What are we going to learn?

- Stepwise query tuning

- How the Cypher query planner works

- Query Profiling

- Operation Costs, Cardinalities and Rows

- Batching Transaction

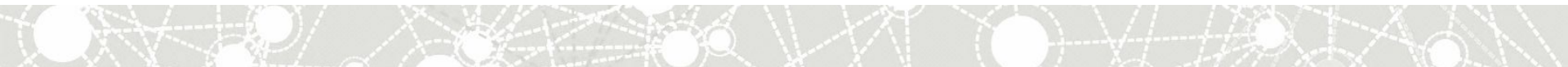- Index Lookup performance

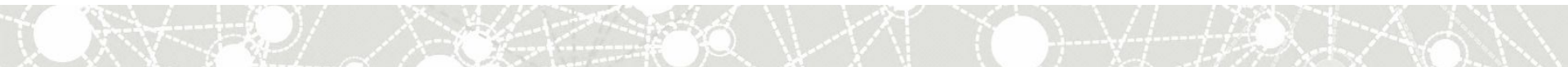Steps for Query Tuning

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:** Focussing on hotspots, build query up, applying tuning techniques

Let's look at some of the query tuning tools

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
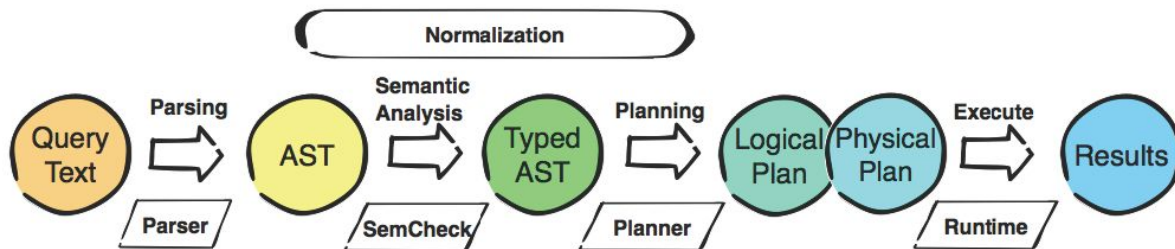- operation improvements
- new operations

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)
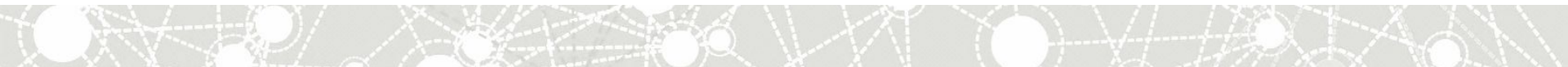6. Execute with interpreted or compiled runtime

Query Plan Cache

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

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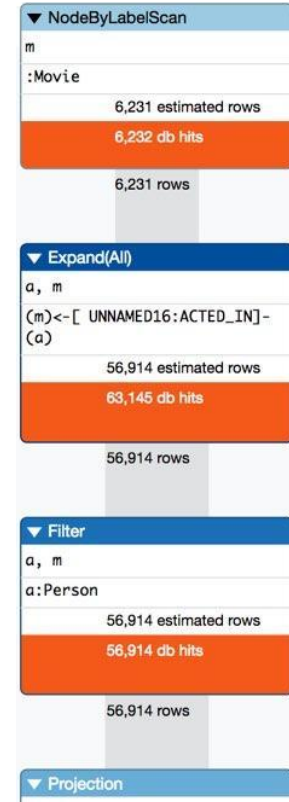Use statements with *parameters*
and the *same simple structure*

Execution Plan, AKA Profile

Clauses are decomposed into **multiple** operators which

have:

- **input rows**

- **output rows**

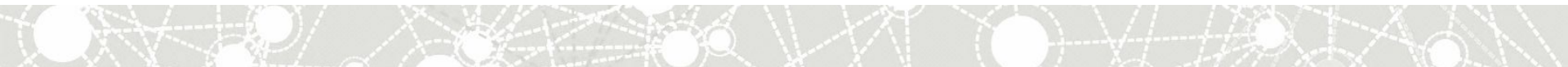- perform database operators, represented by **db hits.**

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

$ explain match (n), (m) return m, n

Rows

Plan

Warn

Code

WARNING This query builds a cartesian product between disconnected patterns.
If a part of a query contains multiple disconnected patterns, this will build a cartesian product between all those parts. This may produce a large amount of data and slow down query processing. While occasionally intended, it may often be possible to reformulate the query that avoids the use of this cross product, perhaps by adding a relationship between the different parts or by using OPTIONAL MATCH (identifier is: (m))

```
explain match (n), (m) return m, n
```

