



ACTiCLOUD: ACTivating resource efficiency and large databases in the  
CLOUD

Project No: 732366

H2020-ICT-2016-1

D3.8: Neo4j ACTiCLOUD extensions v2.0

<b>Due date of deliverable:</b>	<b>M32 (2019/08/31)</b>
Actual submission date:	M33 (2019/09/18)

**Executive summary:**

This document describes the evolution of the Neo4j graph database within the context of the ACTiCLOUD project. This deliverable is not independent but relies on Deliverable D3.5: “ACTiCLOUD-enabled system libraries”

**List of authors:**

Authors	Affiliation
Jim Webber	NEO
Pontus Melke	NEO
Alex Averbuch	NEO

<b>Dissemination Level</b>	<input checked="" type="checkbox"/>	<b>PU (Public)</b>
	<input type="checkbox"/>	PP (Restricted to other programme participants)
	<input type="checkbox"/>	RE (Restricted to a group specified by the consortium)
	<input type="checkbox"/>	CO (Confidential, only for members of the consortium)
	Where restricted, access granted to:	
<b>Nature</b>	<input type="checkbox"/>	R (Report)
	<input type="checkbox"/>	P (Prototype)
	<input type="checkbox"/>	D (Demonstrator)
	<input checked="" type="checkbox"/>	<b>O (Other)</b>

<b>Review Status</b>	<input type="checkbox"/>	Draft
	<input type="checkbox"/>	WP Leader accepted
	<input type="checkbox"/>	QA approved
	<input checked="" type="checkbox"/>	<b>Coordinator accepted</b>

**Revision History:**

Version	Author(s) (Affiliation)	Notes
0.1	Jim Webber	TOC and initial content written
1.0	Jim Webber	Final Version
1.1	Vasileios Karakostas (ICCS)	Submitted Version

**ACTiCLOUD Consortium:**

Participant No	Participant organisation name	Short name	Country
1 (Coordinator)	Institute of Communication and Computer Systems	ICCS	Greece
2	Numascale AS	NSCALE	Norway
3	Kaleao Limited	KALEAO	UK
4	OnApp Limited	ONAPP	Gibraltar
5	University of Manchester	UNIMAN	UK
6	MonetDB Solutions B.V.	MDBS	Netherlands
7	Neo Technology	NEO	Sweden
8	UMEA University	UMU	Sweden



NUMSCALE



**Confidentiality:**

This document contains proprietary and confidential material of certain ACTiCLOUD contractors, and may not be reproduced, copied, or disclosed without appropriate permission. The commercial use of any information contained in this document may require a license from the proprietor of that information.

THIS DOCUMENT IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES

(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS DOCUMENT, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

**Table of Contents**

<b>1</b>	<b>Introduction .....</b>	<b>8</b>
1.1	Purpose of this Document .....	8
1.2	Graph Databases.....	8
<b>2</b>	<b>Alignment with ACTiCLOUD’s objectives, business scenarios and use cases.....</b>	<b>10</b>
<b>3</b>	<b>Evolution of Neo4j for ACTiCLOUD .....</b>	<b>12</b>
3.1	Cypher .....	15
3.2	Lifecycle of a Cypher Query .....	16
3.3	Physical Planning.....	17
3.4	Physical Compilation.....	20
3.4.1	Compiled Expressions .....	21
3.4.2	Fused Pipelines .....	22
3.4.3	Fused Example .....	22
3.4.4	Non-fused (operator-at-a-time) Example.....	23
3.4.5	Pull (row-at-a-time) Example.....	24
3.5	Execution .....	25
3.5.1	Push vs Pull .....	26
3.5.2	Batched Execution .....	26
3.5.3	Operator Fusing .....	26
3.5.4	Terminology .....	28
3.5.5	Parallel Execution.....	29
3.5.6	Design.....	29
3.5.7	Reactive API / Back Pressure .....	32
<b>4</b>	<b>Interim Evaluation.....</b>	<b>33</b>
4.1	Query 1 .....	34
4.2	Query 2 .....	35
4.3	Query 3 .....	35
4.4	Query 4 .....	36
4.5	Query 5 .....	37
<b>5</b>	<b>Software Assets.....</b>	<b>38</b>
<b>6</b>	<b>Next Steps.....</b>	<b>38</b>

**Figures**

Figure 1: A graph of money laundering criminals and institutions.....	8
Figure 2: Base ACTiCLOUD architecture. ....	13
Figure 3: Example Query.....	16
Figure 4: The life of a Cypher Query. ....	17
Figure 5: For a given query we find all variables and parameters of the query - here n, r, c, and param - and allocate dedicated memory slots. Whenever we read or write, we access the directly accessed memory (array) instead of accessing the computing their location (hashmap) . ....	18
Figure 6: Breaking up a Logical Plan into pipelines. ....	20
Figure 7: Expression Tree. ....	21
Figure 8: Executing Pipeline 3 in normal mode (left) and fused (right). ....	28
Figure 9: Example execution. ....	31
Figure 10: Runtime Performance.....	34

**List of Abbreviations**

Abbreviation / Acronym	Meaning
JVM	Java Virtual Machine
LBDC	Linked Data Benchmark Council ( <a href="http://ldbouncil.org/">http://ldbouncil.org/</a> )
RDBMS	Relational Database Management System
OLTP	Online Transaction Processing
OLAP	Online Analytical Processing

# 1 Introduction

## 1.1 Purpose of this Document

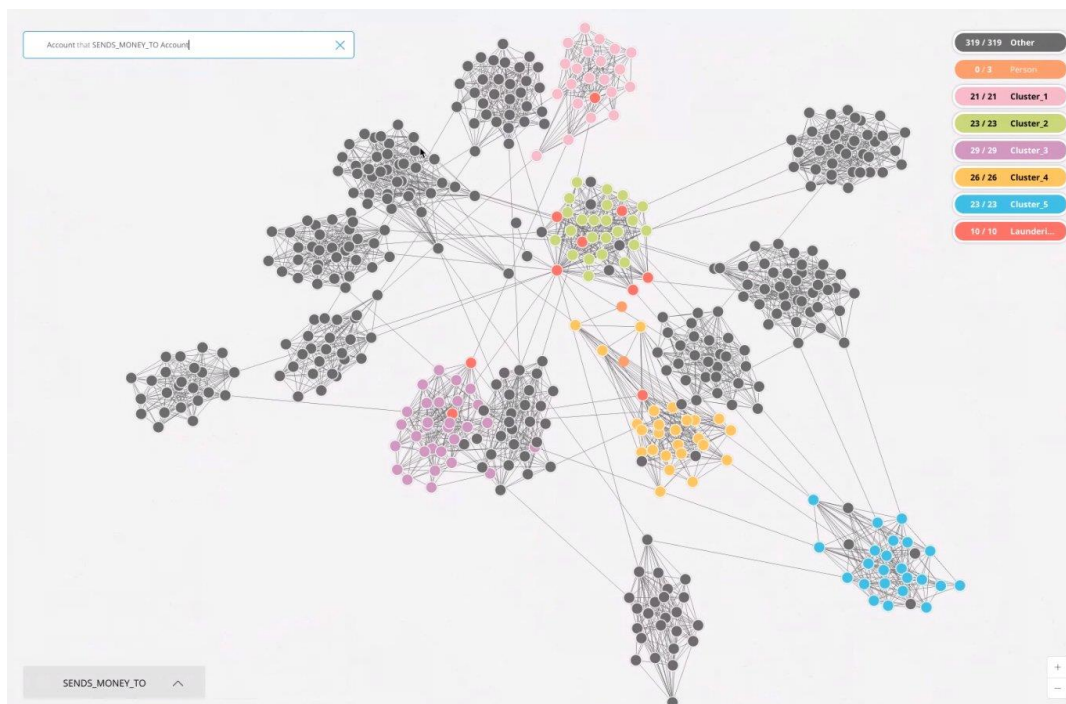
This document describes the evolution of the Neo4j graph database within the context of the ACTiCLOUD project. It establishes the wider business context which drives technical changes in the implementation of Neo4j as part of the ACTiCLOUD stack. Further, the document details the research and development challenges and achievements within Neo4j that enable it to exploit the enhanced compute and storage resources from the ACTiCLOUD platform.

## 1.2 Graph Databases

A graph is a data structure which consists of vertices (nodes) connected by edges (relationships). The labelled property graph model advances this basic model by:

- Allowing nodes and relationships to store property data;
- Ensuring relationships are directed and typed to capture the association between the two nodes they connect;
- Nodes have zero or more labels to distinguish their role in the network.

Using these simple constructs, we can represent many problem domains simply, intuitively and in high fidelity.



**Figure 1:** A graph of money laundering criminals and institutions.



The labelled property graph model provides both modelling clarity as we see in Figure 1 and opportunities for performance that outstrip other technical approaches such as RDBMS, and make graphs a versatile approach for general-purpose data processing.

The simplicity and performance of graph databases like Neo4j has helped to make them the fastest-growing database category from 2012 to present according to analysts at <http://db-engines.com>. Graph databases allow end-users to deliver systems rapidly with improved performance and agility.

## 2 Alignment with ACTiCLOUD’s objectives, business scenarios and use cases

The specific objectives achieved are presented in Table 1.

