COMP5338 – Advanced Data Models

Week 7: Neo4j Internal and Data Modelling

Dr. Ying Zhou
School of Computer Science

Outline

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

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

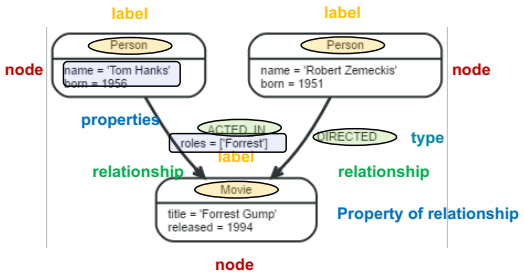


Table concept is not part of the data model

The database is a large graph, could contain independent subgraphs

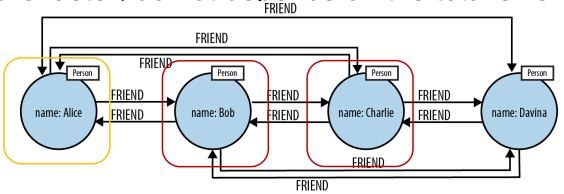
Community edition only supports one user database; Enterprise edition supports multiple user databases

Logic data model and physical storage model could be totally different

It is theoretically possible to construct a property graph model with any storage backend

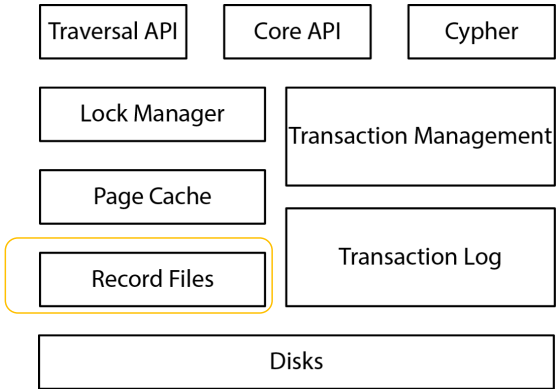
Index-free Adjacency

- 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 a 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 4-11 are based on Graph Database chapter 6.1 and 6.2

Neo4j Architecture



Page 163 of Graph Database

Property Graph and Store files

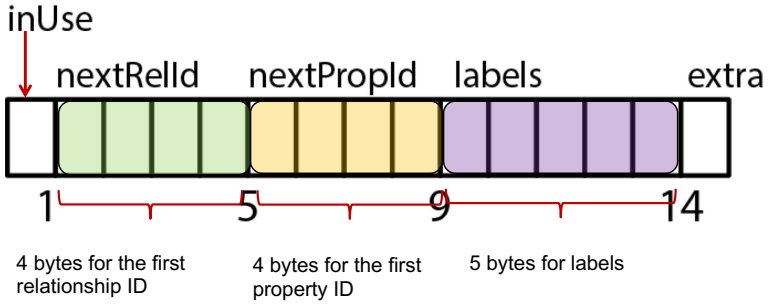
- Graph data is stored in *store files* on disk
  - Nodes, relationships, properties, labels and types all have their own store files.
    - Check under **<neo4j-home>/data/databases/neo4j**
  - Separating graph structure and property data promotes fast traversal
- Node, relationship, property, label and type all have *system assigned* IDs
- They are stored as fixed length record in respective stores
- user's view of their graph and the actual records on disk are structurally dissimilar

```
1.1M  1 Oct 22:56 neostore.relationshipstore.db
 96K  1 Oct 22:56 neostore.relationshipstore.db.id
8.0K  1 Oct 22:17 neostore.relationshiptypestore.db
 40K  1 Oct 22:56 neostore.relationshiptypestore.db.id
8.0K  1 Oct 22:17 neostore.relationshiptypestore.db.names
 40K  1 Oct 22:56 neostore.relationshiptypestore.db.names
```

```
568K  1 Oct 22:56 neostore.nodestore.db
 88K  1 Oct 22:56 neostore.nodestore.db.id
8.0K  1 Oct 21:09 neostore.nodestore.db.labels
 40K  1 Oct 22:56 neostore.nodestore.db.labels.id
1.3M  1 Oct 22:56 neostore.propertystore.db
1.2M  1 Oct 22:56 neostore.propertystore.db.arrays
 48K  1 Oct 22:56 neostore.propertystore.db.arrays.id
112K  1 Oct 22:56 neostore.propertystore.db.id
8.0K  1 Oct 22:17 neostore.propertystore.db.index
 40K  1 Oct 22:56 neostore.propertystore.db.index.id
8.0K  1 Oct 22:17 neostore.propertystore.db.index.keys
 40K  1 Oct 22:56 neostore.propertystore.db.index.keys.id
8.0K  1 Oct 21:34 neostore.propertystore.db.strings
 48K  1 Oct 22:56 neostore.propertystore.db.strings.id
```

