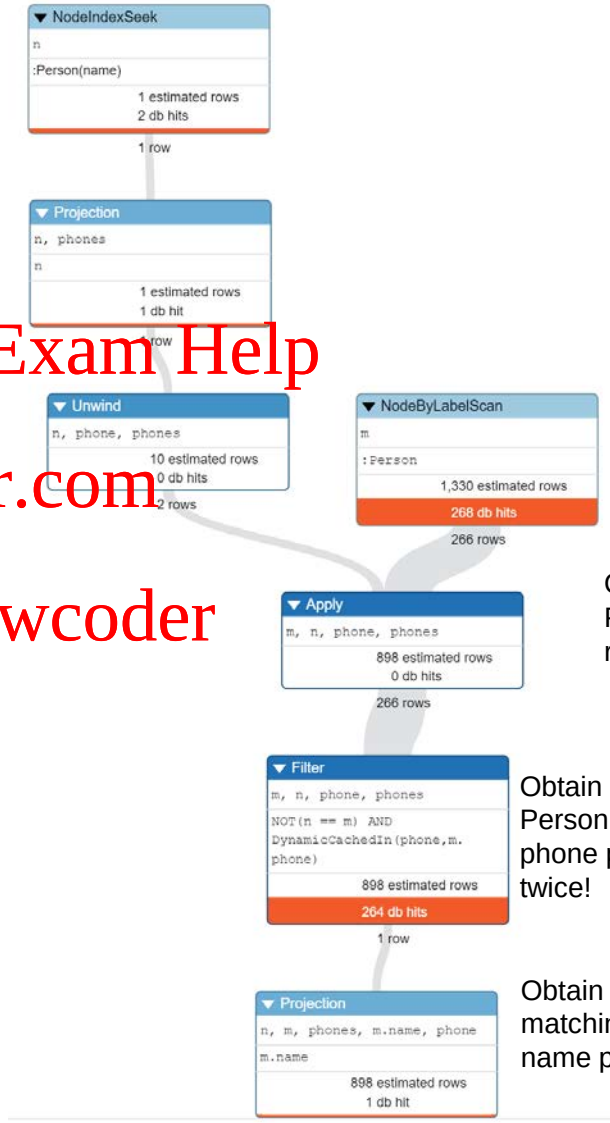
- Neo4j supports index on properties of labelled node
- Index has similar behaviour as those in relational systems
- It can be built on single or composite properties
- Create Index
  - **CREATE INDEX ON :Person(name)**
- Drop Index
  - **DROP INDEX ON :Person(name)**

Query:

MATCH **(bacon:Person {name:"Kevin Bacon"})-[*1..4]-(hollywood)**
RETURN DISTINCT hollywood

A relatively complex query and plan

```
MATCH  (n:Person{name:  "Tom  Hanks"})
WITH n.phone as phones, n
UNWIND phones as phone
MATCH (m:Person)
WHERE phone in m.phone and n<>m
RETURN m.name
```

Obtain all 133 Person nodes records twice!

Obtain 132 Person nodes's phone property twice!

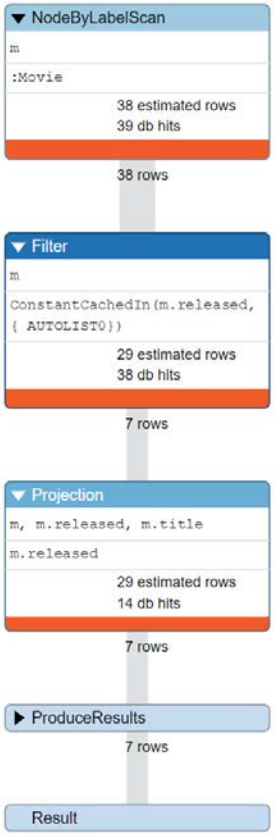
Obtain the matching node's name property

Apply works by performing a nested loop. Every row being produced on the left-hand side of the Apply operator will be fed to the leaf operator on the right-hand side, and then Apply will yield the combined results