Table 1: Objectives and Achievements to-date

ACTiCLOUD Strategic Objectives	ACTiCLOUD Strategic sub-Objectives	Mechanical Sympathy	Parallel Runtime
SO1: Effective utilisation of cloud resources.	SO1.1: Resource efficiency	✓	✓
	SO1.2: Performance stability	✓	✓
SO2: Deployment of resource demanding applications in the cloud	SO2.1: Scalability in resource provisioning	✓	✓
	SO2.2: Elasticity in resource provisioning	N/A	N/A

The critical achievement in this deliverable is that Neo4j can scale to challenging OLTP workloads and large OLAP graph analytic queries. The changes in Neo4j and the ACTiCLOUD stack allow access to large aggregate compute and memory for such resource intensive workloads.

While the database could, in principle, be engineered over time as a highly distributed platform, there are well-known drawbacks pertaining to performance and efficiency pertaining to distributed graph processing<sup>1</sup>. Instead ACTiCLOUD provides cloud middleware that allows a single large machine to be presented to the database, and the re-engineering of Neo4j allows to database to take advantage of that large aggregate machine through parallel processing of queries.

In building an ACTiCLOUD-enhanced version of Neo4j we aim to achieve:

- Access to large aggregate RAM and CPU cores which are only physically and economically feasible via cloud infrastructure; And thus:
- The ability to perform large graph analyses which were previously too demanding for the single threaded computation.
- We demonstrate this with a benchmark which mimics both business intelligence and online transaction processing in the graph.

<sup>1</sup> For example, DSE Graph published benchmarks on a large server farm which Neo4j easily beat by several orders of magnitude on a single thread on a desktop computer, see <http://graphconnect.com>.

The ACTiCLOUD platform should make big graph data a competitive reality for Neo4j and its users, especially for large graph analyses which consume a great deal of RAM and cores. Since graphs are the foundation for much of the interesting future work in data, we believe that ACTiCLOUD's scalable resource provisioning scheme should enhance Neo4j's capabilities and advance its business.

### 3 Evolution of Neo4j for ACTiCLOUD

Neo4j is a graph database which stores data as a network of (labelled) nodes and (typed, directed) relationships each containing key-value property data. The so-called labelled property graph model is a performant and high-fidelity model.

In contrast to RDBMSs, Neo4j does not use relational *joins* across tabular data since that approach scales poorly with both data size and number of joins. Instead, Neo4j queries *traverse* the graph via pointer chasing which is mechanically sympathetic<sup>2</sup> to the underlying computer system (and respectful of the memory hierarchy). When a graph can be held in working memory, traversing it is extremely performant.

Ultimately Neo4j is responsible for the safe storage (via transactions) and rapid query of graph data (via traversals) while providing a humane interface onto that data for interactive users and application developers.

Neo4j's historic sweet spot is OLTP use cases where database queries are typically bound to a subgraph that can be cached in RAM and processed by a single thread. For highly demanding analytical workloads, however, Neo4j's single-machine architecture can struggle to match user demand. For example RAM sizes for commodity servers are typically smaller than overall graph size for the most demanding workloads. Furthermore historically Neo4j has been unable to use all available cores on a server to answer a single query, instead offering one-core-per-query from its OLTP roots.

It is clear that very large graph analyses require both large amounts of RAM and CPU cores. While virtual memory ensures that queries can proceed, it is not nearly as fast as physical memory: as virtual memory is swapped in and out of physical memory during computation, performance degrades. Furthermore being confined to a single thread of execution limits the exploitation of modern multicore systems.

The Numascale servers in ACTiCLOUD provide both for large numbers of cores and lots of aggregate RAM. In principle, these address both of Neo4j's historic limitations and enable the database to support more ambitious workloads. In practice the ability of the underlying platform to provide memory locality across physical servers is paramount.

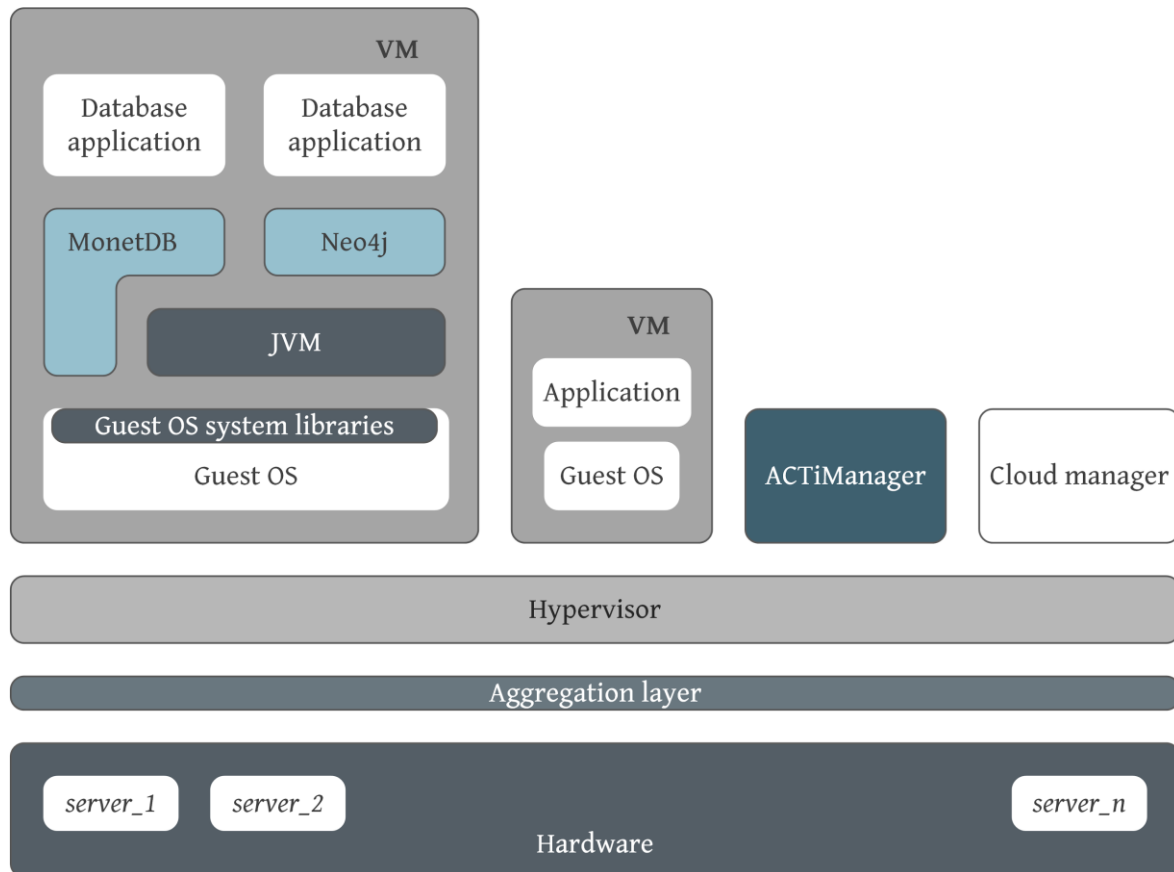
Neo4j is a graph database implemented in Java and runs on the JVM, both the industry standard Hotspot JVM and the experimental HyperScaleVM (HJVM) developed as part of ACTiCLOUD (in deliverable D3.6) from the University of Manchester. The combination of HJVM and Neo4j within ACTiCLOUD tests the hypothesis that a database can run atop a multi-computer JVM while treating the JVM as a single virtual machine which transparently handles thread dispatch and (remote) memory management.

As shown in Figure 2, architecturally Neo4j sits towards the top of the ACTiCLOUD stack, atop the JVM and beneath the application layer. In the case of ACTiCLOUD that application is a graph-

---

<sup>2</sup> See [https://wa.aws.amazon.com/wat.concept.mechanical\\_sympathy.en.html](https://wa.aws.amazon.com/wat.concept.mechanical_sympathy.en.html)

database benchmark derived from the LDBC<sup>3</sup>. The LDBC-like benchmark thoroughly exercises the database (and thus the ACTiCLOUD stack) in a way representative of a real production workload.



**Figure 2:** Base ACTiCLOUD architecture.

In evolving Neo4j to support online graph analytics alongside online transaction processing, a number of technical and research challenges need to be addressed. While the abundance of RAM and cores provided by ACTiCLOUD is a blessing, to take advantage of that abundance requires the redesign and implementation of a new set of runtime components within Neo4j, specifically:

1. Mechanically sympathetic runtime.  
While the JVM is generally a technical enabler, it is not a panacea. In particular the popular Hotspot JVM has a garbage collector which can pause or stop all progress (so called “stop-the-world semantics”). In such cases the Neo4j database appears to have stopped. We consider the stopped server to be unavailable, which is a fault that must be

<sup>3</sup> Linked Data Benchmark Council (LDBC) was originally an FP7 project. It is now a vendor-neutral body which produces benchmark specifications for graph databases.

tolerated through redundancy or better still avoided. To reduce aggressive garbage collection, within ACTiCLOUD Neo4j has implemented a Cypher<sup>4</sup> runtime which is sympathetic to the JVM and produces very little garbage and so drastically reduces the amount of garbage collection pauses that can occur. This is implemented as a vectorized runtime in which registers are allocated once per database instance and reused across many queries. In doing so garbage collection becomes small and frequent, substantially reducing the need for stop-the-world collections. This is no mere implementation detail, but a fundamental building block for the ultimate goals of Neo4j in ACTiCLOUD. Without a mechanically sympathetic runtime, we cannot move to a parallel implementation lest the parallel implementation simply accelerate the production of garbage and invokes the garbage collector penalty even more frequently.

