Cypher Query Tuning

My query is too slow!
I can't change the model
I need to tune!
But how?

What are we going to learn?

- Stepwise query tuning
- How the Cypher query planner works
- Query Profiling
- Operation Costs, Cardinalities and Rows
- Batching Transaction
- Index Lookup performance

Steps for Query Tuning

Steps for Query Tuning

1. **Test:** Run slow query with `PROFILE` on representative dataset
2. **Measure:** Examine Query Plan & note hotspots
   a. intermediate rows
   b. db-hits
   c. total runtime
3. **Change: the model/adapt question** or
4. **Tune: the query**
5. **Step by step:** Focus on hotspots, build query up, apply tuning techniques

Let's look at some of the query tuning tools

Query Planning determines how a Query is executed

The Cypher Query Planner

Cypher Query Planner

Historically RULE planner based on heuristics

Since Neo4j 2.2 (greedy) COST planner based on database statistics

Since Neo4j 2.3 (IDP) COST planner based on database statistics

Since Neo4j 3.0 COST planner for write statements

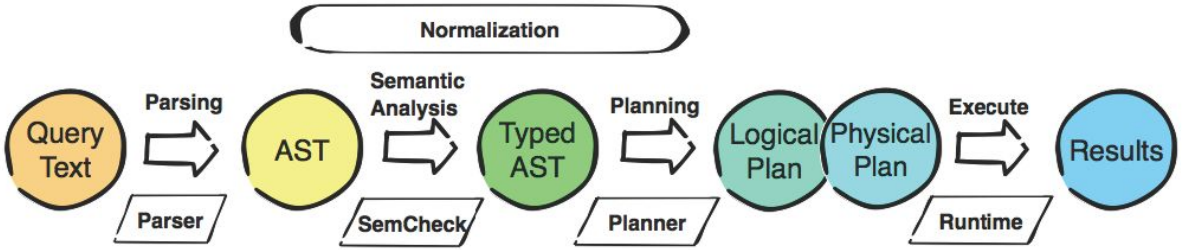
Continuous improvement

- rewriting of the query
- operation improvements
- new operations

Plans are compiled and **cached**

Cypher Query Processing

1. Parse into Abstract Syntax Tree (AST)
2. Semantic Checking
3. Rewrite
4. Logical Planning using transactional DB-Statistics
5. Physical Execution Plan (cached)
6. Execute with interpreted or compiled runtime



neo4j.com/blog/introducing-new-cypher-query-optimizer/

Query Plan Cache

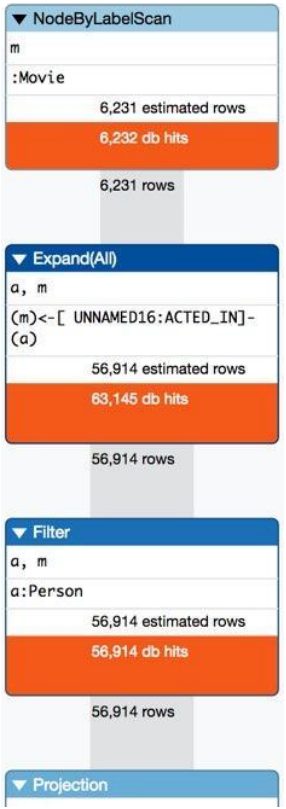
1. Check query string against cache
2. Auto-Parameterize (parse + replace literals with parameters)
3. Check statement string again
4. Parse and Rewrite
5. Check against AST cache
6. Compile
7. Add to cache(s)
8. Execute

To utilize the caches:
Use statements with *parameters* and the *same simple structure*

Execution Plan, AKA Profile

Clauses are decomposed into **multiple** operators which have:

- **input rows**
- **output rows**
- perform database operators, represented by **db hits.**



