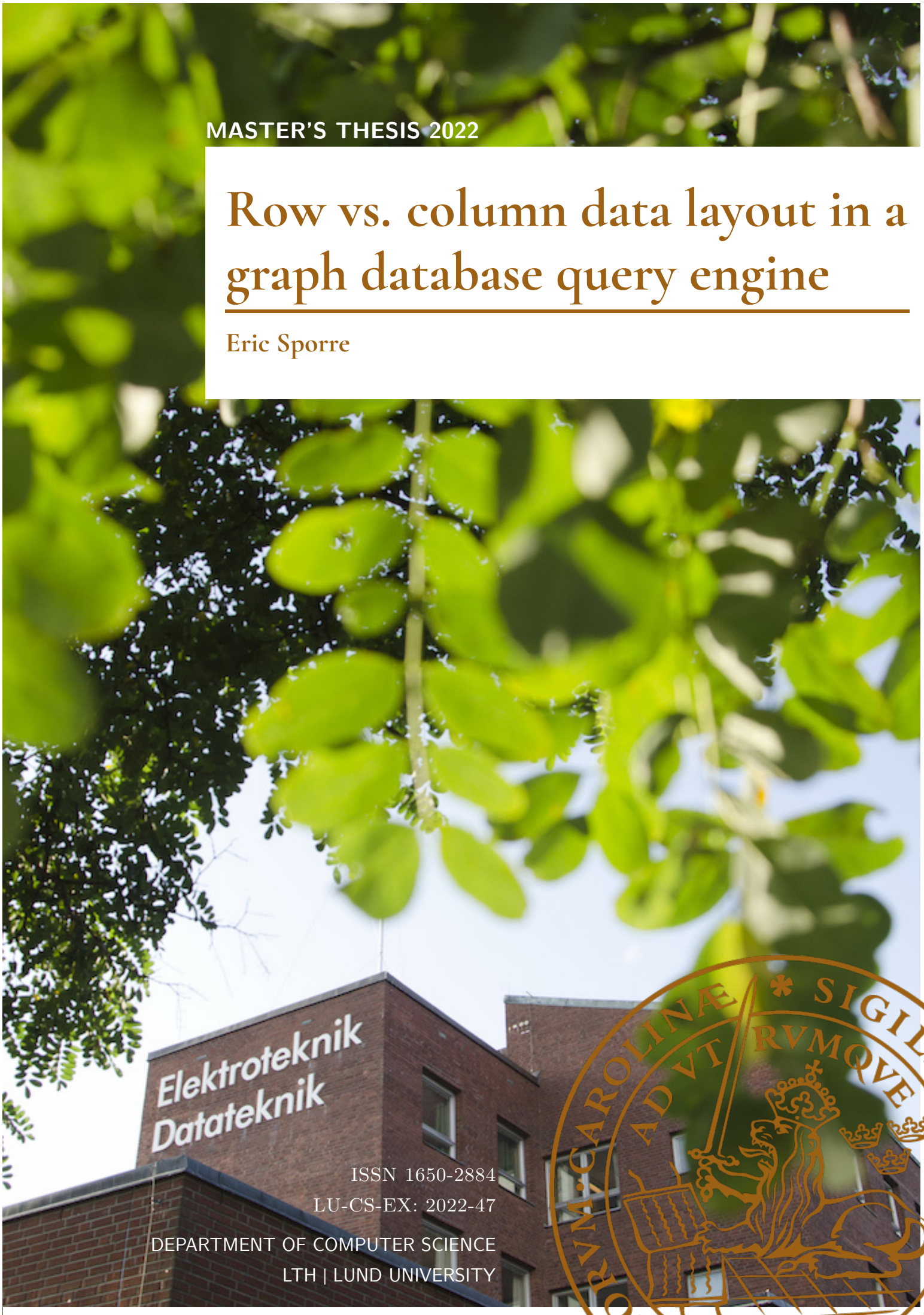


MASTER'S THESIS 2022

Row vs. column data layout in a graph database query engine

Eric Sporre

ISSN 1650-2884
LU-CS-EX: 2022-47

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2022-47

Row vs. column data layout in a graph database query engine

Rad- eller kolonnordning vid exekvering i en grafdatabas

Eric Sporre

Row vs. column data layout in a graph database query engine

Eric Sporre
tna13esp@student.lu.se

July 6, 2022

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisors: Henrik Nyman, henrik.nyman@neo4j.com
Luigi Nardi, luigi.nardi@cs.lth.se

Examiner: Jacek Malec, jacek.malec@cs.lth.se

Abstract

This thesis aims to examine if there is any performance improvement to be gained by changing the memory layout from row-wise to column-wise inside of the Neo4j query engine. In order to test this a column-wise representation was created along with new implementation for a few operators to better leverage the potential of the new memory layout, such as using SIMD. This change means that the query execution strategy is changed from the current approach, which relies upon fusing and compilation, to a vectorized approach instead.

The conclusions drawn were that a performance improvement was achievable by combining the new column-wise layout in combination with vectorized solutions. These solutions are limited however, since they can only be used for value types and might not be suitable for all operators. The memory layout change or the use of the new vectorized implementations are not enough on their own to yield an improvement; only in combination do they improve upon the state-of-the-art compilation strategy currently in use.