## 2. Morsel runtime.

Building on the mechanically sympathetic runtime, the Morsel<sup>5</sup> runtime treats queries as pipelines of operators which can be executed as independent units. Better performance is achieved by executing pipelines in batches and by operator fusing. Large dataflows are optimized away by fusing operators all the way down at the bytecode level, and any data that has to be transferred does so through the inexpensive slots in the mechanically sympathetic runtime.

## 3. Parallel runtime.

Historically Neo4j has run graph queries in a single thread, and parallelism has been achieved by running multiple queries concurrently. However there are use cases for graph analytics where we would like a single query to consume several cores to boost its performance.

Building on the Morsel runtime, the design goal of the parallel runtime is to enable operators in Cypher queries to be executed in parallel where possible **and** cost-effective, or serially where not (e.g. lack of available cores). The idea is to get the best performance for large graph analytical queries rather than naively get maximal (potentially fine-grained) parallelism.

In designing and implementing these features, the performance of Neo4j has improved (most notably with the parallel runtime on traditional multicore hardware). The database has become more stable and faster (since the mechanically sympathetic runtime no longer antagonises the JVM's garbage collector), and supports increasingly ambitious business workloads expressed as graph analyses to be run. The large RAM provided by ACTiCLOUD enables Neo4j to host large graphs for analysis, and the large number of cores allow increasingly large graph analytical workloads to be processed.

---

<sup>4</sup> Cypher is Neo4j's query language. It can be helpfully thought of as "SQL for graphs."

<sup>5</sup> Inspired by "Morsel-Driven Parallelism: A NUMA-Aware Query Evaluation Framework for the Many-Core Age" by Leis et al in Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data Pages 743-754, Snowbird, Utah, USA — June 22 - 27, 2014.

The mechanically sympathetic runtime enables smooth continuous operation of existing single-threaded queries and is both resource efficient owing to low memory use and performant because the database spends its time doing user work rather than garbage collection. Furthermore both the mechanically sympathetic runtime and the (in-progress) parallel runtime are both able to transparently access large numbers of RAM and CPU cores through the underlying ACTiCLOUD platform and JVM.

The Morsel and parallel runtimes together allows Neo4j to efficiently parallel process workloads. This in turn enables end users to choose to consume more resources to process a given query more quickly. It is pleasing that, as we shall show, the technology has matured and speedups are achieved on standard multicore hardware. However the results so far are less impressive on the NUMA platform of ACTiCLOUD. We believe this is because of the way Neo4j (and most databases) allocate memory as a large long-lived chunk does not map well to the fine-grained memory semantics of the NUMA platform.

The achievements for the core functionality and many operators of parallel runtime are complete to the extent that we can run LDBC benchmarks, LDBC-like benchmarks and other database workloads with the expectation that performance will be good. The Neo4j engineering team are continuing outside the scope of the ACTiCLOUD project to increase the coverage of parallel operators within the product and performance-enhance the software further for exploitation. We shall now examine the details of what has been built in detail.

### 3.1 Cypher

Recall that Neo4j pioneered a new generation of declarative query languages with Cypher in 2012. Cypher allows users to store and retrieve data from a graph database. It is a deliberately humane language built around ASCII-art representations of nodes (vertices) and relationships (edges). For example, the query:

```
MATCH (a:Person) -[:KNOWS]->(b:Person)
WHERE a.name = "Bob"
RETURN b.name
```

will query the database for the names of all persons who know someone named “Bob” and return Bob’s friends at depth 1.

To illustrate the new features of Neo4j’s parallel runtime, we will use a running example throughout this report, based on the following query:

```
MATCH (a:Person)-[:KNOWS]->(b:Person)
      -[:KNOWS]->(c:Person)

WHERE a.citizen='Iceland' AND a.surname='Li' AND
      b.citizen='China' AND b.surname='Li' AND
      c.citizen='China' AND
      c.surname='Aðalráðsson'
```

**Figure 3: Example Query.**

This query looks matches in the graph where a node (a) with label `Person` and an outgoing `KNOWS` relationship to a node (b) with a label `Person` and an outgoing `KNOWS` relationship to a node (c) which has a label `Person`. In more plain language, it is a pattern for finding immediate friends and friend-of-friends in a social network.

The `WHERE` clause in the query binds that pattern into the graph. It binds node (a) to those nodes in the graph which have a `citizen` property with value `Iceland`, and a `surname` property with value `Li`. It binds node (b) to those nodes in the graph which have a `citizen` property with value `China`, and a `surname` property with value `Li`. It binds node (c) to those nodes in the graph which have a `citizen` property with value `China`, and a `surname` property with value `Aðalráðsson`. Finally the `RETURN` clause in the query returns the `firstname` property of any nodes matched by the bound nodes a, b, and c.

While this query is simple in its intent, it can be used to showcase how the new runtimes developed for ACTiCLOUD work.

### 3.2 Lifecycle of a Cypher Query

Query evaluation is a complex process, requiring the participation of many database components, beginning at Parsing, then moving through Semantic Checking, AST Rewriting, Logical Plan Optimization, Logican Plan Rewriting, Physical Planning, Physical Compilation, and finally Execution.

The first six steps, Parsing to Logical Plan Rewriting, involves the process of taking the declarative query string and determine the most efficient way of executing the given query given the current state of the database. This is done through considering different Logical Plans - which are sets of an imperative ordered set of steps to access the data - and finding the best one.

The last three steps (Physical Planning, Physical Compilation, and Execution) concern taking a single Logical Plan and working out how to execute that plan as efficiently as possible. As part of the ACTiCLOUD project, Neo4j components involved in these three steps have received major redesigns, and the changes have been packaged together into a major new runtime component, currently named the Morsel runtime.



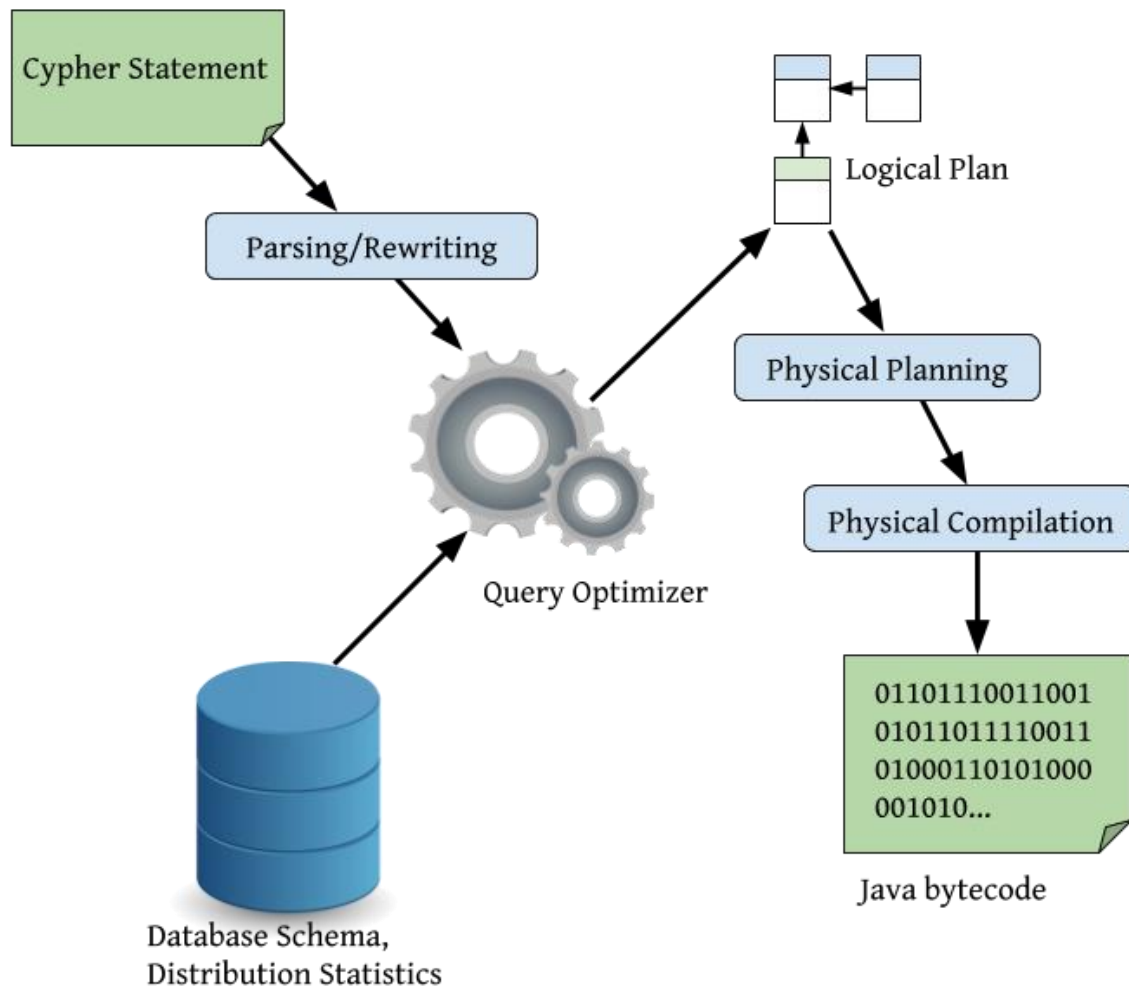


Figure 4: The life of a Cypher Query.