Profile a query using EXPLAIN

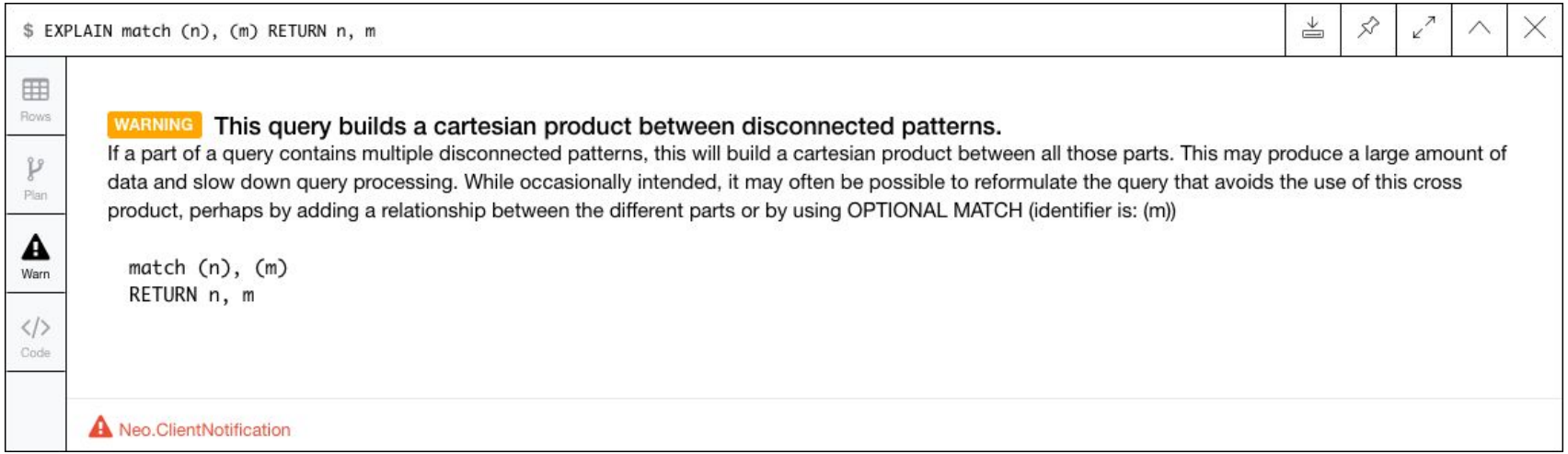
EXPLAIN shows the execution plan **without** actually executing it or returning any results.

It also does syntactic and semantic checking and generates warnings, provides **estimated** rows from **database statistics**.

EXPLAIN provides warnings

on cardinalities, unused symbols, eagerness, and more.

Pay Attention to these



Profiling a query: EXPLAIN vs PROFILE

EXPLAIN

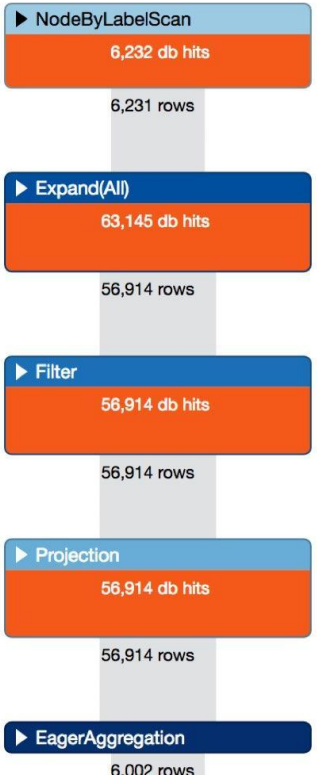
shows the execution plan **without** actually executing it or returning any results. Also does syntactic and semantic checking and generates warnings, provides **estimated** rows from **database statistics**.

PROFILE

executes the statement and returns the results along with profiling information, **actual** rows and **db-hits**

How do I profile a query?

```
PROFILE
MATCH (m:Movie)<-[:ACTED_IN]-(a:Person)
RETURN m.title, count(*) as cast
ORDER BY cast DESC
LIMIT 10
```