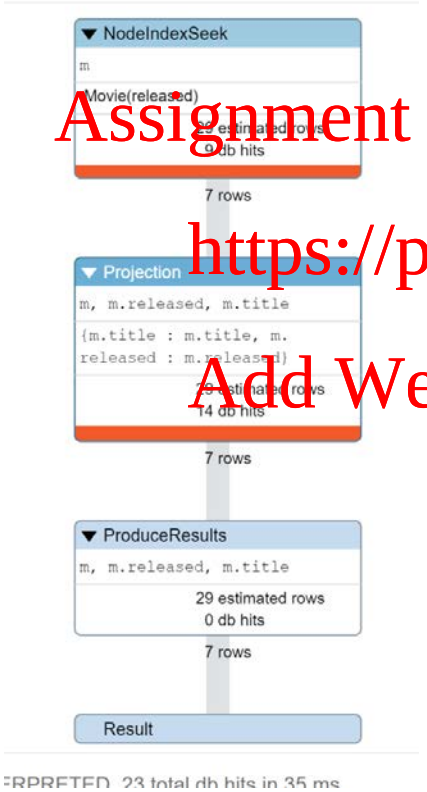
Comparing Execution Plans

```
MATCH (m: Movie)
WHERE m.released IN [1999,2003]
RETURN m.title, m.released
```

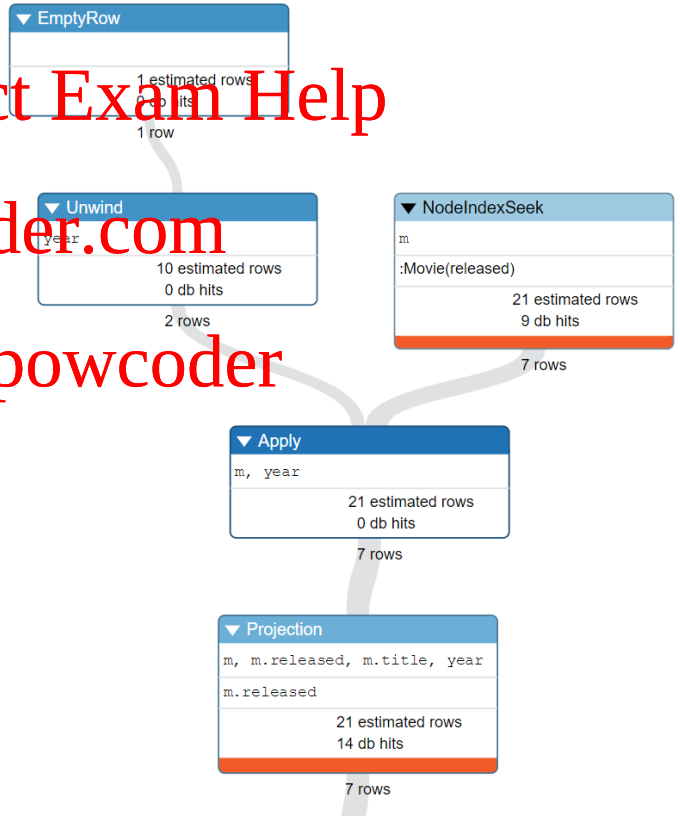
```
UNWIND [1999,2003] as year
MATCH (m: Movie)
WHERE m.released = year
RETURN m.title, m.released
```



Earlier version



version 3.2

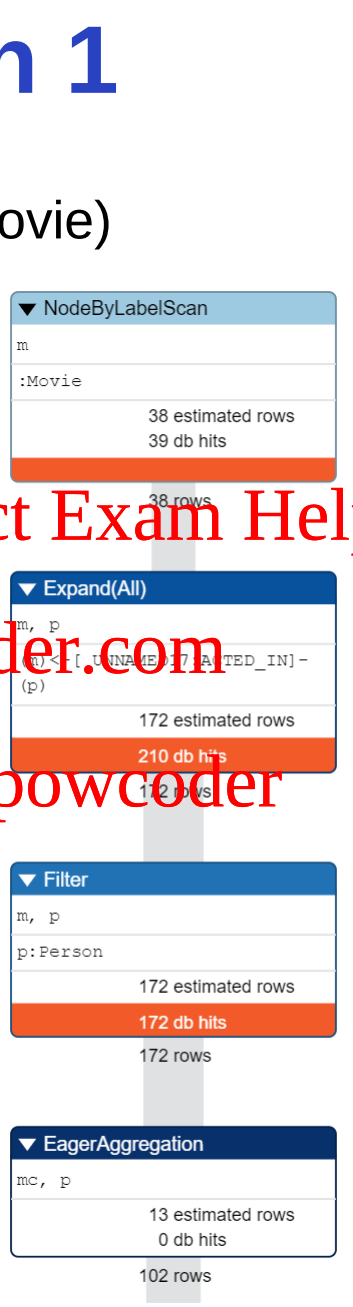


Another example of comparison

- Question: Find out a list of person who has acted in at least three movies and also directed at least one movie
- Cypher is powerful and flexible
  - It is possible to write very different queries that produce the same results
  - The performance could have big difference
  - The DB engine does not have much knowledge to rewrite the queries as those in SQL
    - Not based on relational algebra