Keywords: Neo4j, graph database, query engine, query execution, memory layout

Contents

1 Introduction **5**
- 1.1 Goals . . . 5
- 1.2 Related work . . . 6
  - 1.2.1 Morsel-Driven Parallelism . . . 6
  - 1.2.2 Vectorization vs. Compilation in Query Execution . . . 6
  - 1.2.3 DSM vs. NSM: CPU Performance Tradeoffs in Block-Oriented Query Processing . . . 7
- 1.3 Contribution . . . 7

2 Background **9**
- 2.1 Relational databases . . . 9
- 2.2 Graph databases . . . 10
- 2.3 The Cypher query language . . . 10
- 2.4 ACID . . . 10
- 2.5 CPU infrastructure . . . 11
- 2.6 JMH . . . 12
- 2.7 Java incubator module and Vector API . . . 12

3 Query Engine Architecture **15**
- 3.1 Parsing . . . 15
- 3.2 Semantic analysis . . . 17
- 3.3 Logical planning . . . 17
- 3.4 Execution . . . 18
  - 3.4.1 Pipelined runtime . . . 18
  - 3.4.2 Fusing operators and compilation . . . 18
  - 3.4.3 Producing results . . . 18

4 Implementation **21**
- 4.1 Implementing column-wise memory layout . . . 21
- 4.2 Projections and the project operator . . . 22

4.3 Filter operator . . . 22
4.4 Aggregation operator . . . 22
4.5 Column-wise copying between morsels . . . 23
4.6 Benchmark design . . . 23

5 Evaluation **25**
5.1 Experimental setup . . . 25
5.2 Results . . . 25
5.2.1 Projection operator . . . 25
5.2.2 Aggregation operator . . . 26
5.2.3 Column-wise copying between morsels . . . 28

6 Discussion **29**
6.1 Operator performance . . . 29
6.1.1 Project operator . . . 29
6.1.2 Aggregation operator . . . 30
6.1.3 Column-wise copying . . . 30
6.2 The filter operator . . . 31
6.3 Potential improvement . . . 31
6.4 Limitations . . . 31

7 Conclusions and future work **33**
7.1 Conclusions . . . 33
7.2 Future work . . . 33
7.2.1 On-the-fly transition . . . 33
7.2.2 Combining compilation and vectorization . . . 34

References **35**

Chapter 1
Introduction

Neo4j is a native graph database with a focus on being able to handle highly connected data. It is ACID compliant and uses a custom created query language called Cypher. Execution speed of queries is always of interest when comparing different databases so improving and researching possible advancements is desirable. Today the state-of-the-art Neo4j query engine utilizes morsel-driven parallelism in combination with a compilation strategy with a constant row-wise memory layout during execution. We introduce our goals and scope in section 1.1, present related work in the area in section 1.2 and finally describe the contribution of this work in section 1.3.

1.1 Goals

The main goal of the project is to examine if there is any difference between using row or column data layout for intermediate results inside the query engine. More specifically the implication on CPU efficiency caused by the different data layouts, and under what conditions and for which query execution operators one layout may be preferable to the other. In order to fully realize the potential of a new memory layout optimization, such as using SIMD, will be introduced.

We limit the scope by focusing on a few operators which we believe might most benefit from a different memory layout. Furthermore the scope is limited to operators that do not increase cardinality and the new operator implementations are also limited in the sense that they

- may not be fused with other operators to create larger pipelines
- can not utilize a compilation strategy
- materialize their intermediate results after each operator

1.2 Related work

Researching new ways to improve the performance of databases and query engines is not a new concept. In this section three different ways of enhancement from related work are presented.

Section 1.2.1 describes a model to achieve efficient parallelization in query engines, section 1.2.2 focuses on different approaches of the actual query execution and finally 1.2.3 instead concerns different memory layouts during execution.

1.2.1 Morsel-Driven Parallelism

Morsel-driven parallelism was presented in Leis et al. [8] as a modern solution to better take advantage of the increase of number of cores. The presented solution divides work into smaller batches, called morsels, hence the name, which can be dynamically distributed between threads and executed in arbitrary order, meaning morsels are independent of one another.

A major difference compared to the earlier more common Volcano approach [4], which utilizes an open(), next(), close() API for each individual operator, the morsel-driven approach instead executes as far as possible inside an operator pipeline. A pipeline might need to come to an end in case synchronization is required, as is the case for sorting or aggregation since all morsels need to complete up until that point. Another reason for pipeline breakage are operators that affect morsel sizes by adding or removing tuples. To maintain evenly sized morsels, repartitioning into new morsels is executed between pipelines.

Since morsels can be dynamically allocated between threads the number of threads currently occupied with a single query can easily be adapted to account for the workload in case more queries are submitted.

Furthermore, Leis et al. also claim that the size of the morsel is not critical to execution time as long as the size is large enough to overcome the scheduling overhead. Their conclusion is that the size should be set to the smallest size which breaches the overhead.