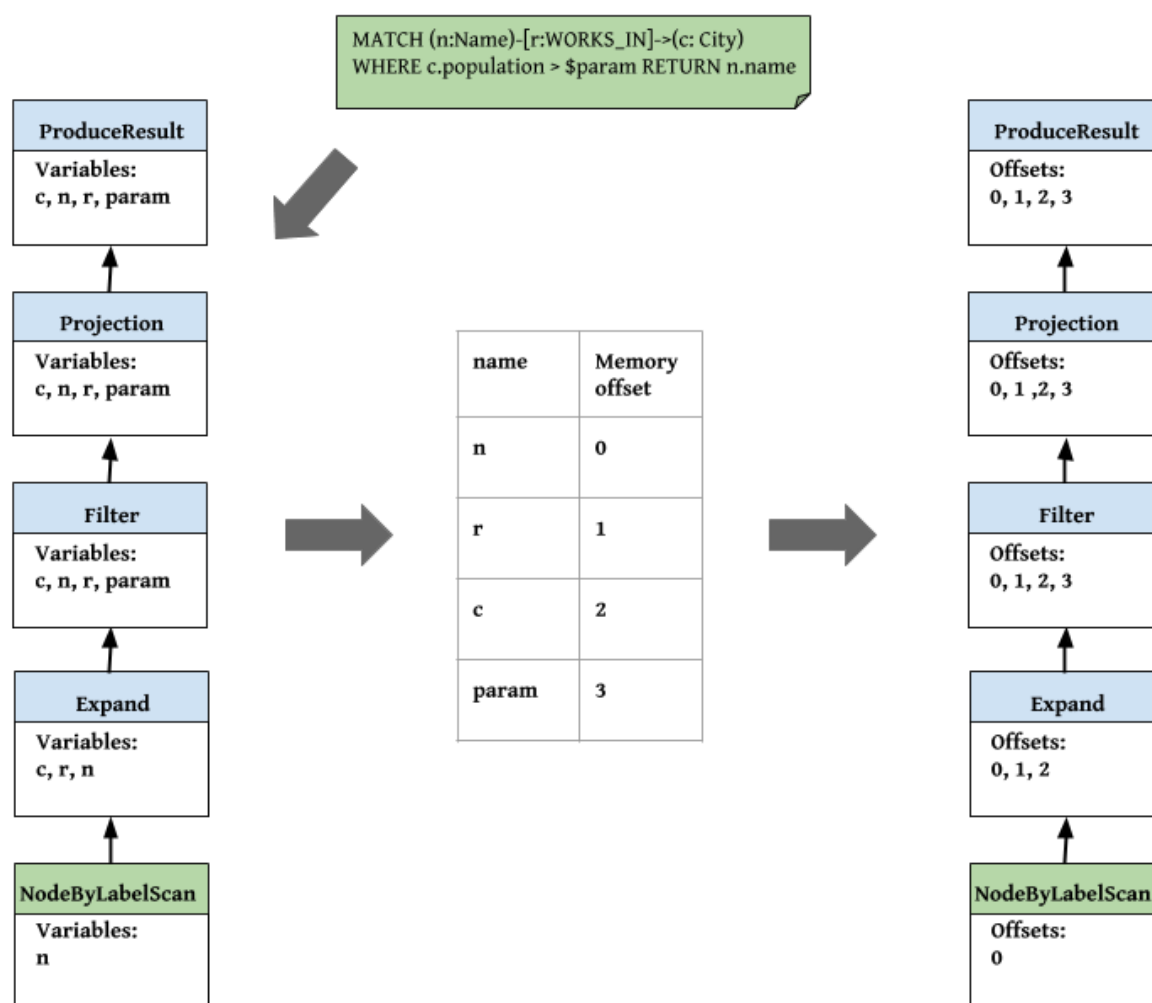
### 3.3 Physical Planning

In Neo4j Physical Planning is the step of query evaluation where two main tasks have been performed: Slot Allocation, and Pipeline Building. In physical planning we have four important items:

- Logical Plan: imperatively ordered set of operators required to execute the query.
- Physical Plan: an executable representation of the Logical Plan.
- Pipeline: a subset of a logical plan that consists of a group of operators that can be executed as an independent unit.
- Slot: an offset directly into the memory where data is stored.

As in relational database management systems, in Neo4j the process of evaluating a query

involves passing (intermediate) rows through the physical operators (Figure 4) of that query. A Cypher query contains variables and parameters which are objects that can hold a single data value (e.g., a column of a row). During the execution of a query these are accessed often and in many cases the type of the variable is completely known ahead of execution. Slot Allocation is the process of defining which variables and parameters each physical operator requires, and when possible the type of those columns. This allows Neo4j to encode rows in a more efficient format. For example a query like `MATCH (a)-[r]->(b) RETURN a, r, b` will need to allocate memory to store the two nodes `a` and `b` as well as the relationship `r`. In the Slot Allocation we make sure that we have dedicated memory slots available to provide uncontended reads and writes to those memory locations.



**Figure 5:** For a given query we find all variables and parameters of the query - here **n**, **r**, **c**, and **param** - and allocate dedicated memory slots. Whenever we read or write, we access the directly

accessed memory (array) instead of accessing the computing their location (hashmap) <sup>6</sup>.

When developing the Morsel Runtime, Physical Planning was extended to include a new task: Buffer Allocation. During execution, rows are passed between Pipelines, and depending on the Pipelines involved this passing of rows can take different forms, for example it may be lazy or require a barrier to perform eager aggregation. Buffer is the abstraction we use to describe how Pipelines communicate with one another, and Buffer Allocation is the process of defining the type of Buffers that connect Pipelines.

The output of Physical Planning is an Execution Graph Definition which encapsulates a Logical Plan as well as the outputs of all of the above tasks.

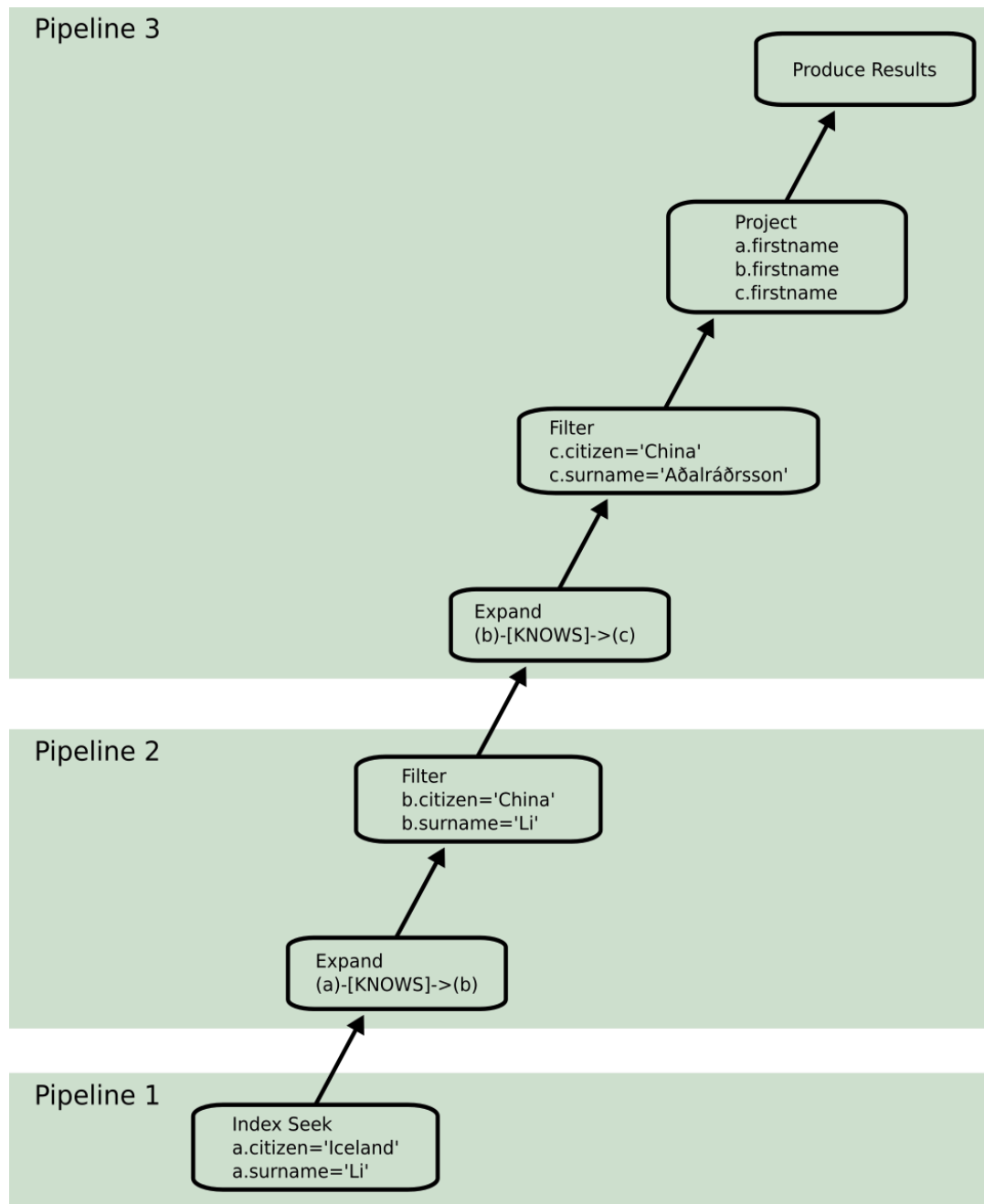
Pipeline building is the process of assigning multiple (consecutive) operators to groups (pipelines). The knowledge of which operators can be executed together as one pipeline enables Neo4j to generate physical plans that execute more efficiently. For example, it is often possible to avoid materialization of intermediate rows between operators of the same pipeline.