⚠ Neo.ClientNotification

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**

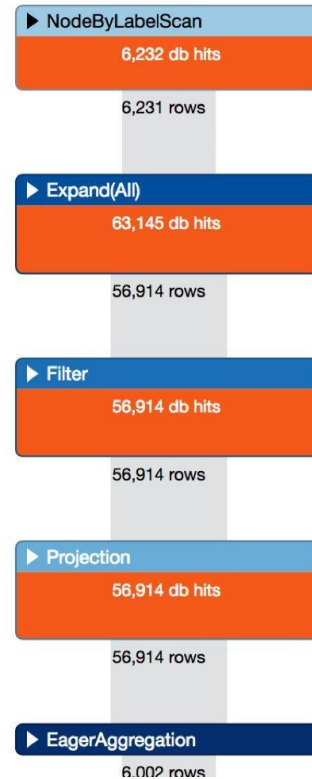# How do I profile a query?

```cypher
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;
```
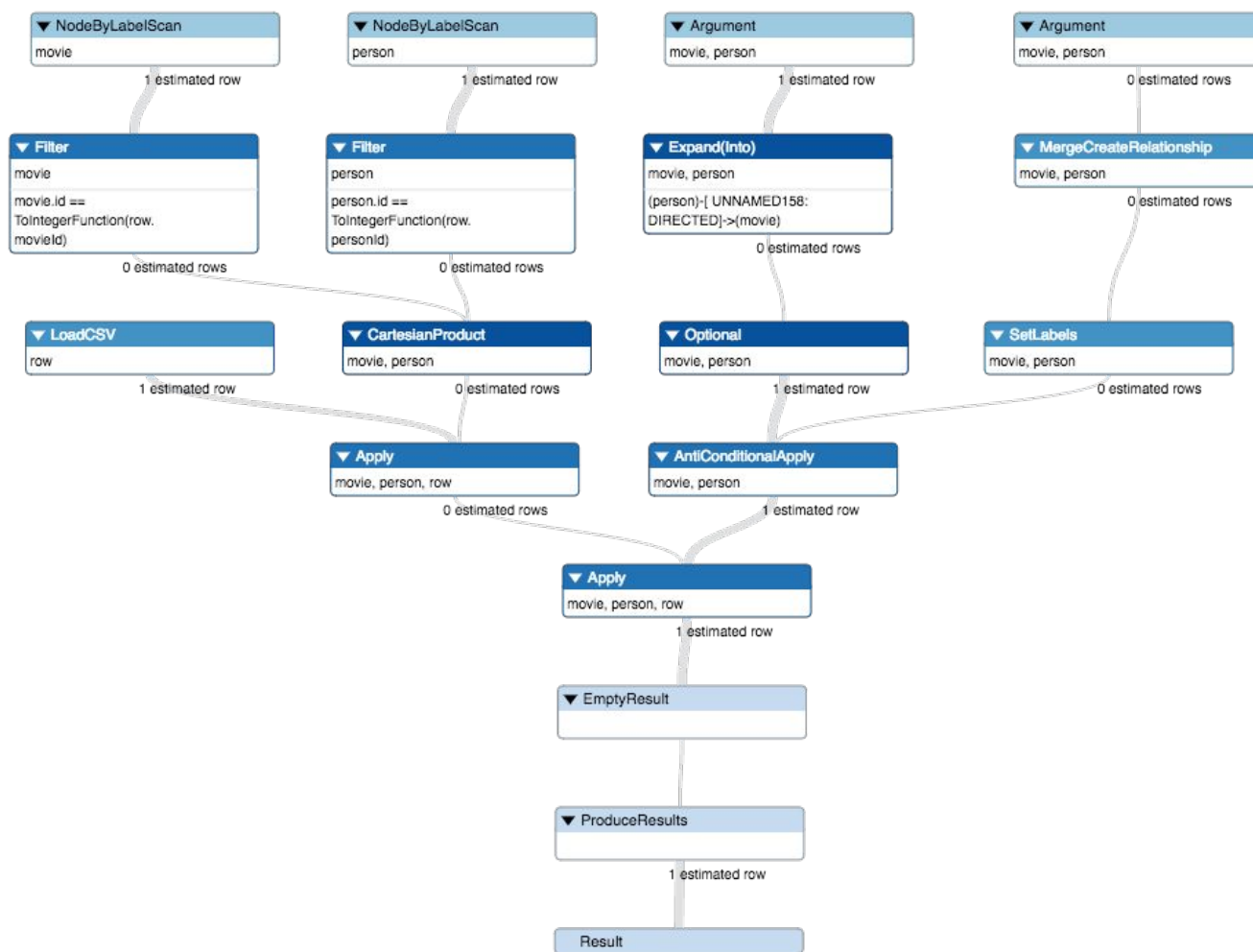
At a high level, the goal is simple:

get the numbers down

- **database hits**

- **rows**

- **runtime**

Keep it lazy.