1.2.2 Vectorization vs. Compilation in Query Execution

Sompolski et al. [16] sought to compare the performance of solutions using either vectorization or compilation for query execution. Earlier work had shown them that both of these approaches improve performance. The vectorized approach utilizes blocks of multiple tuples for each next() method instead of just a single tuple. This block-based approach reduces interpretation logic overhead along with blocks themselves allowing for optimization to be carried out by the compiler. On the other hand, the compilation approach tries to solve a query by compiling it in its entirety of parts of it which has been shown to have many benefits and outperform an interpreted approach.

The findings Sompolski et al. arrive at are that vectorization and compilation both have advantages and disadvantages and that the better option of them is therefore dependent on the use case. They also emphasize that the true best option is a combination of the strategy

and that one does not need to pick one over the other. One suggested approach is to compile smaller parts of a query, materialize the results and use vectorization in between.

1.2.3 DSM vs. NSM: CPU Performance Tradeoffs in Block-Oriented Query Processing

The Decomposition Storage Model (DSM) – column-wise – and the N-ary Storage Model – row-wise – tend to adhere to discussion surrounding persistent storage. Zukowski et al. [17] instead sought to compare these models during query execution. In order to be able to compare these memory layouts they use "block-oriented" processing – compare the block based approach using multiple tuples in section 1.2.2 – which is an approach where multiple tuples are handled at a time instead of a singular one. Since these blocks contain many tuples they can either be sorted column-wise or row-wise in memory. Tables 1.1, 1.2 and 1.3 are given as a visual aid where the first one represents a sample block containing three tuples. In table 1.2 a row-wise sorting is shown and in table 1.3 a column-wise sorting is shown.

Their findings suggest that the best choice depends on use case, block size, possibility to use SIMD or not. They do note that conversion between the memory layouts can be done quite quickly and cheaply, showing that a query can benefit from utilizing different layouts at different stages.

1	2	3	4
5	6	7	8
9	10	11	12

Table 1.1: Sample matrix with columns coloured to highlight different layouts

1	2	3	4	5	6	7	8	9	10	11	12

Table 1.2: Row-wise sorting of sample matrix

1	5	9	2	6	10	3	7	11	4	8	12

Table 1.3: Column-wise sorting of sample matrix

1.3 Contribution

The idea of using and comparing different memory layouts inside of a query engine during execution has been researched earlier, for example as discussed in section 1.2.3. This also applies to the case of comparing a vectorized and compiled approach, as seen in section 1.2.2. Neither of these cases concerns a graph database however, which this work seeks to investigate.

Chapter 2
Background

2.1 Relational databases

A relational database uses the relational model, introduced by Edgar F. Codd in 1970 [2]. The relational model uses tables to describe entities. Each tuple, row, is an entry and each column is an attribute for a given entity. A key is one or more attributes that together form a unique identifier for a tuple. Using these keys it is possible to find an entity in a different table.

A classic example concerns movie stars and movies. There exist many movies and many movie stars and a movie can have a cast of many movie stars. Likewise, a movie star might appear in many movies. To connect this many-to-many relation between movies and stars, a third table called StarsIn is created. This third table contains keys that uniquely define both the movie and the star to ensure an unambiguous connection [3].

For relational databases there exists a principal query language called SQL (Structured Query Language) with an ISO standard [6]. An example SQL query is given in SQL query 2.1 where the keywords for the language are written in all uppercase. The FROM keyword indicate what table to look at, the WHERE keyword places a conditional requiring all returned tuples to match the condition. Finally, the SELECT keyword indicates which attributes to include in the tuple. In the example, the star matches all attributes, but the query could be limited to only match some.

```
SELECT *
FROM Movies
WHERE name = 'The Matrix' AND year = 1999
```

SQL query 2.1: SQL query which returns all attributes for the movie named The Matrix released in 1999

2.2 Graph databases

Graph databases, as opposed to the relational kind previously described, utilize a graph data model. In this model relationships are materialized, instead of inferred by comparing keys. This key difference allows for major speedups when handling highly connected data since the graph database can take advantage of the relationship pointers, compared to costly join operations in the relational case .

Comparing the movie and movie star example given previously for relational databases, section 2.1, we see that the additional table for StarsIn could be represented by a relationship instead, directly connecting movies and movie stars [14].

In Neo4j specifically, the property graph model is used. In comparison to a regular graph, the vertices are called nodes and the edges are called relationships. Additionally, nodes and relationships may be marked with a label to specify what sort of entity they represent; compare this labelling to the table representation in a relational database. This means that there is a way of grouping nodes together: in the movie example from earlier one would label all stars with a star label, all movies with a movie label and all relationships connecting them with a starsIn label [9].

2.3 The Cypher query language

The Cypher query language (Cypher from here on) is a query language created by Neo4j specifically to query graphs. Cypher aims to be easy to read and understand by using ASCII art to allow a user to almost draw queries [14] .