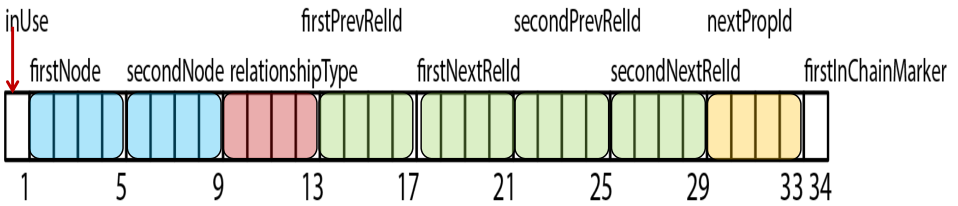

Node store file

- 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



Implications

- Both Node ID and Property ID are of 4 bytes
  - The maximum ID value is $2^{32}-1$
    - There is a maximum number of nodes/relationships in a database
  - 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

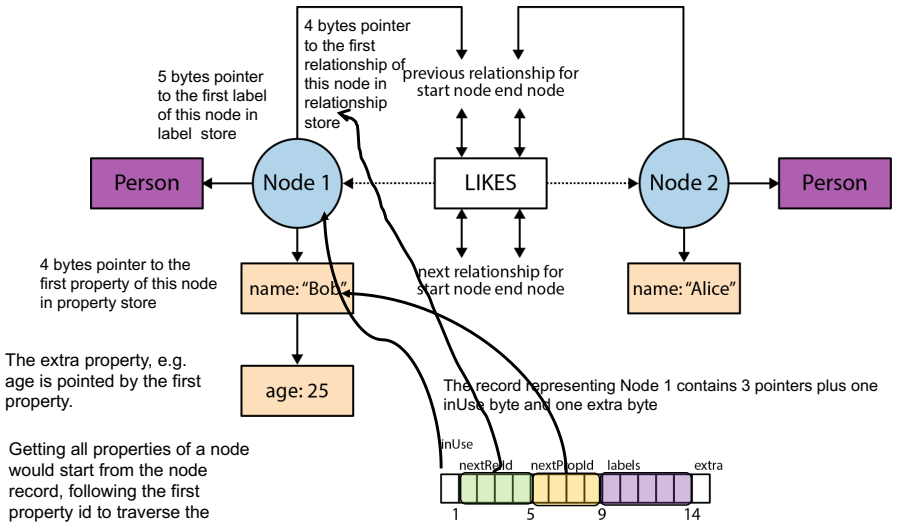
Other Files

- **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**

```
568K  1 Oct 22:56 neostore.nodestore.db
 88K  1 Oct 22:56 neostore.nodestore.db.id
8.0K  1 Oct 21:09 neostore.nodestore.db.labels
 40K  1 Oct 22:56 neostore.nodestore.db.labels.id
 48K  1 Oct 22:56 neostore.propertystore.db
1.1M  1 Oct 22:56 neostore.relationshipstore.db
 96K  1 Oct 22:56 neostore.relationshipstore.db.id
8.0K  1 Oct 22:17 neostore.relationshiptypestore.db
 40K  1 Oct 22:56 neostore.relationshiptypestore.db.id
8.0K  1 Oct 22:17 neostore.relationshiptypestore.db.names
 40K  1 Oct 22:56 neostore.relationshiptypestore.db.names
 48K  1 Oct 22:56 neostore.propertystore.db.strings
```