The logic of when to start/break a pipeline is rule-based, depending on a set of heuristics. For example, one of the signals that triggers the creation of a new pipeline is a cardinality increasing operator, such as Index Seek or Expand.

For an example pipeline building output refer to Figure 6, which illustrates the pipelines of the sample query from Figure 3.

---

<sup>6</sup> Both data structures are theoretically  $O(1)$ . But in practice the physical cost of array lookup is cheaper and that cumulative saving is multiplied by the number of times the operation is executed.



**Figure 6:** Breaking up a Logical Plan into pipelines.

### 3.4 Physical Compilation

In Physical Compilation, an Execution Graph Definition is compiled into an ExecutableQuery. This includes converting Buffer Definitions into Buffer implementations and Slot Definitions into Morsels (covered later), but by far the most complicated conversion is that of Logical Operators

into Physical Operators.

In the Morsel Runtime, Physical Compilation has received a complete redesign, introducing three major new concepts: Batch Operators, Compiled (Generated JVM Bytecode) Expressions, and Fused (Generated JVM Bytecode) Pipelines.

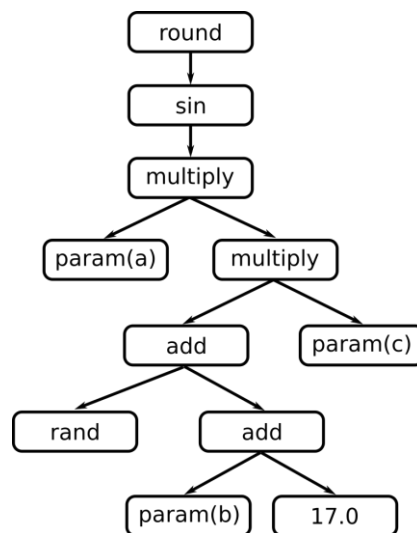
### 3.4.1 Compiled Expressions

To avoid the overheads associated with evaluating deep expression trees - which are often part of a WHERE clause and thus executed for every row - expression trees are compiled into JVM Bytecode.

For example, consider this expression:

```
round( sin( $a * ( (RAND() + ($b + 17.0) ) * $c ) ) )
```

This simple case already maps to an expression tree of depth seven, comprised of 11 expressions, which may need to be evaluated on many rows.



**Figure 7: Expression Tree.**

When not compiled, this expression maps to a tree of 11 Expression objects, and each evaluation results in 11 virtual function calls. Compilation eliminates these nested objects, mapping the expression to one instance of the following generated class.

```
Value compute( Row row, Transaction tx, MapValue params )
{
    return round(
        sin(
            multiply(
                params.get( "a" ),
```

```

        multiply(
            add(
                rand(),
                add(
                    params.get( "b" ),
                    doubleValue( 17.0 )
                )
            ),
            params.get( "c" )
        )
    )
}

```

### 3.4.2 Fused Pipelines

When a Pipeline contains more than one operator (the common case) those operators can all be compiled (as JVM Bytecode) into one nested loop. The most obvious advantage of this is it avoids materializing intermediate rows between operators, but when combined with Batched execution there are other significant benefits.

Currently the Morsel runtime has partial support for Pipeline Fusing, as it can not yet generate code for all operators. However, there is support for partially fusing pipelines: fusing first all operators until we encounter one that can not be fused.

See below for pseudocode and diagram in Figures 7 and 8 showing fused and non-fused variants of Pipeline 3 from the example query.

### 3.4.3 Fused Example

When we fuse a pipeline we turn all the operators of the pipeline into a single loop all working from the same input morsel. For example the **Filter** operator is now turned into a simple `if` statement. Variables are stored in local variables which means that the data can be stored in CPU registers.

```

// --- Fused: Expand, Filter, Project ---
while (inputMorsel.hasNextRow()) {
    long a = inputMorsel.getLongAt( <offset> )
    long b = inputMorsel.getLongAt( <offset> )
    RelationshipCursor bRelationship = cursorPool.allocate(b)
    // --- Expand ---
    while (bRelationship.next()) {
        long c = bRelationships.otherNode()
        // --- Filter ---
        Value cCitizen = dbAccess.nodeProperty( c, 'citizen', nodeCursor,
                                                propertyCursor )
    }
}

```

```

        Value cSurname = dbAccess.nodeProperty( c, 'surname', nodeCursor,
                                                propertyCursor )
    if (cCitizen.equals('China') && cSurname.equals('Aðalráðsson')) {
        // --- Project ---
        Value aName = dbAccess.nodeProperty( a, 'name', nodeCursor,
                                                propertyCursor )
        Value bName = dbAccess.nodeProperty( b, 'name', nodeCursor,
                                                propertyCursor )
        Value cName = dbAccess.nodeProperty( c, 'name', nodeCursor,
                                                propertyCursor )

        // return result columns
    }
}
inputMorsel.moveToNextRow()
}

```

### 3.4.4 Non-fused (operator-at-a-time) Example

In the non-fused case each operator will operate individually one morsel-at-a-time. In this example the **Expand** operator will write data to the output morsel until the morsel is full or there is no more data. The output from **Expand** is passed along as the input of **Filter** which in turn will emit only the rows satisfying the predicate and pass along those rows to the **Project** operator.

```

// --- Expand ---
while (inputMorsel.hasNextRow()) {
    long b = inputMorsel.getLongAt( <offset> )
    RelationshipCursor bRelationship = cursorPool.allocate(b)
    while (bRelationship.next() && outputMorsel.hasNextRow()) {
        long c = bRelationships.otherNode()
        outputMorsel.setLongAt( <offset>, c )
        outputMorsel.moveToNextRow()
    }
    inputMorsel.moveToNextRow()
}

outputMorsel.resetToFirstRow()

// --- Filter ---
Morsel writingMorsel = outputRow.shallowCopy()
while (outputMorsel.hasNextRow()) {
    long c = outputMorsel.getLongAt( <offset> )
    Value cCitizen = dbAccess.nodeProperty( c, 'citizen', nodeCursor,
                                                propertyCursor )
    Value cSurname = dbAccess.nodeProperty( c, 'surname', nodeCursor,
                                                propertyCursor )
}

```

```

        if (cCitizen.equals('China') && cSurname.equals('Aðalráðsson')) {
            writingMorsel.copyFrom(outputRow)
            writingMorsel.moveToNextRow()
        }
        outputMorsel.moveToNextRow()
    }

    outputMorsel.resetToFirstRow()

// --- Project ---
while (outputMorsel.hasNextRow()) {
    Value aName = dbAccess.nodeProperty( a, 'name', nodeCursor,
                                           propertyCursor )
    Value bName = dbAccess.nodeProperty( b, 'name', nodeCursor,
                                           propertyCursor )
    Value cName = dbAccess.nodeProperty( c, 'name', nodeCursor,
                                           propertyCursor )
    outputMorsel.setRefAt( <offset>, aName )
    outputMorsel.setRefAt( <offset>, bName )
    outputMorsel.setRefAt( <offset>, cName )
}

outputMorsel.resetToFirstRow()
// return result columns

```

### 3.4.5 Pull (row-at-a-time) Example

In the traditional iterator or pull-based model data flows in the opposite direction. We start by calling next on **Project** which will call next on **Filter** which in turn will call next on **Expand**. In this way data will flow from the leaf operator one row-at-a-time.

```

// --- Result Results ---
while (project.hasNextRow()) {
    Row row = project.nextRow()
    // return result columns
}

// --- Project ---
Row row = filter.nextRow()
Value aName = dbAccess.nodeProperty( a, 'name', nodeCursor, propertyCursor)
Value bName = dbAccess.nodeProperty( b, 'name', nodeCursor, propertyCursor)
Value cName = dbAccess.nodeProperty( c, 'name', nodeCursor, propertyCursor)
row.setRefAt( <offset>, aName )

```



```

row.setRefAt( <offset>, bName )
row.setRefAt( <offset>, cName )
return row

// --- Filter ---
Row row = expand.nextRow()
long c = row.getLongAt( <offset> )
Value cCitizen = dbAccess.nodeProperty( c, 'citizen', nodeCursor,
                                         propertyCursor )
Value cSurname = dbAccess.nodeProperty( c, 'surname', nodeCursor,
                                         propertyCursor )
if (cCitizen.equals('China') && cSurname.equals('Aðalráðsson')) {
    return row
}

// --- Expand ---
RelationshipCursor bRelationship = // initialized
Row row = // initialized
if (bRelationship.next()) {
    row.setLongAt( <offset>, c )
    return row
} else {
    while (input.hasNextRow()) {
        row = input.nextRow()
        long b = inputRow.getLongAt( <offset> )
        bRelationship = cursorPool.allocate(b)
        if (bRelationship.next()) {
            row.setLongAt( <offset>, c )
            return row
        }
    }
}
}

```

### 3.5 Execution

Once planning is complete we end up with an optimal - or at least close to optimal - Logical Plan. The task of finding the most efficient way of executing that Logical Plan though comes with its own architectural decisions and trade-offs. Historically Neo4j has been optimized for single-threaded OLTP workloads, meaning that it allows multiple queries to run concurrently to achieve a large amount of work in aggregate. However, when it comes to OLAP-type of workloads running on a system with an abundance of CPU cores, forcing all queries to run on a single execution thread is a disadvantage.