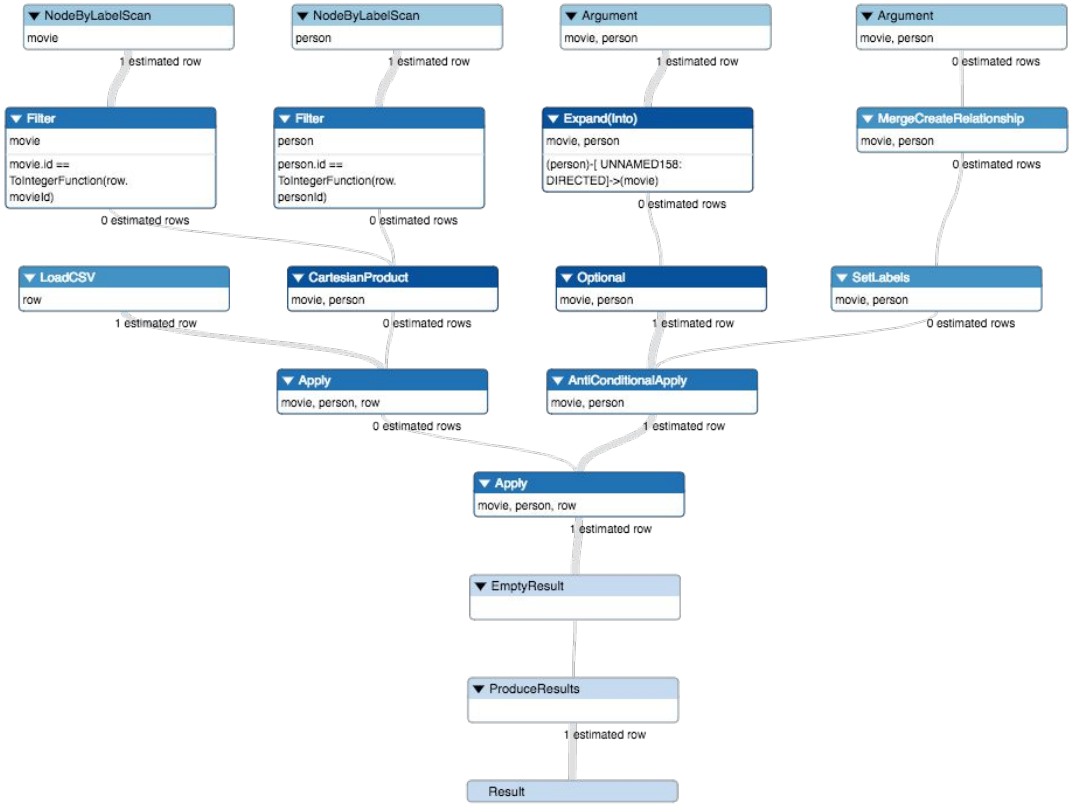



```
Cypher version: CYPHER 3.0, planner: COST, runtime: INTERPRETED.
183205 total db hits in 546 ms.
```

Example profile

```
EXPLAIN
LOAD CSV WITH HEADERS FROM {directors_url} AS row

MATCH (movie:Movie {id:toInteger(row.movieId)})
MATCH (person:Person {id:toInteger(row.personId)})
MERGE (person)-[:DIRECTED]->(movie)
ON CREATE SET person:Director
```

Example profile




```
EXPLAIN
LOAD CSV WITH HEADERS FROM {directors_url} AS row

MATCH (movie:Movie {id:toInteger(row.movieId)})
MATCH (person:Person {id:toInteger(row.personId)})
MERGE (person)-[:DIRECTED]->(movie)
ON CREATE SET person:Director;
```

The Counts-Store
Transactional Database Statistics

Introducing the Counts Store

Neo4j keeps **transactional database statistics**, which contain cardinalities of various graph entities:

- node counts for labels `(:Label)`
- outgoing / incoming per type and label
- `(:Label)-[:TYPE]->()`
- `()-[:TYPE]->(:Label)`
- Index selectivity 0.0 … 1.0 `(:Label {property})`

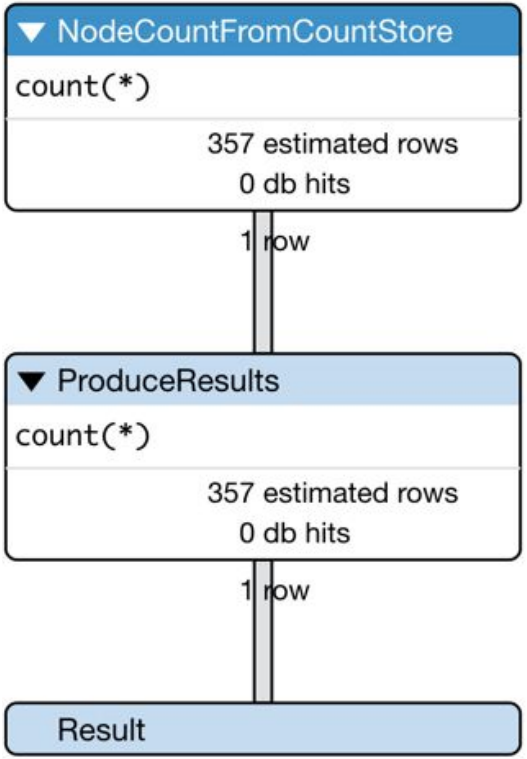
It uses these determine execution plans for queries.