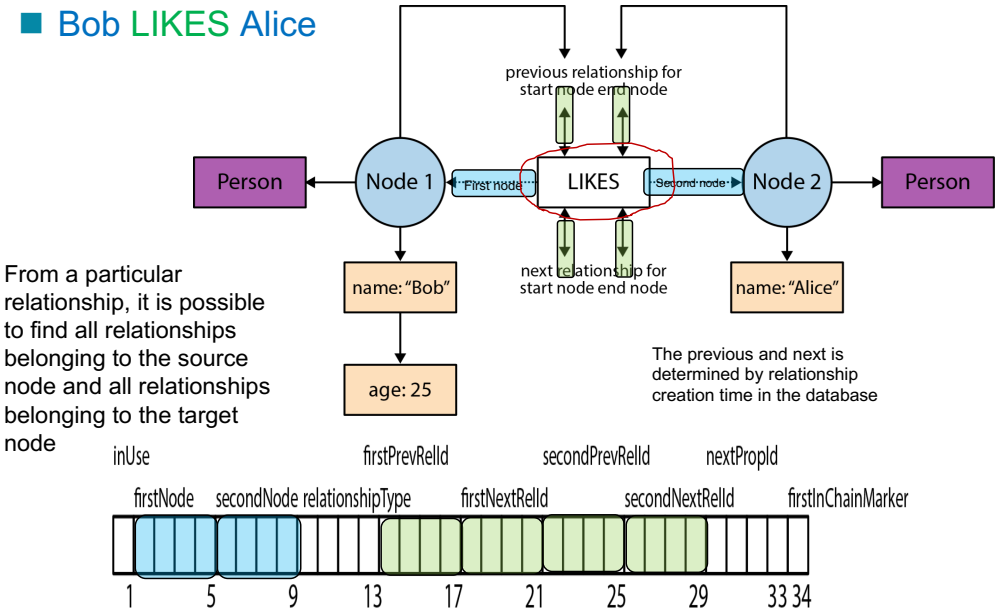

Node structure

- Bob LIKES Alice

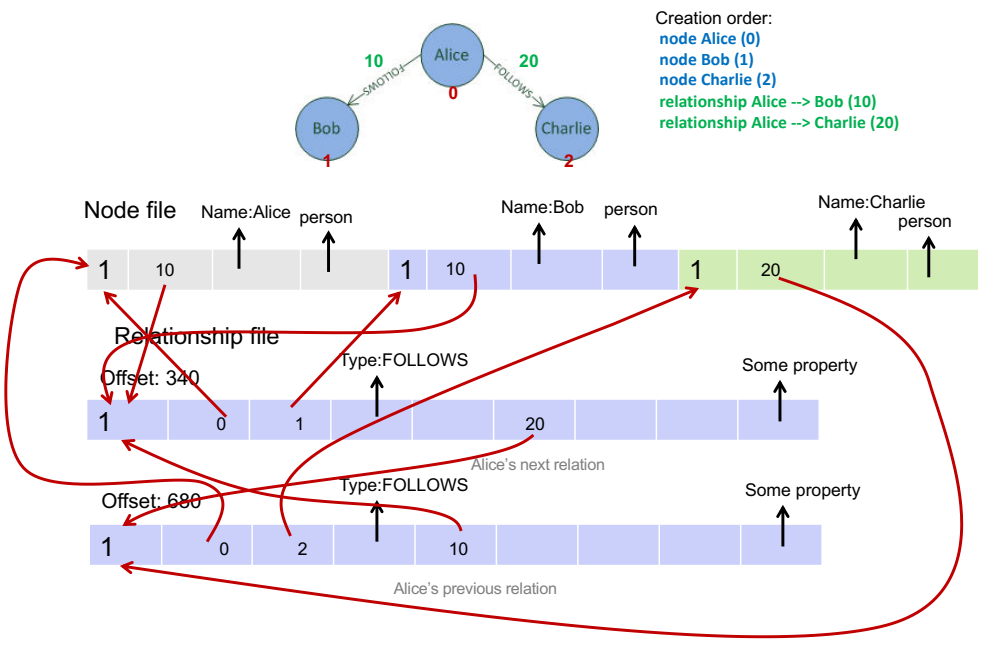


Relationship structure

- Bob LIKES Alice

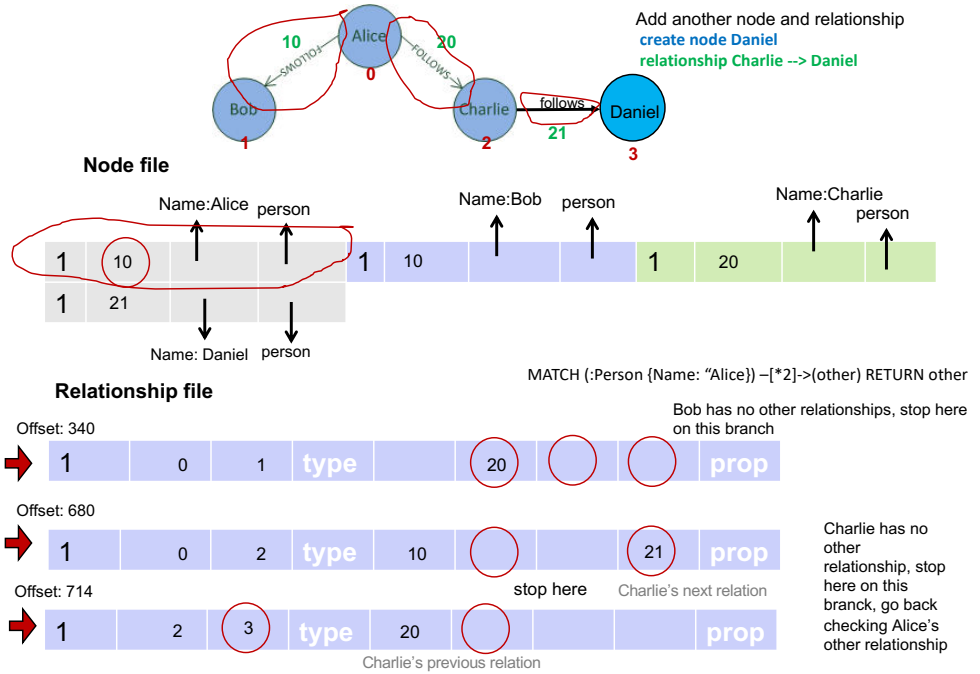


Doubly linked list



Create a relationship may need to update multiple records

Doubly linked list (cont'd)



"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

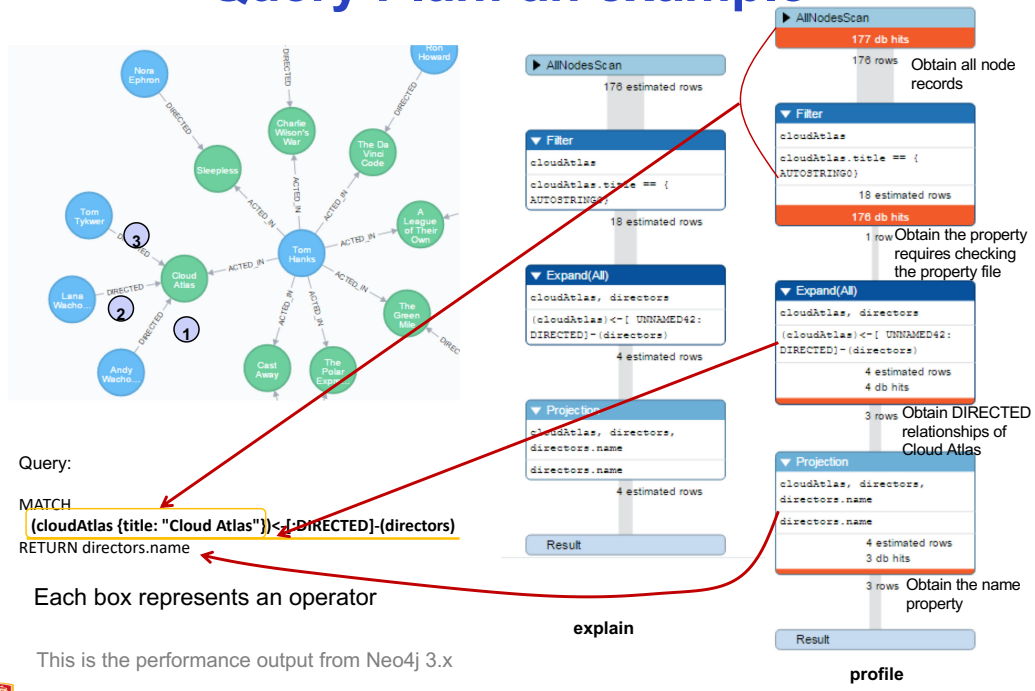
Outline

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

Neo4j Query Execution

- Each Neo4j Query is turned into an execution plan by an **execution planner**
- The **execution plan** is a tree-like structure consists of various **operators**, each implements a specific piece of work
- Query plan stages
  - Starting point (leaf node)
    - Obtaining data from storage engine
  - Expansion by matching given pattern in the query statement
  - Row filtering, skipping, sorting, projection, etc…
  - Combining operations
  - Updating
- Execution plans are evaluated based on statistics maintained by database
  - The number of nodes having a certain label.
  - The number of relationships by type.
  - Selectivity per index.
  - The number of relationships by type, ending with or starting from a node with a specific label.