In 2.1 an example is given, where ASCII arrows represent directed relationships, and nodes are surrounded by parentheses to form round nodes, how they are normally pictured in graphs.

openCypher is the specification for Cypher and works to create a standardized query language for graphs , GQL, together with ISO [10] [5].

```
MATCH ({name:"Johan"}) -[:starredIn]-> () <-[:starredIn]- (costars)
RETURN costars.name
```

Cypher Query 2.1: A cypher query showing the relationships being represented by ASCII arrows

2.4 ACID

ACID is an abbreviation for Atomicity, Consistency, Isolation, and Durability. For a database system to be considered ACID compliant all four of these properties need to be guaranteed for each and every database transaction. Atomicity means that a transaction cannot be divided; the consequence is that either the entirety of a transaction succeeds, or the entirety of it fails (requiring all changes made as part of the transaction to be rolled back). Consistency guarantees that the database is in a valid state after each and every transaction; no constraints

or similar can be violated. Isolation ensures that transactions that are run in parallel yield the same result as if they had been sequential. Durability, finally, states that upon transaction completion, the change is permanently stored and able to survive a system failure. [13]

2.5 CPU infrastructure

This section on CPU infrastructure does not aim to fully describe the actual inner workings of the CPU nor the actual infrastructure of it. Instead, the goal of this section is to describe different quirks of the CPU that might affect performance.

Cache hit or miss

Modern CPUs use a tiered structure of memory caches starting with smallest and fastest caches moving onto larger and slower caches. Typically a CPU has three tiers of caches, labeled L1 through L3, with L1 being the smallest and fastest and L3 the largest and slowest.

A cache hit indicates that the data needed by the processor for the next instruction was able to be retrieved from the cache instead of from a different memory medium, i.e. RAM or disk. A cache miss, in contrast, is instead when the data could not be retrieved from the desired cache. Since getting data from either a slower cache, RAM or disk can be magnitudes slower, a cache miss might have massive impact on the performance of a program.

Since cache sizes are small, a different data layout might allow for the desired data to fit in a faster cache and therefore increase performance.

SIMD - Single Instruction Multiple Data

As the name hints at, Single Instruction Multiple Data, SIMD, is a type of parallelism applicable when a single CPU instruction should be used for multiple data entries. This parallelism allows for loading of multiple data entries as well as performing the desired instruction in parallel. Naturally this increases performance compared to the behaviour of a simple sequential loop. A requirement for SIMD to be possible is that the registers on the CPU are large enough to fit multiples of a given data type, e.g. a 256-bit register can fit four 64-bit integers.

In tables 2.1 and 2.2 an example through visualization is given. In table 2.1 we see that the registers are full and that addition is carried out for each column, resulting in increased parallelism. In contrast, table 2.2 shows that space is left empty inside of the registers and that only one addition is carried out. In order to add all the values done in SIMD example, a loop would have to be constructed.

The available SIMD instruction set depends on the CPU, but one example is Streaming SIMD extensions, SSE, for Intel x86 architecture.

First register	1	2	3	4
Second register	2	3	4	5
Result register	3	5	7	9

Table 2.1: An example of SIMD summing values from first and second registers in parallel

First register	-	-	-	1
Second register	-	-	-	2
Result register	-	-	-	3

Table 2.2: An example summation not using SIMD. Note the empty spaces in the registers

2.6 JMH

The Java Microbenchmark Harness, JMH, is just as the name suggests, a Harness built for creating and running benchmarks targeting the JVM. [11]

2.7 Java incubator module and Vector API

A Java incubator module is meant as a way for JDK release projects to provide a tool or API to developers in order to receive feedback. An incubator module makes no promise that the current API will remain the same upon release, or even that it actually will be released. The incubator module is strictly opt-in and uses a custom prefix of jdk.incubator. [1]

The Java Vector API is an incubator module which strives to bring an API for vectorized code into Java. It aims to provide an easy to use and understand API, work regardless of platform and in case it fails to created vectorized code, fall back on a different solution. [15]

In Scala code listing 2.1 an example using the vector API is given. The method in question simply adds the values from two arrays together and outputs them to an output array. Note how the SPECIES variable relegates to SPECIES_PREFERRED. This allows the API to determine the proper amount of lanes to use given the architecture the machine runs and the data type in use. The method call getLongVector can then collect the right number of values from the array in question. Note that a second non-vector loop to handle the tail of the arrays is required, which is left out in the example.[12]

```
def vectorAddition(a: Array[Long],
                   b: Array[Long],
                   c: Array[Long]): Unit {
    val SPECIES: VectorSpecies[java.lang.Long] =
                            LongVector.SPECIES_PREFERRED
    while (i < SPECIES.loopBound(a)) {
        val va = a.getLongVector(SPECIES, i, a)
        val vb = b.getLongVector(SPECIES, i, b)
        val resultVector = va.add(vb)
        resultVector.intoArray(c, i);
        i += SPECIES.length()
    }
}
```

Scala code 2.1: A method to add two arrays together using the Vector API

Chapter 3
Query Engine Architecture

This section describes the inner workings of the query engine inside of the Neo4j database. It aims to describe how a query posed by a user in the Cypher query language is transformed into actual results for the user, but also break down each step and explain what is consumed and produced by each process.

Figure 3.1 shows a flowchart demonstrating the process the query engine carries out in order to turn the user query into results.