What gets counted?

```
MATCH (n)
MATCH (n:Label)
MATCH ()-[r]->()
MATCH (:Label1)-[r]->()
MATCH ()-[r]->(:Label1)
MATCH ()-[r:X]->()
MATCH (:Label1)-[r:X]->()
MATCH ()<-[r:X]-(:Label1)
```

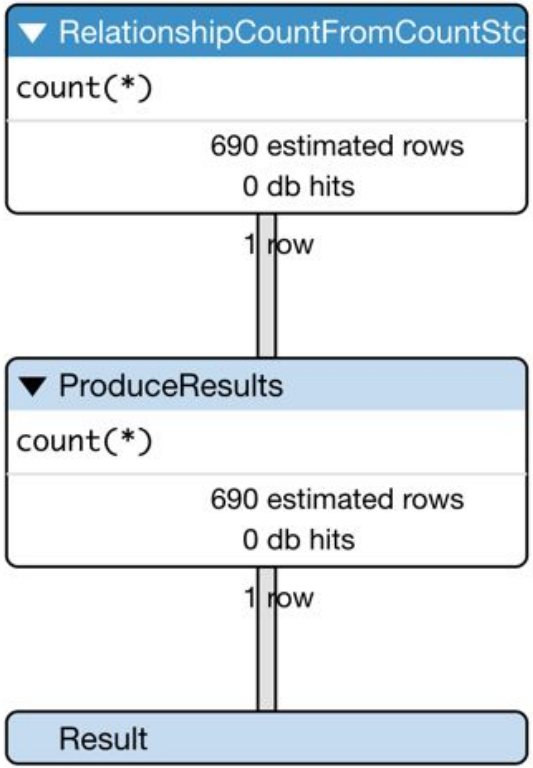
NodeCountFromCountStore

```
MATCH (n)
RETURN count(*)

MATCH (n)
RETURN count(n)

MATCH (n:Label)
RETURN count(*)
```



RelationshipCountFromCountStore

```
MATCH ()-->()
RETURN count(*)

MATCH (:Label)-->()
RETURN count(*)

MATCH ()-->(:Label)
RETURN count(*)
```



Count Store Tricks

Only works with no other aggregation key

Works with `count(*)` and `count(n)`

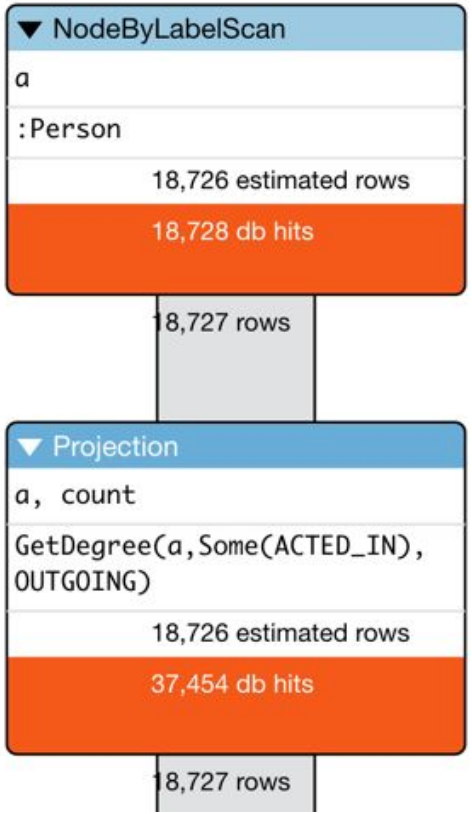
Solution: `UNION ALL` of Maps

```
MATCH (:Person) RETURN {person:count(*)} as count
UNION ALL
MATCH (:Movie)  RETURN {movie:count(*)} as count
```

Node Degree - GetDegree

Neo4j stores the degrees of dense nodes in the rel-group-store.

```
MATCH (a:Person)
RETURN a,
       size( (a)-[:ACTED_IN]->() ) as count
ORDER BY count DESC
LIMIT 10
```



The Counts-Store
Try it out:

```
:play
```

```
http://guides.neo4j.com/advanced_cypher
```

Query Tuning Goal

What's our goal?

At a high level, the goal is simple: get the numbers down

How?

- database hits
- rows
- runtime

Keep it lazy.

What is a database hit?

neo4j

> “an abstract unit of storage engine work.”