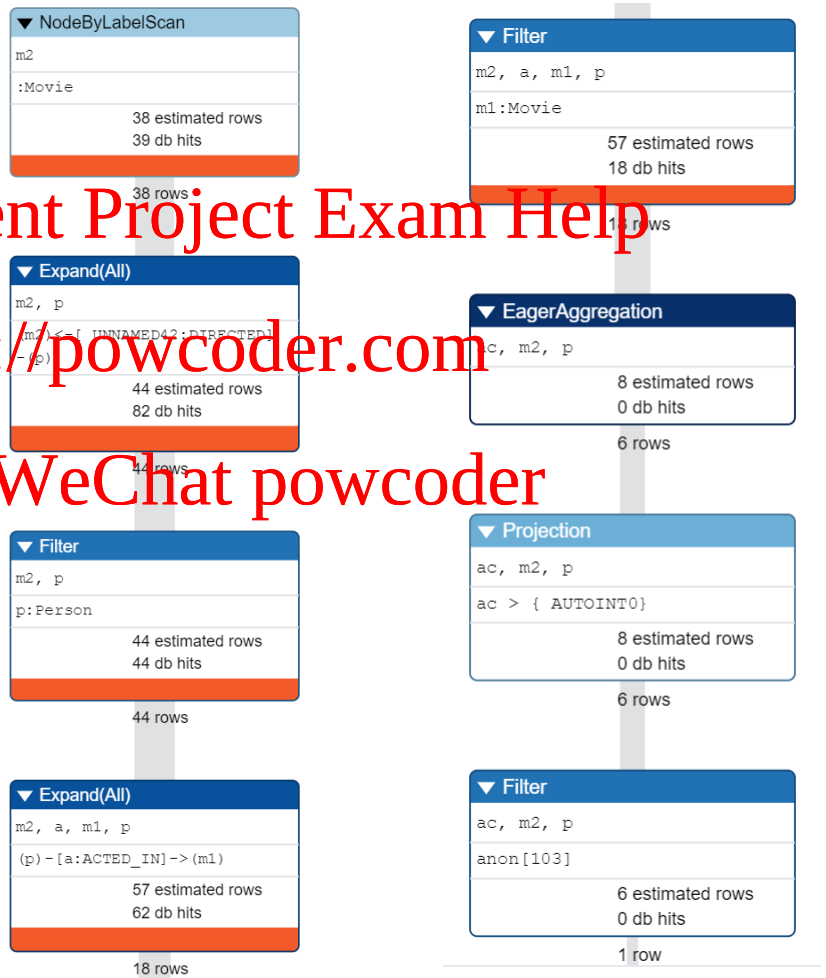
Option 1

```
MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
WITH p, count(m) AS mc
WHERE mc > 2
MATCH (p)-[:DIRECTED]->(m2:Movie)
RETURN p.name, m2.title
```



Option 2

```
MATCH (m1:Movie)<-[a:ACTED_IN]-(p:Person)-[:DIRECTED]->(m2:Movie)
WITH p, count(distinct m1) as ac  m2
WHERE ac > 2
RETURN p.name, m2.title
```



Transactions

- Neo4j supports full ACID transactions
  - Similar to those in RDBMS
- Uses locking to ensure consistency
  - Lock Manager manages locks held by a transaction
- Logging
  - Write Ahead Logging (WAL)
- Transaction Commit Protocol
  - Acquire locks (Atomicity, Consistency, Isolation)
  - Write Undo and Redo records to the WAL
    - for each node, relationship, property changed is written to the log
  - Write commit record to the log and flush to disk (Durability)
  - Release locks
- Recovery – if the database server/machine crashes
  - Apply log records to replay changes made by the transactions