3.1 Parsing

Parsing is the very first step performed by the query engine once given a query by the user. The input from the user is a simple string which is analysed, or parsed, partly to ensure that the query posed by the user adheres to the rules of the Cypher language but more importantly to analyse what the user wants to achieve. The goal is to produce an Abstract Syntax Tree (AST) which is a tree expression of the query. In Cypher query 3.1 an example query is given; a corresponding albeit simplified AST can be seen in figure 3.2. The query matches all nodes with the label "Foo" given that the attribute "value" on them is less than 3. Finally it then returns the name attribute of all the nodes which it matched. In the AST we see that the query is split into a match and a return subtree. The match subtree contains the pattern matching information about the name of the label as well as the information about the conditional on the value created by the where clause. Finally the return subtree specifies what to return. Since this AST was simplified for ease of explanation, do note that this does not reflect what an actual AST created by a Cypher parser would return.

Whether or not the created AST does in fact make sense or is valid is not evaluated during parsing, only that an AST can be created is verified. An example of where a valid AST can not be created would for instance be if the literal 3 in the where clause was left out, or if the rules of the Cypher language were broken.

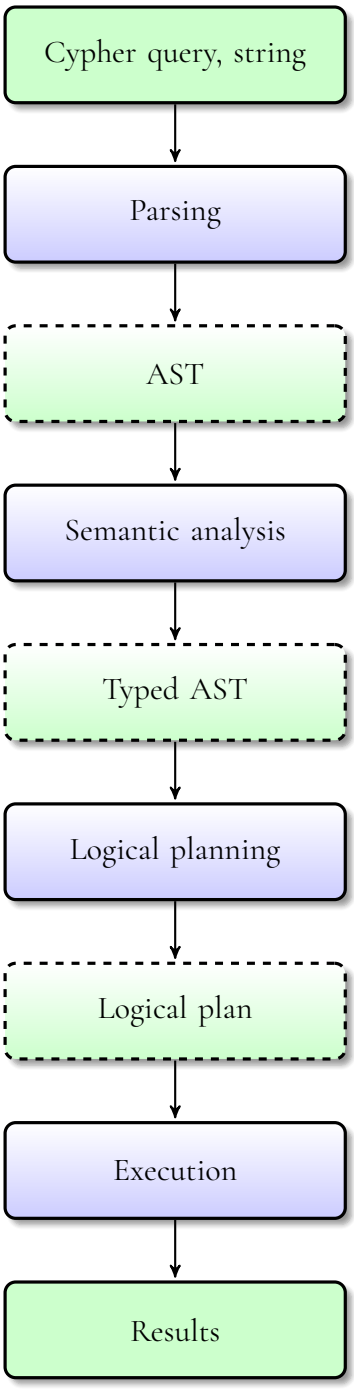


Figure 3.1: A flowchart demonstrating the process inside the query engine. Start and end values for the process are shown with full borders and green background. Steps in the process are shown with full borders and a blue background. Intermediate products created by a step to be input for a following step are shown with dashed borders.

```
MATCH (n:Foo)
WHERE n.value < 3
RETURN n.name
```

Cypher Query 3.1: Simple sample query

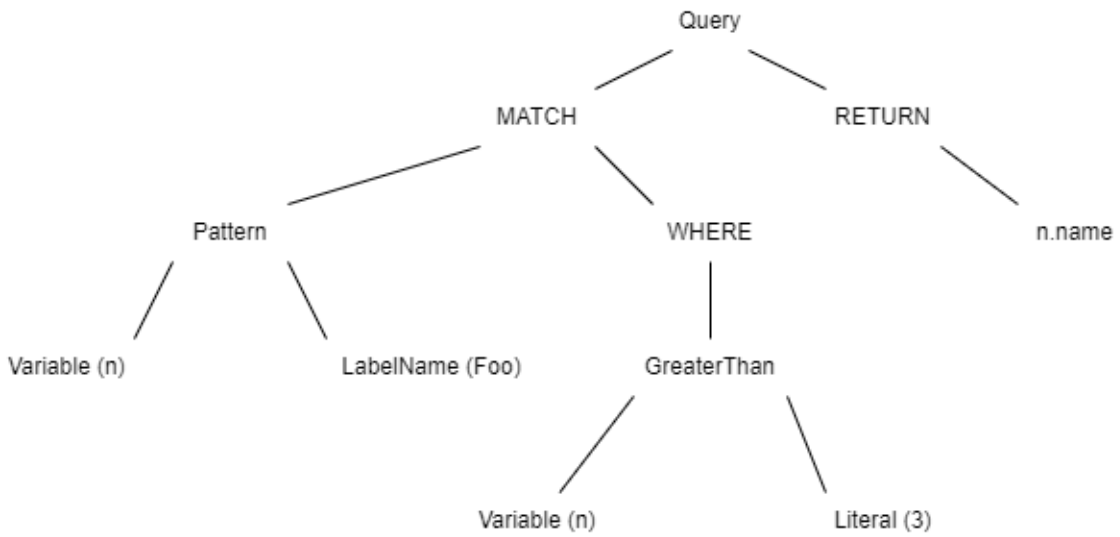


Figure 3.2: Simplified AST corresponding to sample Cypher query

3.2 Semantic analysis

Once the query string has been turned into a valid AST during parsing, the AST undergoes semantic analysis. This step ensures that the created AST does in fact make sense and is valid to proceed with. The main tasks during semantic analysis are checking type compatibility and scoping of variables. Type compatibility ensures that operations make sense, comparing a node to a numerical number being an example of a type incompatibility. Scoping of variables verifies that variables are declared before they are used, since calling an undeclared variable is not valid. Once the semantic analysis is complete, the AST has been turned into a typed AST.