Operators cause database hits

Operators cause db-hits

Each operator generates db hits, they are **multiplied by input rows**

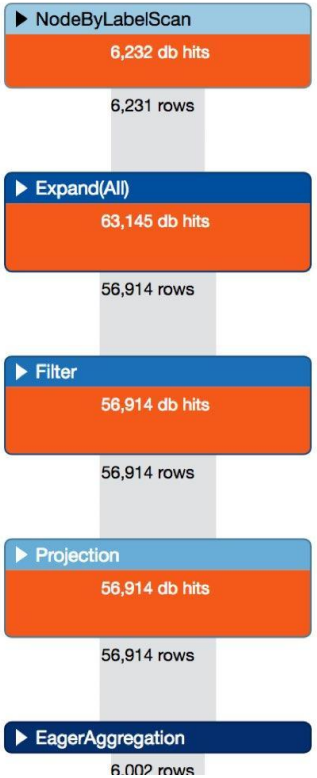
Some operators have a lower cost.

e.g.

- Exchange Expand(All) with **Expand(Into)**
- Exchange Expand(All) + Count with **GetDegree()**
- Exchange Aggregation on unique property with Aggregation on **entity**
- Remove label check if relationship-type is specific enough

How do I profile a query?

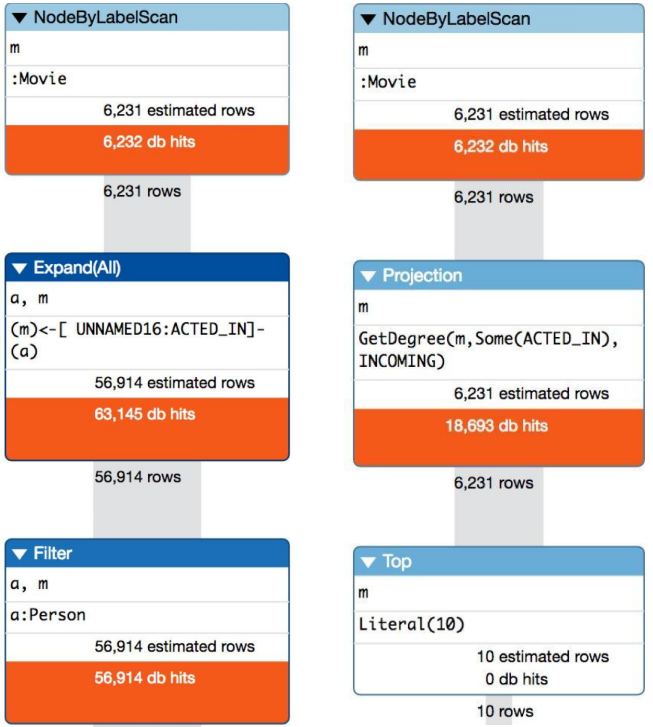
```
PROFILE
MATCH (m:Movie)<-[:ACTED_IN]-(a:Person)
RETURN m.title, count(*) as cast
ORDER BY cast DESC
LIMIT 10
```




```
Cypher version: CYPHER 3.0, planner: COST, runtime: INTERPRETED.
183205 total db hits in 546 ms.
```

Example: Expand -> GetDegree

```
PROFILE
MATCH (m:Movie)
RETURN m.title,
       size( (m)<-[:ACTED_IN]-() )
       AS cast
ORDER BY cast DESC
LIMIT 10
```




```
Cypher version: CYPHER 3.0, planner: COST,
runtime: INTERPRETED. 24925 total db hits in 207 ms.
```

Rows (Cardinalities)

Controlling Cardinalities

Input rows into an operator
= **times** the operator is executed
= multiplies the number of db-hits and individual **output rows**

Getting output rows down, reduces input rows for next operator

e.g. you're only interested in distinct results, not number of paths,
get the data/rows down to distinct in-between values.

Or get the number of times a index lookup is done down.
Or use the label or index with the higher selectivity to drive the query.