"
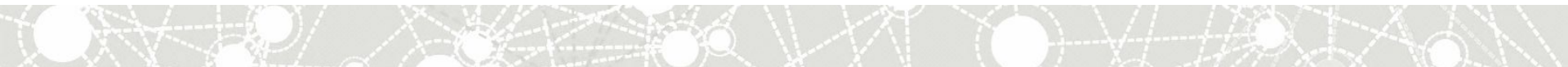
an abstract unit of storage
engine work.

"

Operators cause Database Hits
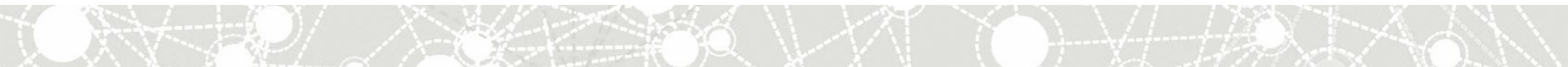
Operators cause db-hits

Each operator has db-hits associated,
 they are **multiplied by input rows**

Improve by switching operator for a cheaper one

Examples:
- 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.
```


▶ NodeByLabelScan
6,232 db hits
6,231 rows

▶ Expand(All)
63,145 db hits
56,914 rows

▶ Filter
56,914 db hits
56,914 rows

▶ Projection
56,914 db hits
56,914 rows

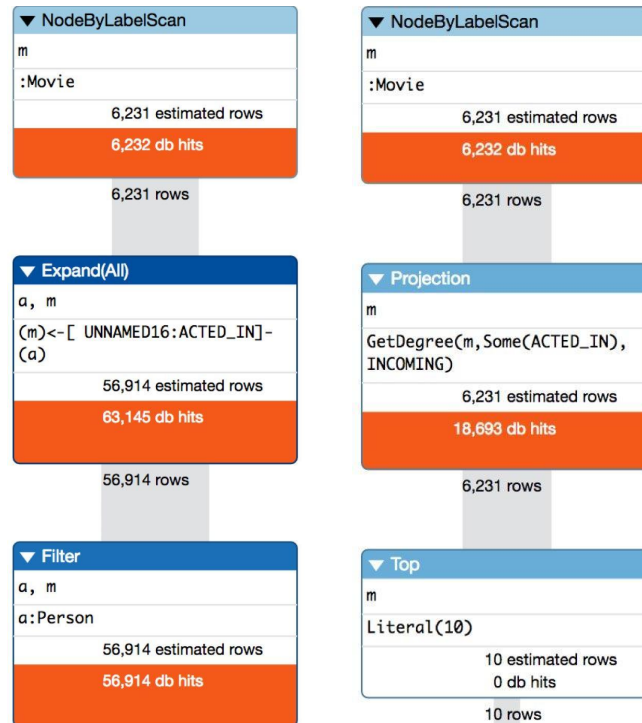▶ EagerAggregation
6,002 rows
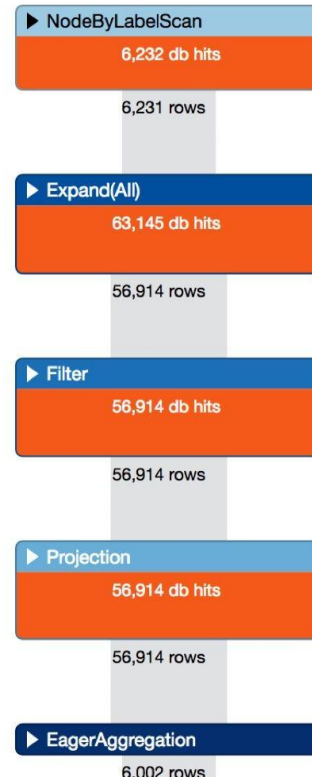
Example: Expand -> GetDegree

```
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.