Outline

- Neo4j Storage
- Neo4j Query Plan and Indexing
- **Neo4j – Data Modeling**
- Neo4j – Graph Algorithms

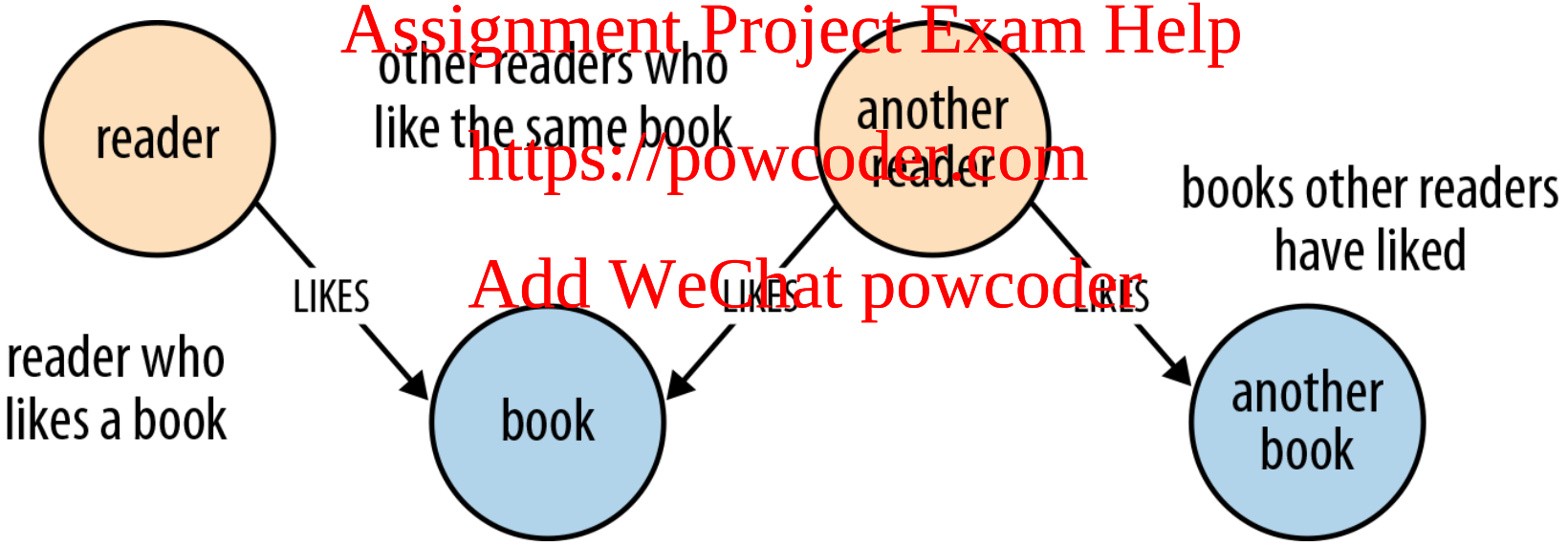
Graph Data Modelling

- Graph data modelling is very closely related with domain modelling
- You need to decide
  - Node or Relationship
  - Node or Property
  - Label/Type or Property
- Decisions are based on
  - Features of entities in application domain
  - Your typical queries
  - Features and constraints of the underlying storage system