Example: Cardinalities - Cross Product

Avoid Cross Products like this

```
MATCH (m:Movie), (p:Person)
RETURN count(distinct m), count(distinct p)
```

Example: Cardinalities - Cross Product

Avoid Cross Products like this

```
MATCH (m:Movie), (p:Person)
RETURN count(distinct m), count(distinct p)
```

Instead do one MATCH at a time

```
MATCH (m:Movie) WITH count(*) AS movies
MATCH (p:Person) RETURN movies, count(*) AS people
```

Example: Cardinalities - Cross Product

Avoid Cross Products like this

```
MATCH (m:Movie), (p:Person)
RETURN count(distinct m), count(distinct p)
```

Instead do one MATCH at a time

```
MATCH (m:Movie) WITH count(*) AS movies
MATCH (p:Person) RETURN movies, count(*) AS people
```

Or use the count store

```
MATCH (m:Movie) RETURN {movies: count(*)} AS c UNION ALL
MATCH (p:Person) RETURN {people: count(*)} AS c
```

Controlling Cardinalities Example

Assume 100 actors per Movie.
Even if we're just interested in distinct movies, we touch **1M paths**.

```
(p:Person)-[:ACTED_IN]->(m:Movie)<-[:ACTED_IN]-(p2:Person)-[:ACTED_IN]->(m2:Movie)
    1          x100         100         x100          10000       x100        1000000

(p:Person)-[:ACTED_IN]->(:Movie)<-[:ACTED_IN]-(p2)
   1          x100         100         x100       10000
WITH DISTINCT p2
               100
(p2:Person)-[:ACTED_IN]->(m2:Movie)
    100         x100         10000
```

If we keep distinct data in between, we touch **10k paths**, i.e. 100 times less.

Controlling Cardinalities Example

```
PROFILE
MATCH  (p:Person)-[:ACTED_IN]->(m:Movie)
                <-[:ACTED_IN]- (p2:Person)
                 -[:ACTED_IN]->(m2:Movie)
WHERE p.name = "Tom Hanks"

RETURN count(*), count(distinct m2)
```

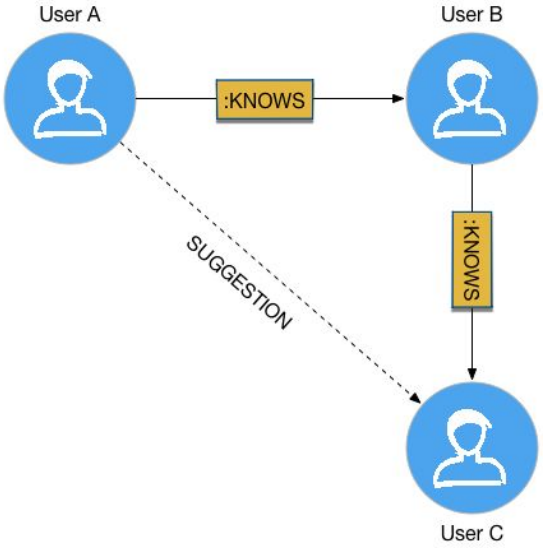
Cypher version: CYPHER 3.0, planner: COST,
runtime: INTERPRETED. **11435 total db hits** in 138 ms.

Controlling Cardinalities Example

```
PROFILE
MATCH  (p:Person)-[:ACTED_IN]->(m:Movie)
                <-[:ACTED_IN]- (p2:Person)

WHERE p.name = "Tom Hanks"

WITH DISTINCT p2

MATCH  (p2)-[:ACTED_IN]->(m2:Movie)
RETURN count(*), count(distinct m2)
```

Cypher version: CYPHER 3.0, planner: COST,
runtime: INTERPRETED. **9671 total db hits** in 60 ms.

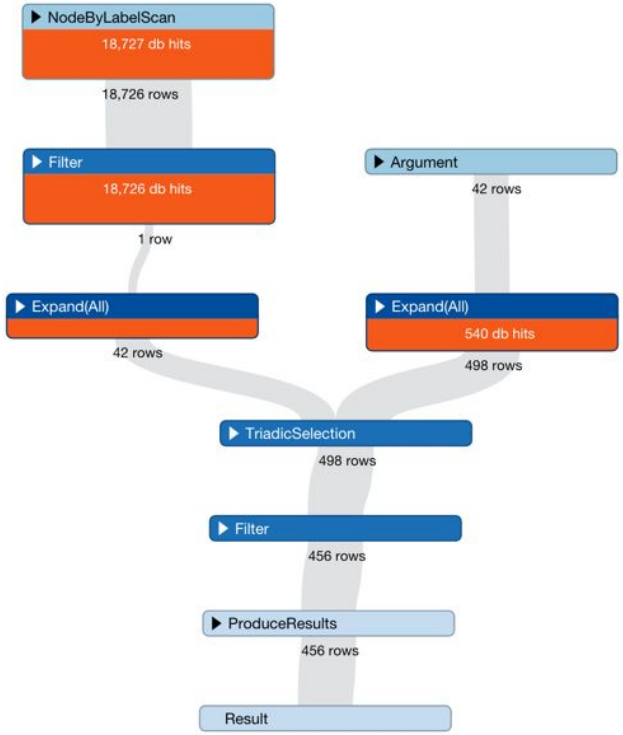
Triadic Selection



- Recommendations form triangles
- Up to 2 steps out
- Check against elements contained in first step