Left query plan

NodeByLabelScan
m
:Movie
6,231 estimated rows
6,232 db hits

6,231 rows

Expand(All)
a, m
(m)<-[UNNAMED16:ACTED_IN]-(a)
56,914 estimated rows
63,145 db hits

56,914 rows

Filter
a, m
a:Person
56,914 estimated rows
56,914 db hits

Right query plan

NodeByLabelScan
m
:Movie
6,231 estimated rows
6,232 db hits

6,231 rows

Projection
m
GetDegree(m,Some(ACTED_IN), INCOMING)
6,231 estimated rows
18,693 db hits

6,231 rows

Top
m
Literal(10)
10 estimated rows
0 db hits

10 rows

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.

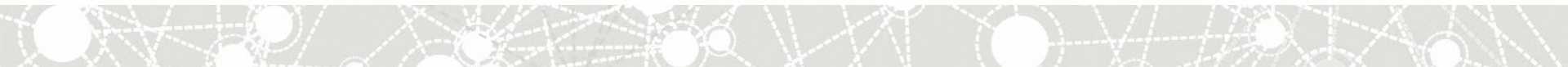# Example: Cardinalities - Cross Product

Avoid Cross Products like this

   MATCH (m:Movie), (p:Person)

   RETURN count(distinct m), count(distinct p)

Instead do one MATCH at a time

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

Better, use 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 cast 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.
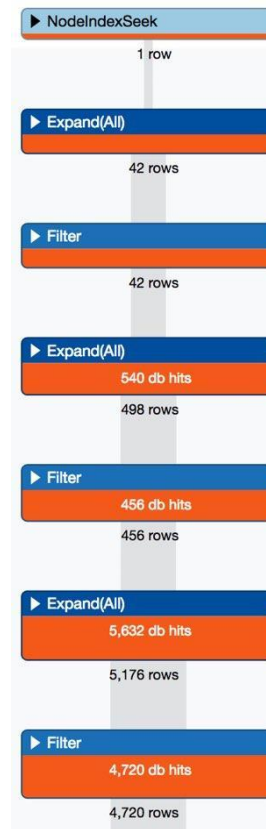
Controlling Cardinalities Example

```
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.

▶ NodeIndexSeek
1 row

▶ Expand(All)
42 rows

▶ Filter
42 rows

▶ Expand(All)
540 db hits
498 rows

▶ Filter
456 db hits
456 rows

▶ Expand(All)
5,632 db hits
5,176 rows

▶ Filter
4,720 db hits
4,720 rows

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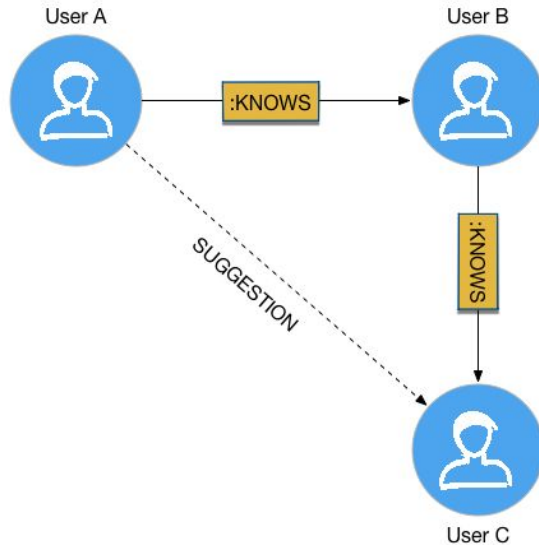
Cypher version: CYPHER 3.0, planner: COST,
runtime: INTERPRETED. **9671 total db hits** in 60 ms.

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

http://graphaware.com/neo4j/2015/10/20/faster-recommendation-graphaware-reco-neo4j-triadic.html
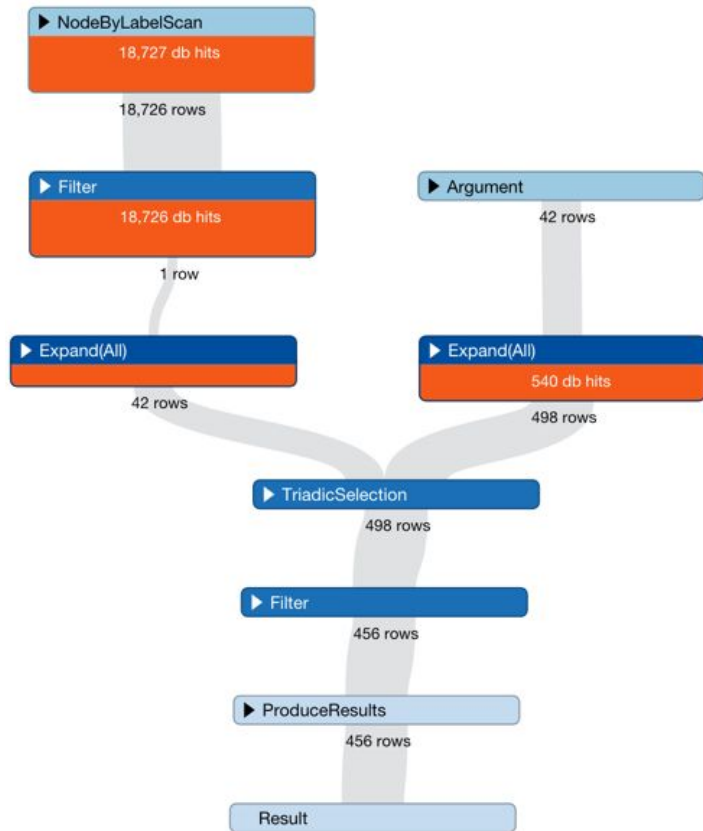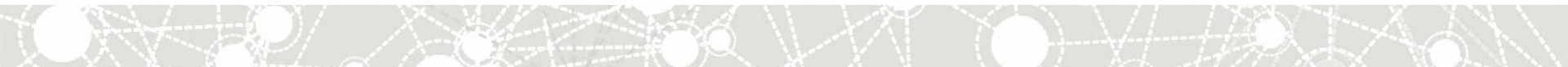
Work In Progress (Memory)

Work in Progress

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

Usually Cypher is lazily streaming, i.e. doesn't consume additional memory, 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

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

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

- Esp. 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(*);
```