Slides 35-39 are based on Chapter 4 of Graph Databases book

Node vs. Relationship

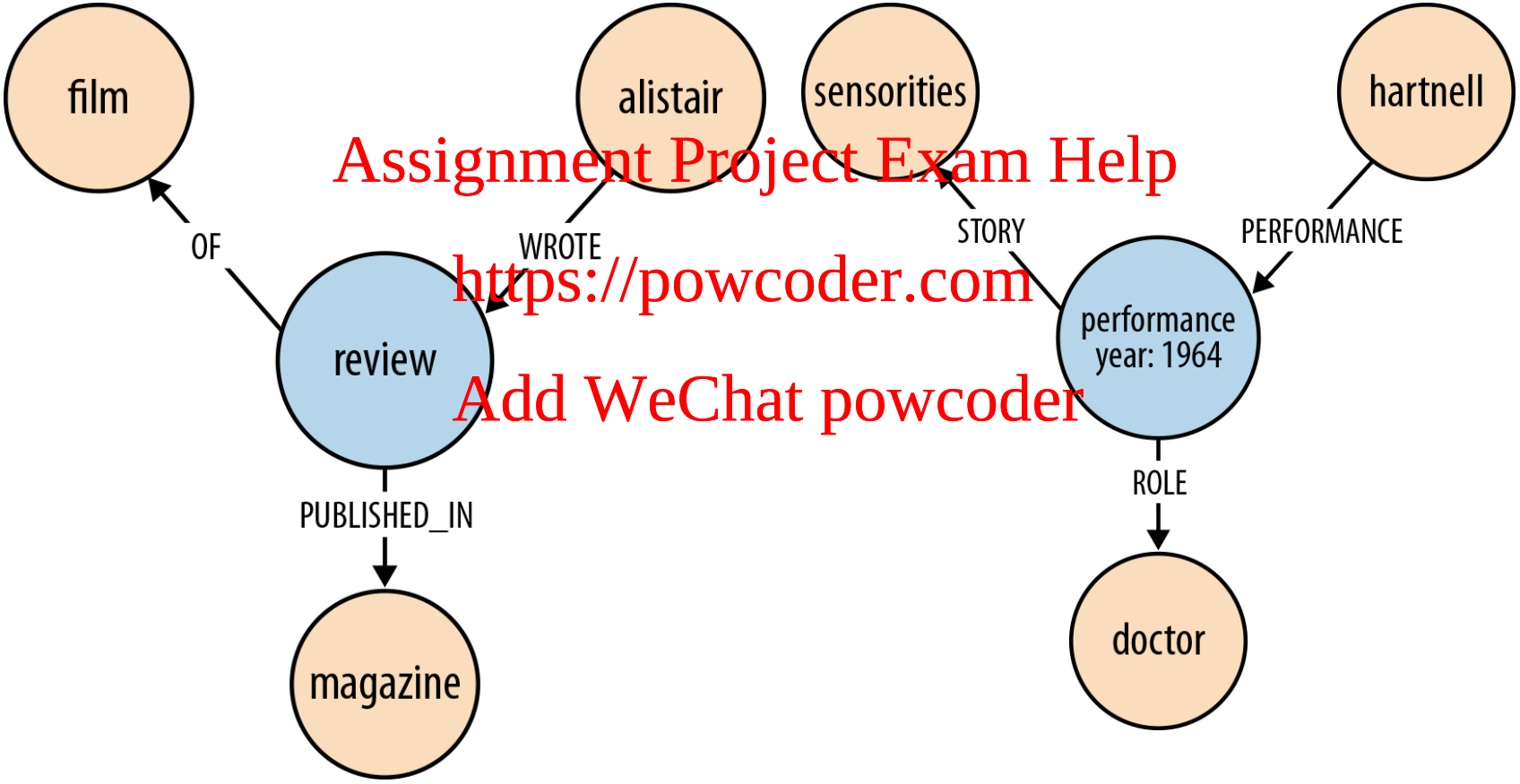
- Nodes for Things, Relationship for Structures
  - **AS A** reader who likes a book, **I WANT** to know which books other readers who like the same book have liked, **SO THAT** I can find other books to read.




```
MATCH (:Reader {name:'Alice'})-[:LIKES]->(:Book {title:'Dune'})
<-[:LIKES]-(:Reader)-[:LIKES]->(books:Book)
RETURN books.title
```