In the Neo4j query engine the AST is also rewritten to normalize and optimize it. This allows the AST to be cached and reused for later queries. Due to this rewrite of the AST, two queries that differed when they were simple strings might end up with the same typed AST.

3.3 Logical planning

During logical planning the goal is to find the best possible plan to solve what the user requested. Here a "plan" simply refers to a possible way of solving the request. Since there are probably many different ways of solving a query, different plans need to be compared in order to determine the best one. This is achieved by using statistics about the data store and estimating cardinality. The plan with the cheapest cost is then selected as the desired logical

plan. This plan is found using Iterative Dynamic Programming (IDP). [7]

The logical plan can be cached and reused in a similar manner as the AST. This means that the logical planning step can be skipped to save time on future user requests. The plan does however risk going stale if the data store changes since the statistics used to generate it might be outdated. If this is the case a new logical plan can be generated.

3.4 Execution

Once a logical plan has been generated and selected it needs to be executed to actually deliver results to the user. In Neo4j there exist several runtimes capable of executing the logical plan, however only the pipelined runtime will be discussed in this thesis since that is the current state-of-the-art runtime which will be used as a baseline as well as the runtime inside of which changes will be tested.

3.4.1 Pipelined runtime

The pipelined runtime is a block-oriented runtime where the blocks are morsels as described in section 1.2.1. As noted previously in that section, this allows for elastic parallelism with great throughput. To materialize results between pipelines buffers are used.

3.4.2 Fusing operators and compilation

The pipelined runtime tries to fuse operators into longer pipelines to avoid having to materialize results. It also tries to compile these pipelines into optimized executable code for a given pipeline. However, as the number of operators and combinations increases it is not always possible to fully fuse and compile an entire pipeline. In these cases a pipeline can be divided into smaller pipelines which in turn can be completely compiled with the drawback naturally being the need to materialize the results between pipelines adding an overhead cost.

In code listing 3.1 pseudocode explain the process of executing a scan of all nodes, followed by a filter operation and finally producing the results for the user. As can be seen by the pseudocode; there is a need to materialize the intermediate results after each operation. In contrast, in code listing 3.2 pseudocode show a fused approach which combines all three steps inside of one loop, eliminating the need to materialize the intermediate results.

3.4.3 Producing results

Once the suitable pipelines corresponding to the logical plan have been created, simply executing them yields the desired result of the query to the user. Since a pipeline is fully executed to the end and pipelines can be executed in arbitrary order given no need for synchronization, the latency of the first result can be greatly improved compared to the approach of executing all morsels for the initial pipeline first.

```
// Scan all nodes
while (i.size && allNodes.next) {
        out.set(allNodes.node)
}

// Filter
for (data:out) {
        if (predicate(data)) {
                out.set(data)
        }
}

// Produce Results
for (data:out) {
        writeDataToUser(out)
}
```

Pseudocode 3.1: Pseudocode showing how to scan all nodes, filter out on a given predicate and presenting the results to the user

```
// Scan all nodes
while (i.size && allNodes.next) {
    // Filter
        if (predicate(allNodes.node)) {
            // Produce Results
                writeDataToUser(allNodes.node)
        }
}
```

Pseudocode 3.2: Pseudocode showing a fused version of how to scan all nodes, filter out on a given predicate and presenting the results to the user

Chapter 4
Implementation

This section details what work has been carried out implementation-wise. It specifies how a column-wise morsel layout was achieved, what operators were adapted to use this layout, a description of them and what they do, as well as how these operators were rewritten to work with the alternate layout.

4.1 Implementing column-wise memory layout

To allow for parallel query execution, the query engine in Neo4j uses Morsel driven parallelism, described in the related work section. A morsel can be considered a block for block-wise execution, though a morsel is always fixed in size with regards to the number of rows (tuples). Representing block-wise data in a morsel is done with an array. If a row-wise memory layout is used rows are placed sequentially meaning each row achieves good data locality. Finding a specific element inside this array representation can be easily done using equation 4.1. This equation specifies that to find the index of a desired element it can be found by multiplying the row it is in with the number of elements per row and adding the index that element has on that row. The corresponding applies for a column-wise sorting, where equation 4.2 details how to find the index of the desired element in a column abstraction. In detail, the equation specifies that the index of the desired element is the index of the column the element is in times the number of elements per column and the index the element has in that column. Substituting these variables gives a way of relating the column-wise sorting in a row abstraction, since the desired column corresponds to the desired index in a row and elements per column simply equals the total number of rows combined with the fact that the index in the column equates to the desired row we reach equation 4.3. As the previous sentence outlined, this equation states the same as equation 4.2 but with row abstractions.