The goal of this new execution model for the ACTiCLOUD project is to achieve good scaling behaviour on systems with many CPUs while still maintaining the ability for fast execution for

smaller, OLTP types of queries. To this end we have settled on the following architectural decisions which will all be discussed in detail later

- A push-based or bottom-up model where data is pushed from the leaf operators to their parents.
- Batched execution, queries are not executed one row-at-a-time but instead in bigger batches.
- Logical plans are subdivided into pipelines which make up the work units or tasks of the system.
- Pipelines can be fused into tight loops when necessary and compiled into specialized Java bytecode for the query.

### 3.5.1 Push vs Pull

The traditional execution model of most databases is the iterator or “Volcano” model<sup>7</sup>. In this pull-based approach each physical operator calls a next function on its child operators and in this way pulling data from the leaf-operators one row-at-a-time. This is a convenient and flexible approach, but it has some significant drawbacks: The next function will be called for each operator and row leading to many virtual function calls. Furthermore, by its nature the iterator model often leads to poor data locality and hence worse utilization of CPU caches and worse branch prediction performance.

In a push-based model data is instead pushed from the leaf operators to its parent operators. In this way, data will be kept in local variables which makes it possible to directly utilize CPU registers as well as avoiding virtual calls and improving utilization of CPU caches. As illustrated in the “Fused Example” above.

### 3.5.2 Batched Execution

Most database systems execute queries one row-at-a-time. However, this can result in unnecessary interpretation overhead (in the form of virtual method calls), branching, and object allocation. To overcome that, data can be eagerly loaded into memory and then have each physical operator work on the data in tight loop and then pass data to its parent operators. Although this would get rid of some of the interpretation overhead, it would lead to a big memory overhead and for some operators a lot of extra work.

In the Morsel Runtime we take a best-of-both-worlds approach and let the physical operators consume and produce batches (Morsels) of 100-1000 rows before passing data to its parent operators.

### 3.5.3 Operator Fusing

Batching helps a great deal with the interpretation overhead, however, for some operators it introduces extra work. For example, in Pipeline 3 of the example query we have pipeline

---

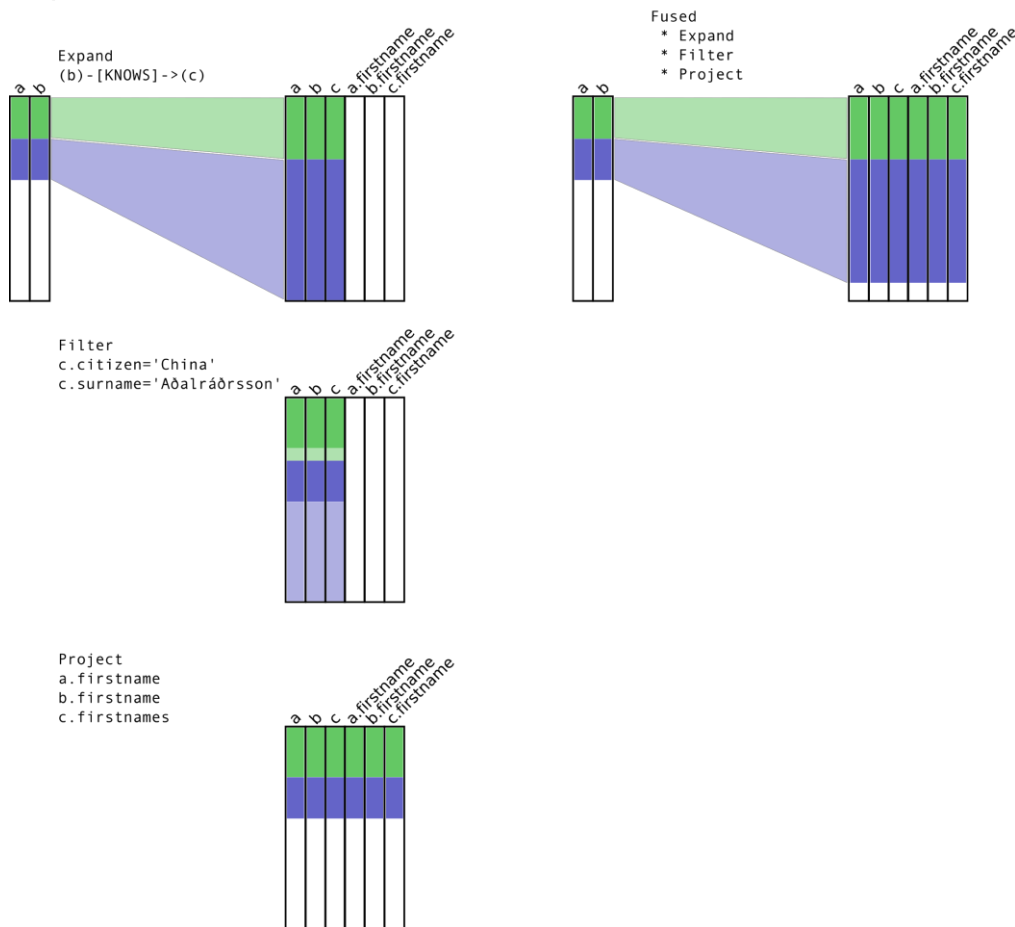
<sup>7</sup> See [https://dbms-arch.fandom.com/wiki/Volcano\\_Model](https://dbms-arch.fandom.com/wiki/Volcano_Model) for an explanation.

consisting of `Expand`, `Filter`, and `Project`. `Expand` will produce one chunk of data and pass it to `Filter` which will need to go through the entire chunk and removing rows that doesn't satisfy the filter predicate before passing the data to `Project` (see left panel in image below).

A more efficient approach is to fuse the entire pipeline into one tight loop as mentioned earlier. In the Morsel Runtime we fuse operators in Pipelines when it is more efficient and emit specialized Java Byte Code for the query at hand. In this way, we not only remove some of the extra work introduced by having to execute one operator-at-a-time, but it also allows us to use local variables for data and temporary data structures in local variables leading to better cache and register utilization and lower memory usage.

For some operators, e.g. `Filter`, fusing doesn't only lead to more efficient execution but also leads to better utilization of the morsel. In the non-fused case, the `Filter` operator must compact the input morsel before passing it along to the next operator. However, when we fuse all the operators in a pipeline we may completely fill the morsel before passing it to the next pipeline, see `Filter`, see Figure 7 below and pseudo code above.

## Pipeline 3



**Figure 8:** Executing Pipeline 3 in normal mode (left) and fused (right).

### 3.5.4 Terminology

- **Pipeline:** a sequence of operators to execute together, in the same work unit (a task), a single pipeline acting on a single morsel.
- **Buffer:** every pipeline has an input buffer. The buffer contains morsels and/or tasks for that pipeline. Buffers are bounded in size to provide back pressure during execution.
- **Execution State:** thread-safe data structure containing all data for a running query
- **Worker:** thread that executes work units to evaluate incoming queries
- **Morsel:** unit of data that a task operates on, typically a batch of 100-1000 rows
- **Task:** the unit of work. Executes one pipeline on one input morsel, and produces one output morsel. If back-pressure prevents a task from completing, it can be rescheduled as a Continuation to resume at a later time.
- **Continuation:** a task that did not finish execution and must be scheduled again
- **Scheduler:** responsible for deciding which unit of work to process next. Scheduling is decentralized, each worker has its own scheduler instance.

### 3.5.5 Parallel Execution

To enable parallel query evaluation many database management systems partition the data (e.g., tables), then pin threads to CPU cores (and therefore to partitions, too). Each thread then accesses only the data from its own partition, and the threads coordinate to evaluate incoming queries. The main advantage of this approach is that it allows each thread to proceed mostly-uncontended, reading CPU-local data.

Though this approach is appealing at first, its limitations quickly become evident when applied to the evaluation of queries over graph data; graphs are practically impossible to partition in a way that strikes a reasonable balance between (a) limiting inter-thread communication and (b) balancing the load across all threads. For this reason, in designing Neo4j's parallel query engine we took a different approach, borrowing ideas from both the database field as well distributed stream processing systems.

### 3.5.6 Design

At the center of a query execution is the execution state, a thread-safe data structure that encapsulates a DAG of buffers (vertices) and pipelines (edges). Each pipeline describes a unit of computation (concretely, a sequence of executable operators), buffers store the data to compute, and data moves through the DAG from leaves to root (user).

For each executing query there is one execution state, which exists for the duration of that query's execution. Worker threads (workers) concurrently access the execution state, writing/reading data to/from buffers. Workers are not bound to any execution state, rather they perform work on any of the currently executing queries. This strategy effectively balances load across all threads (cores).