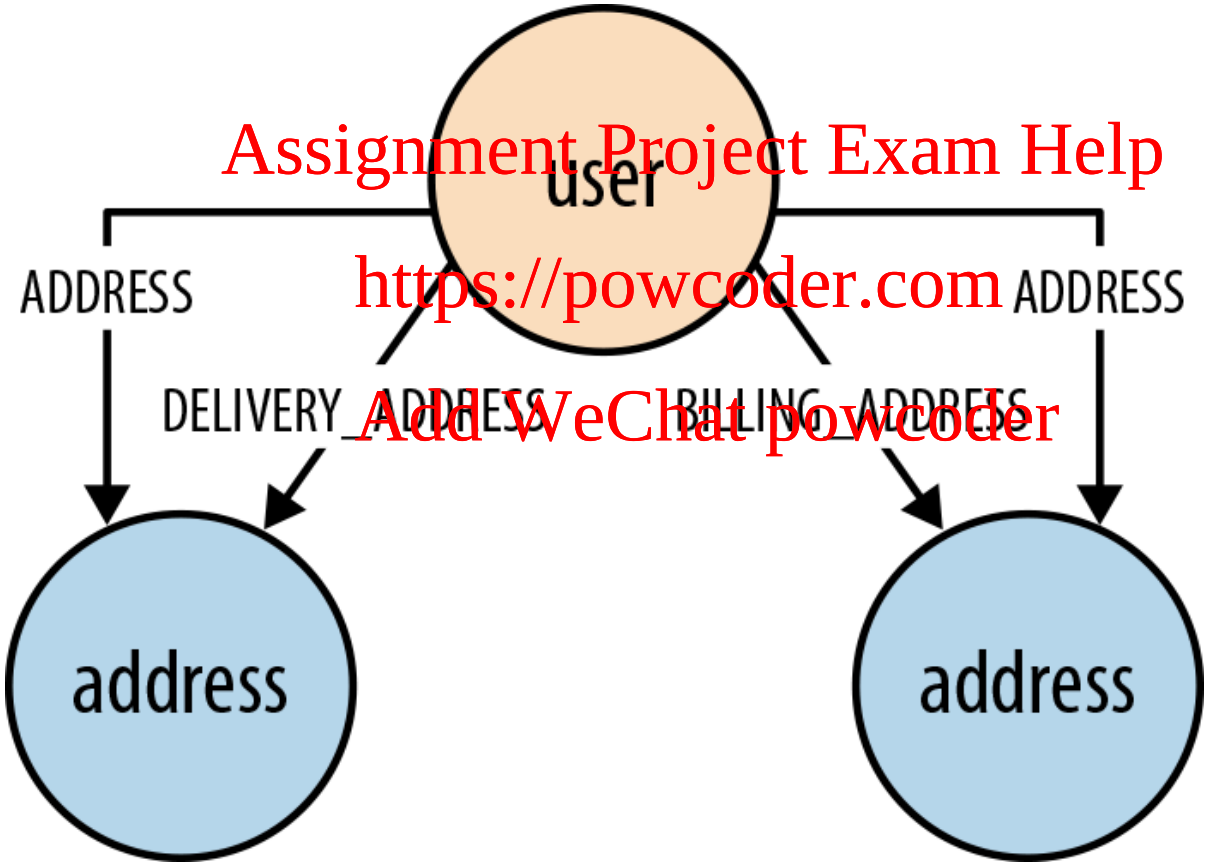
Node vs. Relationship

- Model Facts as Nodes



Node or Property

- Represent Complex Value Types as Nodes



Relationship Property or Relationship Type

- E.g. The relationship between user node and address node can be:
  - typed as **HOME_ADDRESS**, **BILLING_ADDRESS** or
  - typed as generic **ADDRESS** and differentiated using a type property {type:'home'}, {type:'billing'}
- We use fine-grained relationships whenever we have a closed set of relationship types.
  - Eg. there are only a finite set of address types
  - If traversal would like to follow generic type ADDRESS, we may have to use redundant relationships
    - MATCH (user)-[:HOME_ADDRESS|WORK_ADDRESS|
      DELIVERY_ADDRESS]->(address)
    - MATCH (user)-[:ADDRESS]->(address)
    - MATCH (user:User)-[r]->(address:Address)

Outline

- Neo4j Storage
- Neo4j Query Plan and Indexing
- Neo4j – Data Modeling
- **Neo4j –** Graph Algorithms

Graph Algorithm

- In addition to graph query and traversal, a rich set of graph algorithms are provided by Neo4j
  - Used to be part of the Neo4j server
  - It is now moved out as a separate project
- The graph algorithms are implemented as Cypher **procedures** and need to be installed separately
- Procedure is a mechanism to extend Neo4j
  - Take arguments, perform database operation and return result
  - Written in Java and compiled to jar files
  - Once installed, it can be called directly from Cypher

https://neo4j.com/docs/graph-algorithms/3.4/

https://neo4j.com/docs/developer-manual/current/extending-neo4j/procedures/

References

- Ian Robinson, Jim Webber and Emil Eifrem, *Graph Databases*, Second Edition, O'Reilly Media Inc.,
  - You can download this book from the Neo4j site, https://neo4j.com/graph-databases-book/?ref=home
    - Chapter 4, Chapter 6
- Neo4j – Reference Manual
  - https://neo4j.com/docs/developer-manual/current/