- better
  - use degree = 0
  - use map for lookup of genre

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(*);
```
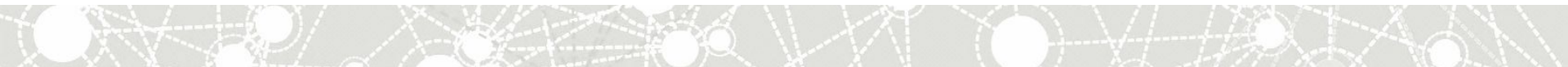
Batching Procedure

```
CALL apoc.periodic.iterate("
  MATCH (m:Movie) RETURN m // driving statement
","
  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
  - for 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

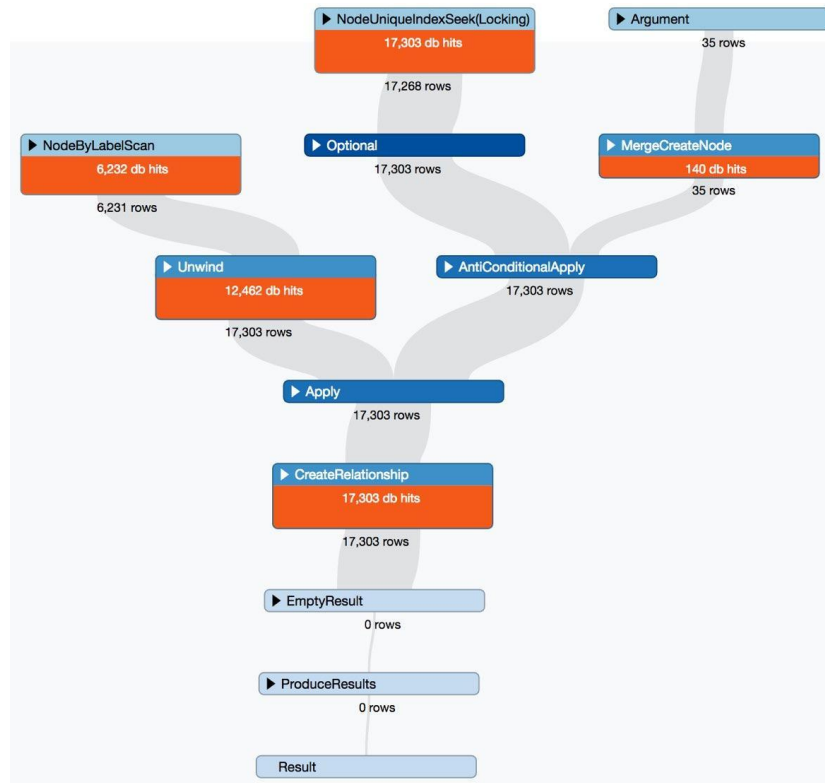Keep the left side just the **plain indexed property**

Movie 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