To limit contention, workers operate on batches of rows. Batches are named morsels. More specifically, workers execute tasks, which are defined as one pipeline operating on one input morsel; one task is one work unit.

All work performed within a task is uncontended, contention can only occur during reading/writing of the execution state, which is kept to a minimum. Typically, the data written to buffers is in the form of morsels. This occurs when a worker takes the output morsel produced by a task and writes it to the output buffer of that task's pipeline.

However, there are two other cases:

1. If back-pressure (e.g., full output morsel) prevents a task from processing all of its input morsel that task is written to a buffer, to be continued (continuation) at a later time.
2. Certain operators produce output as specialized data structures (e.g., aggregations produce maps), which are written to aggregating buffers (accumulators).

The decision of which work unit to process next is delegated to a scheduler.

Schedulers are responsible for two tasks, selecting which query to work on next, and then retrieving the next work unit from a pipeline/buffer of that query. They search (traverse) the

execution states of queries to find the next pending work unit.

As the policies for making these decisions can greatly impact overall system behavior, we have made them pluggable, making it possible to experiment with many different policy implementations.

One could imagine policies that optimize any number of things: limit memory usage (buffer as little as possible), return results to user as soon as possible (schedule later buffers first), limit contention between workers (every worker retrieves work from a different query/pipeline), overall throughput (all queries have equal priority), latency of individual queries (FIFO query priority), etc.

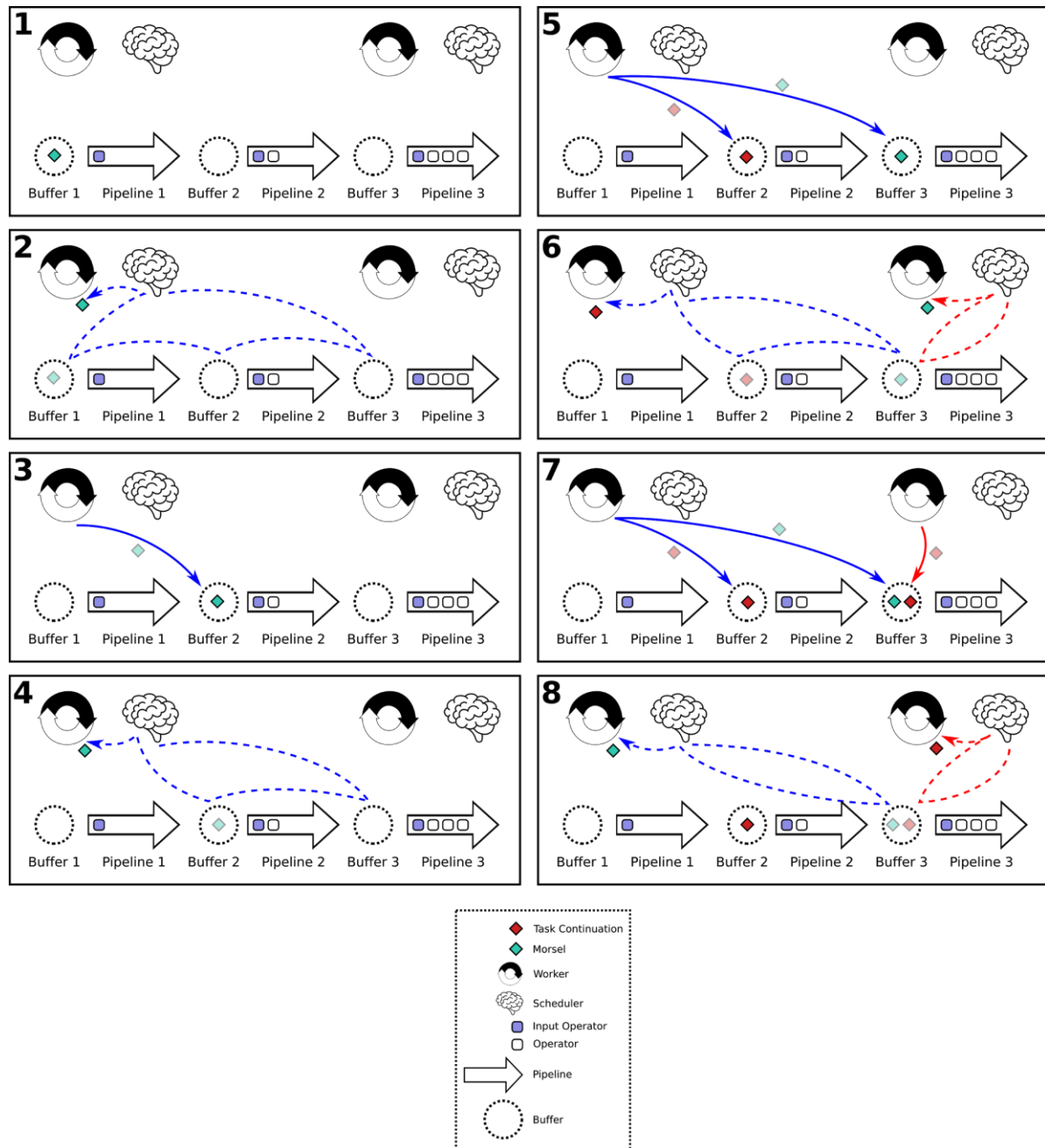
By default, query selection policy is FIFO, and pipeline selection is "return results to user as soon as possible" (starting from the last pipeline and traversing towards the leaf nodes). To avoid scheduling becoming a bottleneck, every worker has its own scheduler instance, i.e., there is no central scheduling component. A worker will delegate to its scheduler the task of retrieving the next task. It then operates on that task and, finally, writes its output to the appropriate buffer.

To better explain the process, consider the example in Figure 8, which uses our same running example.

In the interest of clarity, this example has only two workers, and a single query.

1. When a query is received, its execution state is initialized. This includes placing a morsel into its first buffer (e.g., input of an Index Seek pipeline), to bootstrap execution.
2. Worker 1, via its scheduler, retrieves the first morsel (task) from Buffer 1. Note, either worker could have acquired this work unit, on a first-come, first-served basis.
3. Worker 1 executes the operators (Index Seek) of Pipeline 1 on the morsel it acquired, producing a new output morsel. It then writes that output morsel to Buffer 2.
4. Worker 1 retrieves the next task, this time from Buffer 2.
5. Worker 1 executes the operators (Expand and Filter) of Pipeline 2 on the morsel it acquired. Because Expand is a cardinality increasing operator, processing one input row can result in the writing of many rows to the output morsel. Consequently, Worker 1 produces (fills) a new output morsel but does finish operating on all of its input. Its unfinished task is written back to Buffer 2, to be rescheduled later as a continuation, and its output morsel is written to Buffer 3.
6. There are now two pending work units. Worker 1 retrieves the continuation from Buffer 2, while Worker 2 retrieves the morsel from Buffer 3.
7. Again, Worker 1 produces one continuation and one new morsel. Similarly, due to back pressure Worker 2 is not able to finish the processing of its input morsel, and so writes a continuation to Buffer 3. However, each worker has a different source of back pressure. In the case of Worker 1, it was a full output morsel. Whereas for Worker 2, the source of back pressure was user demand (see Reactive Results section).
8. There are now three pending work units. The workers continue in this fashion, reading/writing work from/to the execution state, until the last work unit has been

processed and all rows returned to the user. At that point all resources claimed by the execution state will be freed.



**Figure 9: Example execution.**

### 3.5.7 Reactive API / Back Pressure

For the ACTiCLOUD project we have also developed a new API for handling reactive results and backpressure in the database. This is required for handling streaming large data volumes back to the client without exhausting memory and without blocking any worker threads. This is a necessary precaution when scaling to large workloads which ACTiCLOUD in principle enables.

The approach we have taken is inspired by the reactive manifesto but with some additional tweaks. There are two interfaces involved in the API, a `QuerySubscriber` and a `QuerySubscription`.

```
public interface QuerySubscriber
{
    void onResult( int numberOfFields );
    void onRecord() throws Exception;
    void onField( AnyValue value );
    void onRecordCompleted();
    void onError( Throwable throwable );
    void onResultCompleted( QueryStatistics statistics );
}

public interface QuerySubscription
{
    void request( long numberOfRecords );
    void cancel();
    boolean await();
}
```

The client implements a `QuerySubscriber` and passes it along when executing the query and a `QuerySubscription` is handed back to the client.

```
MySubscriber subscriber = new MySubscriber()
QuerySubscription subscription = db.execute(query, subscriber);
```

The client is then able to control demand by calling `request` on the subscription and the database will stream the data back to the subscriber as it becomes available.



## 4 Interim Evaluation

We are presently working on the evaluation of the new Neo4j runtime components atop the ACTiCLOUD stack (specifically on a Numascale computer). In the meantime we are able to present interim results from our CI build benchmarks running on x86-64. We make no claims about whether these benchmarks are indicative of Neo4j's final performance on Numascale, but we are pleased with the performance improvements versus previous Neo4j runtimes on the same x86-64 platform.

In the evaluation of the new runtime components we previously used the LDBC - SNB - Interactive benchmark. Our profiling of the LDBC benchmark shows that it missed code paths for much of the new code in the ACTiCLOUD parts of Neo4j.