Query Plan: an example



Evaluation Statistics

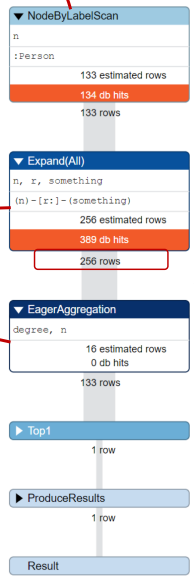
- Each operator is annotated with some statistics
- ***Rows*:** The number of rows that the operator produced. This is only available if the query was *profiled*.
- ***EstimatedRows*:** This is the estimated number of rows that is expected to be produced by the operator.
- ***DbHits:*** Some operator needs to retrieve data from or update data in the storage. A *database hit* is an abstract unit of this storage engine work
  - **Creating** a node, a relationship, a label, a type
  - **Deleting** a node, a relationship,
  - **Getting** a node, a property of a node, the label,…
  - **Getting** a relationship, a property of a label, the type,
  - Updating…

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 DESC
LIMIT 1
RETURN n, degree
```



Obtain all 133 Person nodes records

Obtain133 nodes + 256 relationships = 389 db hits

Memory processing

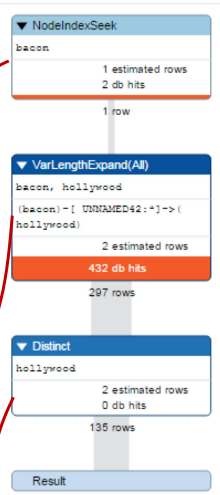
The **ProduceResults** operator prepares the result so that it is consumable by the user, such as transforming internal values to user values. It is present in every single query that returns data to the user, and has little bearing on performance optimisation.

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:

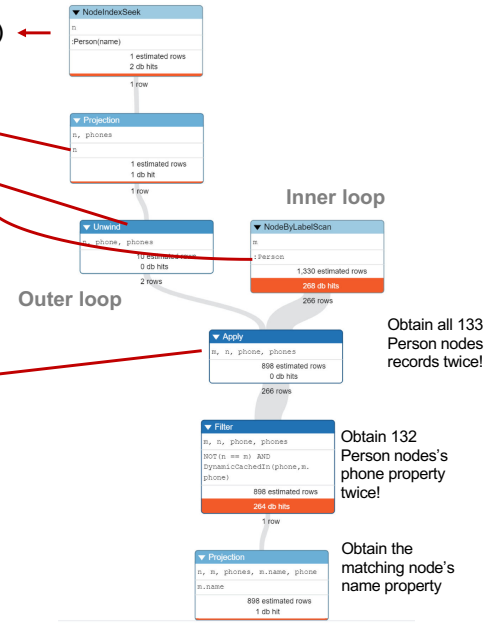
```
MATCH (bacon:Person {name:"Kevin Bacon"})-[*1..4]-(hollywood)
RETURN DISTINCT hollywood
```



Each row represents a path that may include more than one relationship, so the db hits number is larger than row number

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!

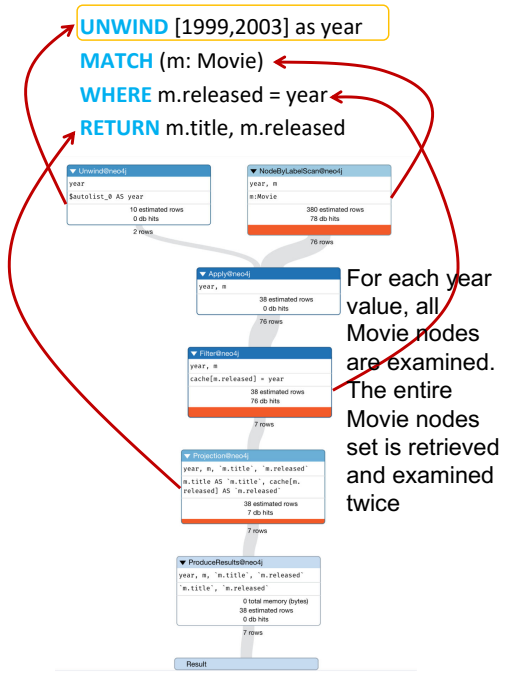
Obtain the matching node's name property

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
```

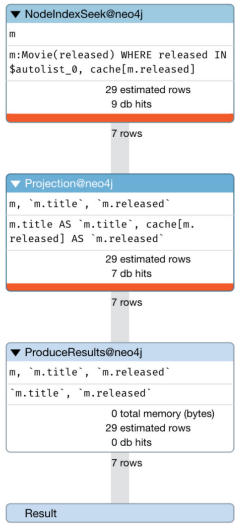

```
UNWIND [1999,2003] as year
MATCH (m: Movie)
WHERE m.released = year
RETURN m.title, m.released
```



For each year value, all Movie nodes are examined. The entire Movie nodes set is retrieved and examined twice