Utilizing these equations a morsel which is column-wise sorted can be implemented, still

represented by an array but with indices calculated accordingly to correspond to setting and getting values along with keeping track of rows and columns.

$$Index = desired\_row * elements\_per\_row + index\_on\_row \tag{4.1}$$

$$Index = desired\_column * elements\_per\_column + index\_in\_column \tag{4.2}$$

$$Index = index\_in\_row * number\_of\_rows + desired\_row \tag{4.3}$$

4.2 Projections and the project operator

The project operator is used when a specific column or columns are desired; compare the projection operator in relation algebra. In addition to yielding a column, it can also be used for arithmetic operations such as adding or multiplying, either applying to a column and constant or two columns. Three solutions for comparing memory layouts for projections were created. The first simply changed what memory layout the morsels were created with and updated accessing methods to yield correct indices. However, since the original implementation of the operator relied upon a row-wise access pattern it was not suitable to take advantage of the column-wise memory layout. Therefor a second solution which better utilized column-wise access was created. This solution inverted the loop order from being row first, column second, to instead being column first. This aligns better with the column-wise memory layout from a data access point of view. Third and last a column-first solution using SIMD was designed to best take advantage of the column-wise layout though it works with row-wise morsels as well. This solution was built using the Java Vector API for the SIMD part.

4.3 Filter operator

The filter operator is used when filtering is required. Examples of possible operations are sorting out and throwing away all results which do not meet a criteria or a certain predicate. While this operator never was implemented as part of this thesis due to a limitation in the Vector API clashing with the implementation of Neo4j, the possibilities of applying SIMD and a columnar memory layout for the filtering problem were considered and examined.

4.4 Aggregation operator

The aggregation operator is used when the results need to be aggregated to a singular result. Examples of possible operations are finding the greatest or smallest value in a collection, or finding the sum if applicable for the collection. Since morsels are used a pre-aggregation step can be executed where aggregation is executed for each morsel, and then a final aggregation of the partial results can be executed.

Since the final aggregation is independent of memory layout, the pre-aggregation step is the interesting part. Since the desire is to aggregate a singular column, a column-wise

memory layout should be able to perform well. The data is also well suited for SIMD since lane reduction can be utilized. First a solution which only changes the memory layout from row-wise to column-wise was built, and secondly a SIMD implementation was created using the Java Vector API. The SIMD solution works independently of memory layout but uses a column-wise access pattern.

4.5 Column-wise copying between morsels

This step is technically not an operator but still worth examining. Since morsels need to be reallocated at the end of each pipeline to avoid uneven workloads, if this process could benefit from a different data layout there might be major gains. However, there exists no general copy case between pipelines since it depends on the operators in question. For the experiment a limitation to only the unwind operator was selected since this operator has a tiny overhead and does not increase cardinality when only unwinding a single number, such as 1 to 1. With this special case in mind, the unwind operator was adapted specifically to be able to copy between pipelines.

The new solution uses SIMD for loading from and storing into morsels, built using the Java Vector API. The implementation was built to work according to a column-wise access pattern but is independent of the underlying memory layout meaning it works for row-wise morsels as well.

4.6 Benchmark design

Micro benchmarks were created by designing a logical plan which was to be executed that heavily relied upon the operator the benchmark was intended to test. In order to test the project operator this meant a plan which only wanted to execute a simple arithmetic operation such as adding a fixed value to all values in a column, or multiplying two columns. For aggregation this means a plan which only wants to find an extreme value or the sum for a given column.

To reduce the overhead during benchmarks, and thereby increase time spent in the desired operator, values were directly supplied which eliminates several steps which a real execution would require to be able to produce results.

Since the logical plan is already in place for the benchmarks, the normal steps a query takes described in section 3 of parsing, semantic analysis and logical planning are all skipped and only the execution stage remains. This further removes overhead which is not of interest and focuses the benchmarks on the parts desired to be evaluated.

To further test the differences a parameter called stride was introduced. This parameter determines the amount of extra attributes supplied as input. This can be used to simulate a query which contains several extra attributes which are not used in the current step of the query but are required at a later point. Increasing the number of attributes affects the morsel, both in size and composition which could affect performance.

The specifics of each benchmarks will be described in greater detail in each corresponding section in the results section.

Chapter 5
Evaluation

This section presents the details surrounding each designed benchmark along with their results for each operator as well as the column-wise copying.

5.1 Experimental setup

All of the benchmarks presented were run on a dedicated server with the following hardware and software installed:

- CPU: 4-core Xeon Skylake-DT
- RAM: 64 GB
- STORAGE: 480 GB SSD and 3TB HDD
- OS: Ubuntu 20.04.2, Kernel 5.4.0-74-generic
- Java Version: "17.0.1" 2021-10-19 LTS Oracle

Do note that the CPU in question supports up to AVX2 which allows 256-bit words, meaning 4 64-bit integers fit.