Accordingly, while LDBC remains the gold standard for widespread graph database benchmarks, we have taken inspiration from it and adapted part of it to help drive out performance for the new runtimes. Our benchmarks are therefore not directly comparable to other LDBC benchmarks, we believe this was an appropriate choice. Our minor deviations are compatible with the typical users in our large commercial and community user base. Furthermore having a benchmark that exercises (as opposed to avoiding) the new code in Neo4j has enabled us to test and tune that code.

The benchmarks are run on both on conventional hardware and the Athens NumaScale cluster. The conventional hardware is from AWS and the specification is: EC2: m5d.2xlarge, CPU: 8, RAM: 32 GB, STORAGE: 1x300 GB NVMe SSD.

The performance benchmarks are in Figure 10, and detailed descriptions of each query, the query plan, and our learning are in the following subsections.

## Evaluation

### Standard Hardware

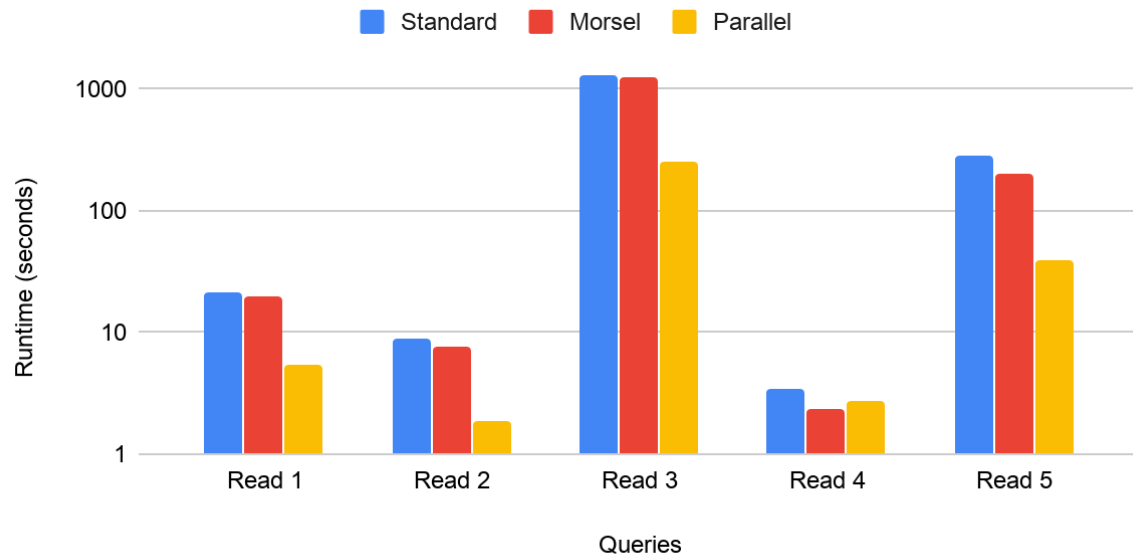


Figure 10: Runtime Performance.

### 4.1 Query 1

The following query computes degree distribution for a graph.

```
MATCH (n)
WITH length((n)--()) AS degree
RETURN percentileDisc(degree, 0.25) AS perc25,
        percentileDisc(degree, 0.50) AS perc50,
        percentileDisc(degree, 0.90) AS perc90,
        percentileDisc(degree, 0.99) AS perc99
```

The plan produced is as follows:

**AllNodeScan** -> **Project** -> **Aggregation** -> ProduceResults

The items in bold are able to be fused into a triple-nested while loop, enabling stack-based variables to be (almost) exclusively used.

The performance increases for parallel operation atop the morsel runtime are because the AllNodeScan operator is parallelized. Since it scans all nodes in the graph there is a great deal of work available to be processed concurrently. However our implementation does not yet

support parallel execution with fusing, and so there is more performance to be gained when that implementation limitation is removed.

As a consequence of this discovery, the parallel `AllNodeScan` operator shows the benefit of parallelizing leaf nodes which will set the direction for future exploitation work.

We discovered via profiling that the largest cost in the query is computing the degree for each node. We were aware that we could have made the degree cost lower by switching on the dense-node feature in Neo4j. We chose not to, however, so that the same LDBC datasets could be used across all queries over the history of the benchmark.

## 4.2 Query 2

The following query finds people where for each person it counts the number of people they know who speak at least two languages that they themselves speak.

```
MATCH (p1:Person)-[:KNOWS]-(p2:Person)
WITH
  person,
  friend,
  size([l IN p1.languages WHERE l IN p2.languages]) AS commonLanguages
WHERE commonLanguages >= 2
RETURN commonLanguages, count(p1)
```

The plan generated is as follows:

```
LabelScan (person) -> Expand (friends)
  -> Filter (about friend, very selective, that have more than 1
    shared language)
  -> Project (about friend, numberOfSharedLanguages)
  -> Aggregation numberOfSharedLanguages, count(person)
  -> ProduceResult
```

We discovered via profiling that the largest cost in the query is retrieving language properties from the Person nodes in the graph. Retrieving that string array from disk dominates<sup>8</sup> the cost of computing the query, however because it is parallelizable, it means the runtime can overlap IO and computation leading to good speedups.

## 4.3 Query 3

The following query is a much more difficult, larger query than the previous. Instead of checking pairs of friends for linguistic compatibility, it checks friends-of-friends which involves many more graph traversals.

---

<sup>8</sup> For details on Neo4j's storage subsystem, see Graph Databases, 2nd Edition, O'Reilly Media, Inc., June 2015. ISBN: 9781491930885

```
MATCH (p1:Person)-[:KNOWS]-(p2:Person)-[:KNOWS]-(p3:Person)
WHERE p1<>p2
WITH
    person,
    friend,
    size([l IN p1.languages WHERE l IN p2.languages]) AS commonLanguages
WHERE commonLanguages > 1
RETURN commonLanguages, count(person)
```

The plan for the query is as follows:

```
LabelScan (person) -> Expand (friend) -> Expand (friend of friend)
-> Filter (about friend, very selective, that have more than X
    shared languages)
-> Project (about friend, numberOfSharedLanguages)
-> Aggregation numberOfSharedLanguages, count(person)
-> ProduceResult
```

As per Query 2, we discovered via profiling that the largest cost in the query is retrieving language properties from the Person nodes in the graph. Retrieving that string array from disk dominates the cost of computing the query, however because it is parallelizable, it means the runtime can overlap IO and computation leading to good speedups.

We do not hit the asymptotic IO limits during parallel IO because of the characteristics of the property store design in Neo4j. We assume that given a large enough number of concurrent reads for properties (or mixed workloads involving the node and relationship stores) that eventually adding parallel work would not result in further speedups.

#### 4.4 Query 4

This query finds all messages sent between people in the system and groups them by when they were sent. It is not a very graph-centric query, but does derive from LDBC in order to benchmark aggregation performance.

```

MATCH (m:Message)
RETURN
    msToWeeks(timestamp() - m.creationDate) AS weeksAgo,
    count(*) AS messagesCreated

```

The plan for the query is as follows:

```
NodeIndexScan -> EagerAggregation -> ProduceResults
```

This is an index-heavy query. As part of the ACTiCLOUD work, we have improved our planner to take advantage of data stored in Neo4j indexes. Specifically in this case, in the planning phase we can identify if the property getting scanned will be used again. When this is the case, we avoid retrieving the property from the property store and re-use the value from the index, halving the amount of IO needed to retrieve an indexed property value.

We see advantages to the new runtime on a single thread because of pipeline fusing: the executed queries are mechanically sympathetic. The more modest performance improvement in the parallel version are because index scans are not yet parallel, so we pay the cost of thread safety. However even the sub-optimal parallel version beats the old serial version.

## 4.5 Query 5

This query finds the correlation between age and size of social network.

```

MATCH (person:Person)-[:KNOWS]-()-[:KNOWS]-(fof:Person)
RETURN
    person.age AS age,
    count(fof) AS popularityOfAge

```

The plan for the query is as follows:

```

NodeIndexScan -> Expand (friend) -> Expand (fof) -> Filter
-> EagerAggregation -> ProduceResults

```

Like Query 3, we have a great deal of work available to parallelize because of the high cardinality of friends-of-friends. Therefore the runtime has a large number of work units to schedule, keeping all worker threads busy. However unlike Query 3, this query is not bottlenecked on IO since the age property value is retrieved directly from the Person index, potentially halving the amount of IO required. The 7.2x speedup is close to the theoretical maximum (an 8 core server), and it is not an embarrassingly parallel problem like a scan. This is very pleasing.

## 5 Software Assets

The software developed for the base architecture for these experiments is available as open source or precompiled binary.

- Source: <https://github.com/neo4j/neo4j/tree/3.4.0>
- Binary: <http://dist.neo4j.org/neo4j-enterprise-3.4.0-unix.tar.gz>

The software developed to exploit the performance of the new runtimes is available in Neo4j 4.0 (open core). This software includes greater support for parallel operators and more extensive optimizations. Please note it is alpha release quality at the time of writing.

- Enterprise edition: <https://neo4j.com/download-thanks-beta/?edition=enterprise&release=4.0.0-alpha09mr02&flavour=unix>

## 6 Next Steps

On completion of the ACTiCLOUD project, our next steps in research and development will be to implement the remainder of parallel operators. Furthermore, we aim to commercialize the research in coming years, making it a standard part of the Neo4j graph database. We anticipate good results and correspondingly we anticipate that the attractiveness of graphs for business intelligence workloads will also grow.