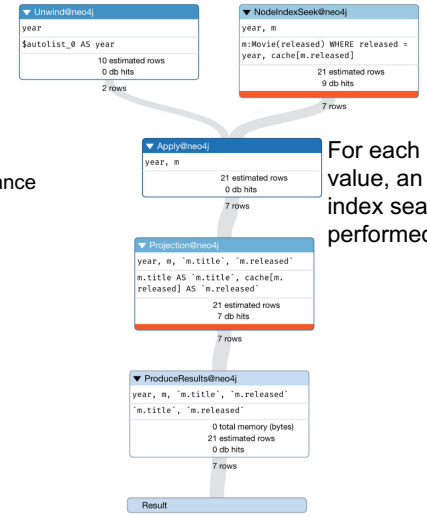
Comparing Execution Plans (with Index)

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



Similar performance

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



For each year value, an index search is performed

CREATE INDEX ON :Movie(released)

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
```

Start with **Movie** label because there are less **Movie** nodes than **Person** nodes

15 person have acted in at least 3 movies

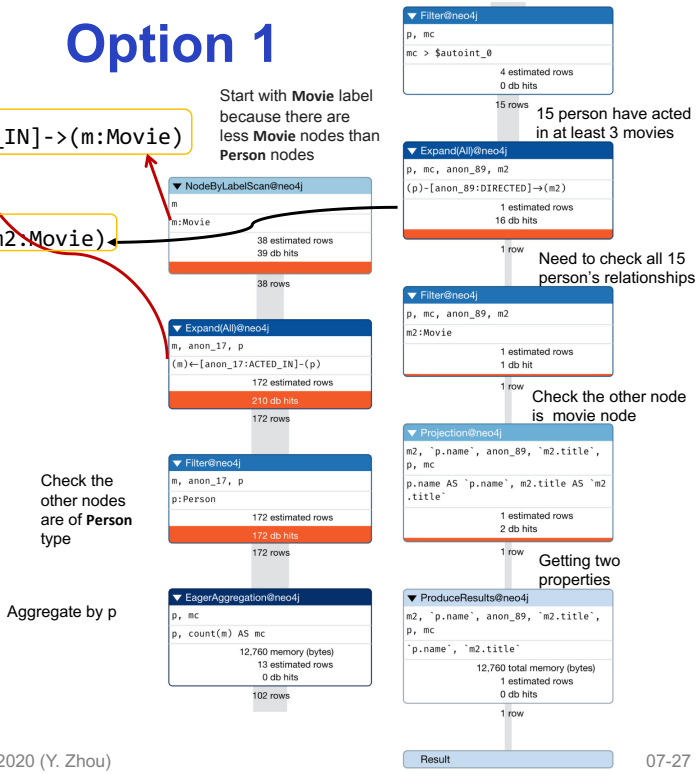
Need to check all 15 person's relationships

Check the other node is movie node

Getting two properties

Check the other nodes are of **Person** type

Aggregate by p



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
```

38 Movie nodes + 44 relationships

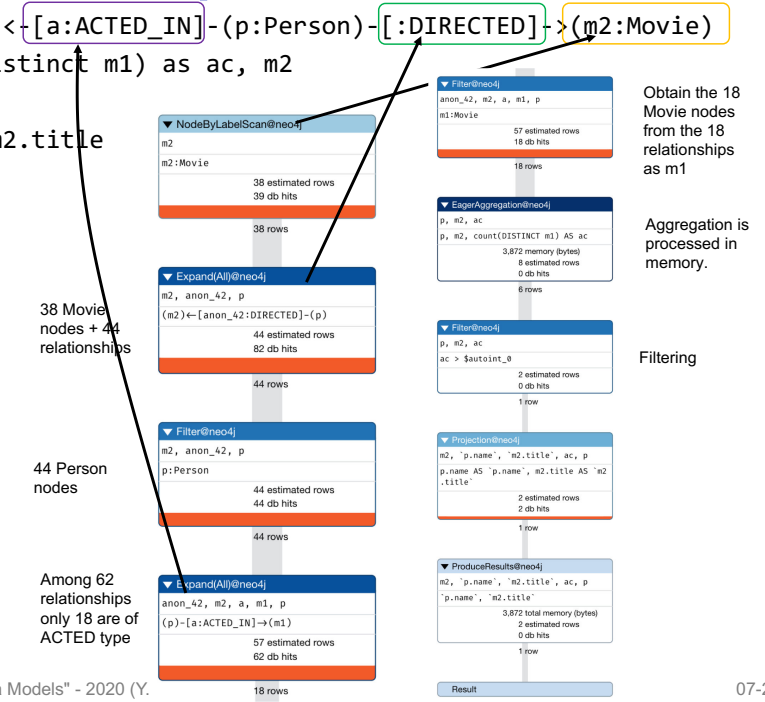
44 Person nodes

Among 62 relationships only 18 are of ACTED type

Obtain the 18 Movie nodes from the 18 relationships as m1

Aggregation is processed in memory.