Triadic Selection

```
MATCH (a:User {name:"A"})
      -[:KNOWS]->(b)-[:KNOWS]->(c)
WHERE a <> c
AND NOT (a)-[:KNOWS]->(c)
RETURN c
```



Caveats:

- need same label
- same relationship-type and direction
- only works on 2-step patterns

Work In Progress (Memory)

Work in Progress

Rows represent the “work in progress“ data, which consumes memory.

Cypher is lazily streaming except when using:

- Sorting
- Aggregation
- Distinct
- Updating (Transaction State)
- Eager

Example

Reduce work in progress by:

- only keep distinct values
- access properties late
- “band filter“ potential inputs (e.g. remove top and bottom 25%)
- use sorting with limit (Top-K window select)
- used index or label with high selectivity to start from

Example

Reduce work in progress by:

- only keep distinct values
- access properties late
- “band filter“ potential inputs (e.g. remove top and bottom 25%)
- use sorting with limit (Top-K window select)
- used index or label with high selectivity to start from

```
MATCH (:Movie) WITH count(*) as movies
MATCH (u:User)
WITH u,
     size( (u)-[:RATED]->() ) as ratings,
     movies * 0.8 as upperBound
WHERE 20 < ratings < upperBound
```

Updating Data (Transaction Size)

Transaction Size

- Neo4j and Cypher are **transactional**
- Isolation requires to keep Transaction-State (delta changes) separate
- Tx-State consumes memory
- *roughly* 1M updates work with 4G heap

- especially critical with:
  - Graph Refactorings
  - Larger Import (Periodic Commit, Eager)
  - Larger Computation / Mass Updates

Transaction Size

- `USING PERIODIC COMMIT` with `LOAD CSV`
- Batch Graph Updates
  - externally
  - with sliding window
  - with pre-condition filter
  - with a batching procedure (`apoc.periodic.iterate`)

Pre-Condition Filter

```
MATCH (m) WHERE not exists (m)-[:GENRE]->()
WITH m LIMIT 10000
UNWIND m.genres as genre
MATCH (g:Genre {name: genre})
CREATE (m)-[:GENRE]->(g)
RETURN count(*);
```

Faster map lookup

We can do even better by populating a map and using that to lookup the genres.

Faster map lookup

```
MATCH (g:Genre)
WITH apoc.map.fromPairs(collect([g.name, g])) AS genres

MATCH (m) WHERE not exists (m)-[:GENRE]->()
WITH genres, m LIMIT 10000
UNWIND m.genres as genre
WITH genres[genre] as g, m
CREATE (m)-[:GENRE]->(g)
```

Mark with Label, Process Batch Window

```
// mark nodes to process
MATCH (m:Movie) SET :Process;

// process marked nodes in batch
MATCH (m:Movie:Process) WITH m LIMIT 10000
REMOVE m:Process // remove mark
UNWIND m.genres as genre
MATCH (g:Genre {name: genre})
CREATE (m)-[:GENRE]->(g)
RETURN count(*);
```

Batching Procedure

```
CALL apoc.periodic.iterate("
  MATCH (m:Movie) RETURN m // driving statement
","
  WITH $m AS m
  UNWIND m.genres as genre // processing statement
  MATCH (g:Genre {name: genre})
  CREATE (m)-[:GENRE]->(g)
",{batchSize:10000})       // 10k per transaction
```

Selectivity

Selectivity

- Start with the smallest starting set
- Label with fewest members
  - e.g. `:Actor` instead of `:Person`
- Relationship with smallest degree
- Index with highest selectivity

Index Lookups (Performance)

Index Lookups

- For Lookups of individual nodes
- Neo4j uses an index provider (Lucene) for
  - lookups by label
  - lookup by property
  - range scans and text lookups
- Fast for individual lookups
- **Many million lookups** (double for relationship-creation) add up
- Lookups by node and relationship id are very fast, but shouldn‘t be used outside of a narrow scope (e.g. write back computation results)

Tips - Index Lookups

Use `EXPLAIN` and `PROFILE` to see if indexes are used

Provide hints: `USING INDEX n:Label(name)`

Make sure **most selective** index is used

Cypher uses by default **only one index** per pattern.