Filtering

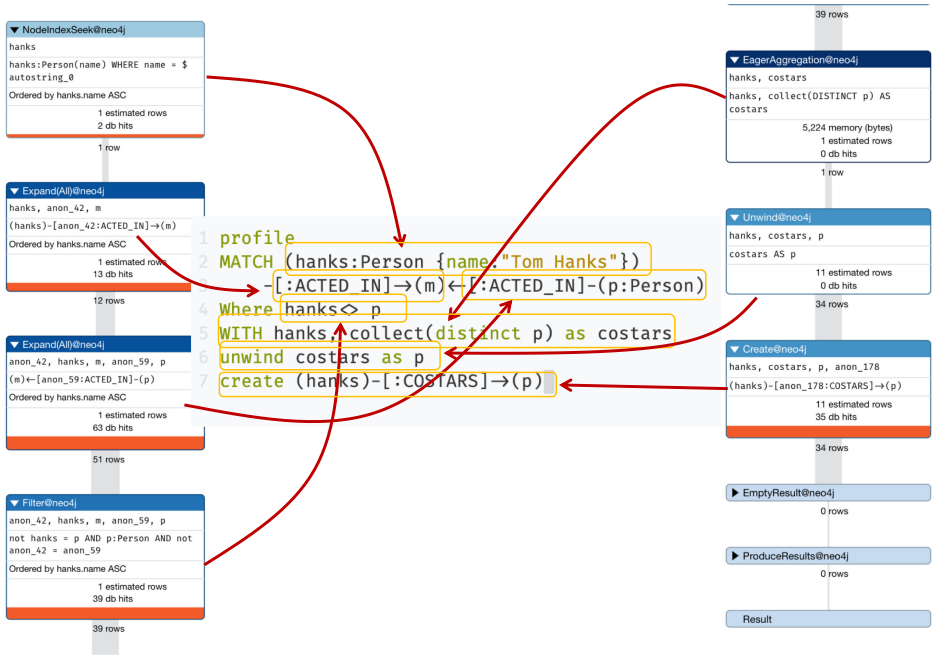


Performance of Creation Operations

- Example: add COSTARTS relationships for Tom Hanks
- Option 1: using CREATE

```
profile
MATCH (hanks:Person {name:"Tom Hanks"})
    -[:ACTED_IN]→(m)←[:ACTED_IN]-(p:Person)
Where hanks<> p
WITH hanks, collect(distinct p) as costars
unwind costars as p
create (hanks)-[:COSTARS]→(p)
```


Profile Result



Cypher version: CYPHER 4.1, planner: COST, runtime: INTERPRETED. 152 total db hits in 272 ms.

Option 2: Using MERGE

```
profile
MATCH (hanks:Person {name:"Tom Hanks"})
    -[:ACTED_IN]→(m)←[:ACTED_IN]-(p:Person)
Where hanks<> p
WITH hanks, p
merge (hanks)-[:COSTARS]→(p)
```

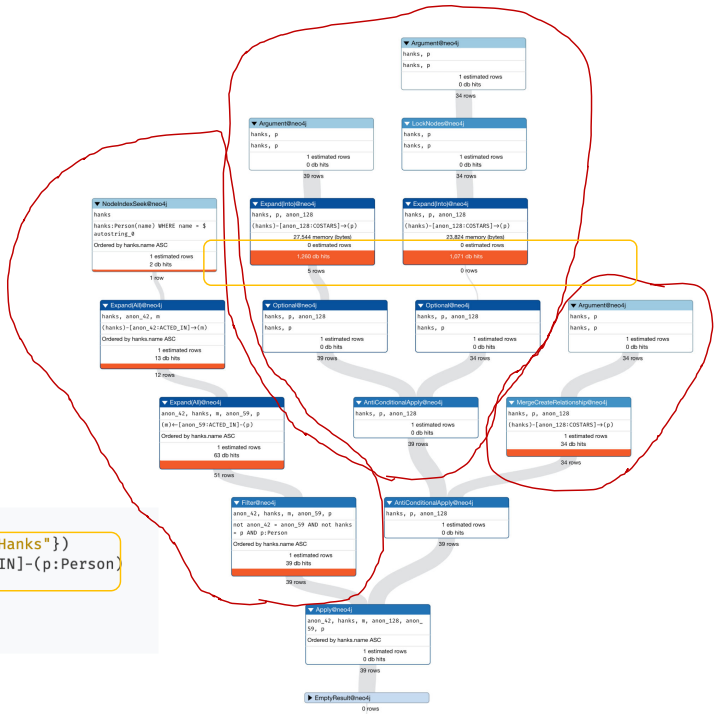
Cypher version: CYPHER 4.1, planner: COST, runtime: INTERPRETED. 2482 total db hits in 1498 ms.

Profile Result

For each p, check if there is a COSTARS relationship between node hanks and node p

Check relationship existence needs to traverse the entire relationship link.

Create the relationship when it does not exists. There are 34 create relationship actions.



```
profile
MATCH (hanks:Person {name:"Tom Hanks"})
    -[:ACTED_IN]→(m)←[:ACTED_IN]-(p:Person)
Where hanks<> p
WITH hanks, p
merge (hanks)-[:COSTARS]→(p)
```

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**

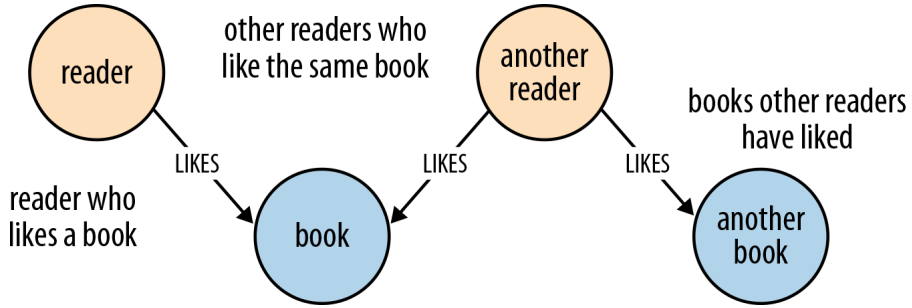
Graph Data Modelling

- 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

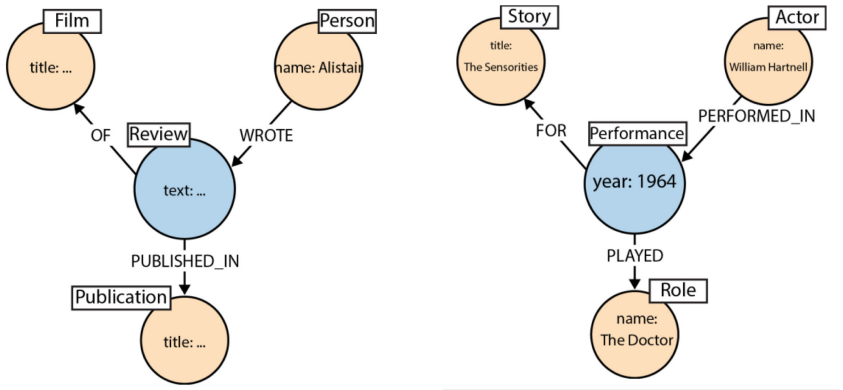
- Nodes for Things, Relationship for Structures
  - **AS A** reader who likes a book, **I WANT** to know which books other readers who like the same book have liked, **SO THAT** I can find other books to read.




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

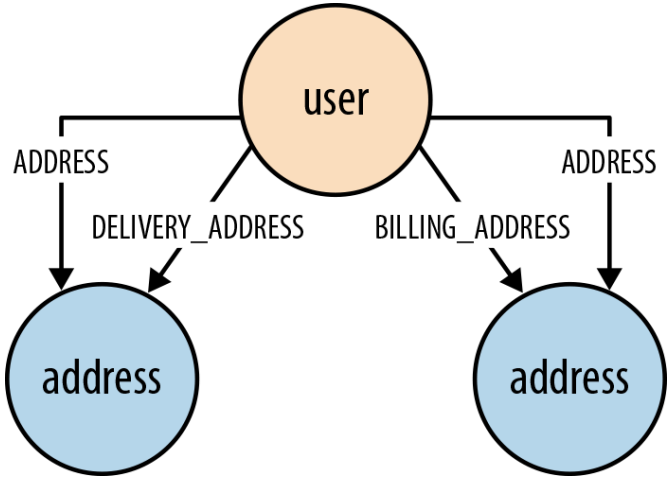
Node vs. Relationship

- 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)

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/
- Neo4j reference manual: Execution plan operators in detail
  - https://neo4j.com/docs/cypher-manual/current/execution-plans/operators/