- Index hints force it to use multiple

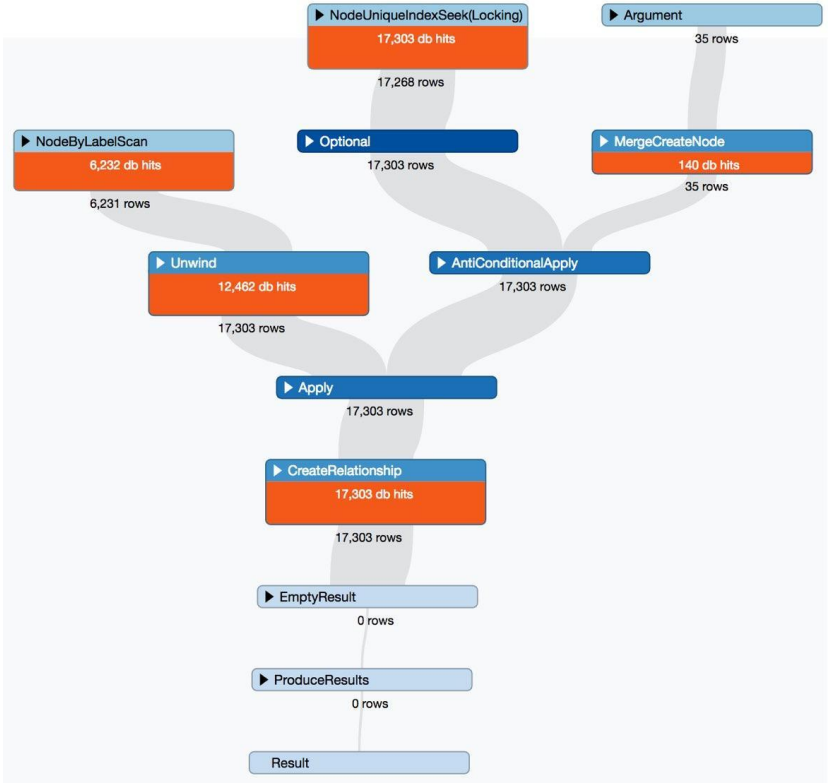
Keep the left side just the **plain indexed property**

- Move all computation to the other side

```
MATCH (g:Genre) WHERE g.name CONTAINS {adjective} + " drama"
```

Index Lookups: Example

```
MATCH (m:Movie)
UNWIND m.genres as genre

// 1 genre lookup per movie-genre
MERGE (g:Genre {name: genre})
CREATE (m)-[:GENRE]->(g);
```



Cypher version: CYPHER 3.0, planner: COST, runtime: INTERPRETED. 53440 total db hits in 1655 ms.

Index Lookups: Example

```
MATCH (m:Movie)
UNWIND m.genres AS genre
WITH genre, collect(m) AS movies
// 1 lookup per genre
MERGE (g:Genre {name: genre})
WITH g, movies
UNWIND movies AS m
CREATE (m)-[:GENRE]->(g);
```

```
Cypher version: CYPHER 3.0,
planner: COST, runtime: INTERPRETED.
36173 total db hits in 2388 ms.
```

Index Lookups: Multi faceted Search

Do **multiple** separate index lookups

Declare the nodes to be the **same**

```
CREATE INDEX ON :Movie(title);
CREATE INDEX ON :Movie(releaseYear);

MATCH (m:Movie) WHERE m.title CONTAINS "The"
MATCH (n:Movie) WHERE 1990 < n.releaseYear < 2000 AND n = m
RETURN m;
```

Caches & Warmup

Caches & Warmup

- Configuration: Heap + Page-Cache
  - Page-Cache possibly datastore-size
  - Heap: 8, 16, 32G depending on operations
- Caches keep active dataset available
  - Query Plan Cache
    - Run query once, e.g. with EXPLAIN
    - **Use Parameters**
  - Page-Cache - memory mapping datastore from disk
  - warmup with
    - all nodes / all relationships operation / properties if needed
    - MATCH (n) RETURN max(id(n))
    - apoc.warmup.run() - node & relationship-pages

End of Module
Cypher Query Tuning