5.2 Results

5.2.1 Projection operator

Benchmark design

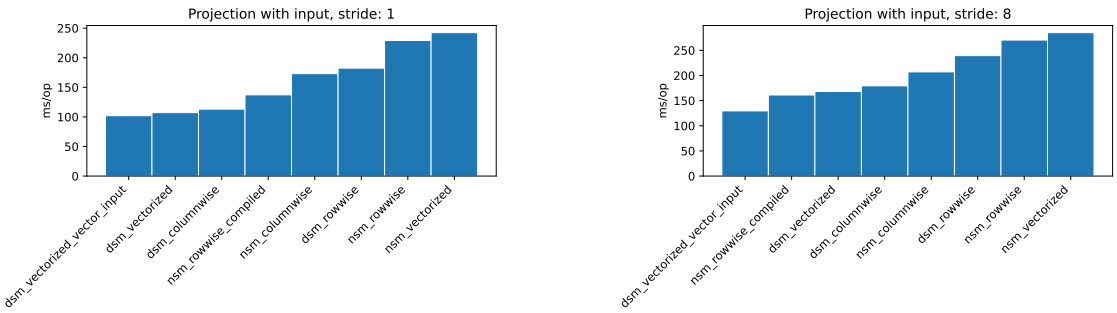
The plan used for the projection benchmark consists of an input operator, followed by the projection operator and ends by producing results. The projection operator is given the task

of adding a constant to the entirety of a column. A secondary benchmark which increased the number of project operators to increase the time spent projecting was also created.

Benchmark results

In figure 5.1 the benchmark results for the projection operator are shown. The current state-of-the-art implementation is labeled *nsm_rowwise_compiled*. The *nsm* and *dsm* prefixes indicate what memory layout is in use, the *vectorized* keyword indicates a case using the new SIMD implementation, *rowwise* indicates usage of the pipelined runtime without fusing or compiling the plan and *columnwise* indicates the usage of the column-wise implementation which does not leverage SIMD. The postfix *vector_input* indicates that a faster, column-wise and block-oriented method has been used for the input part of the plan.

The new vectorized implementation outperforms the compiled one for a low stride value. However, when stride is increased, the compiled variant slightly outperforms the vectorized solution unless the faster input solution is leveraged. If the faster input method is used the vectorized solution is still the fastest.



(a) Projection, stride = 1

(b) Projection, stride = 8

Figure 5.1: Results for projection operator

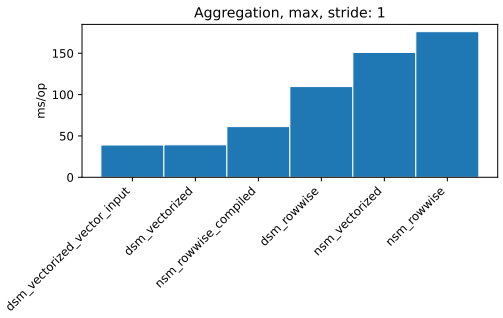
5.2.2 Aggregation operator

Benchmark design

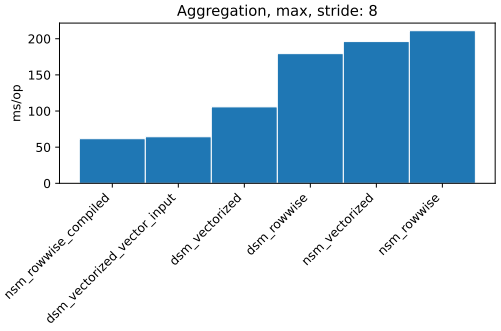
The plan used for the aggregation benchmark consists of an input operator, followed by the aggregation operator and ends by producing results. The aggregation operator is given an aggregation function to carry out: summation, maximum value, or minimum value.

Benchmark results

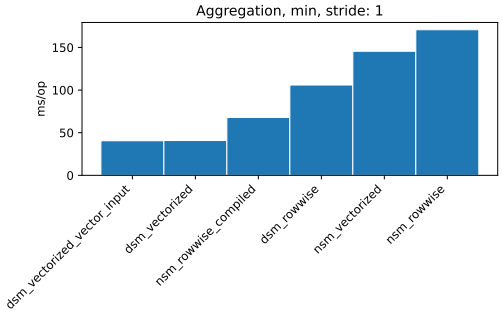
In figure 5.2 the benchmarks results for the aggregation operator, using the previously described plan, are shown. The different plots show different aggregation functions in use - maximum value, minimum value, and summation - as well as different stride, extra unused columns inserted into the input set and the morsels. The current state-of-the-art implementation is labeled *nsm_rowwise_compiled*. The *nsm* and *dsm* prefixes indicate what memory



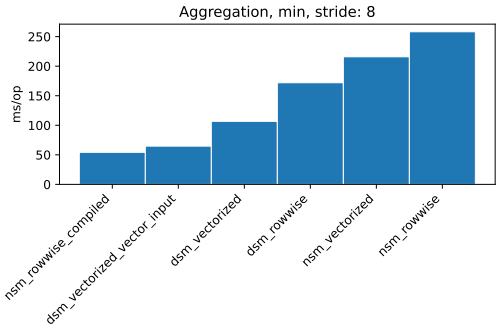
(a) Aggregation, max, stride = 1



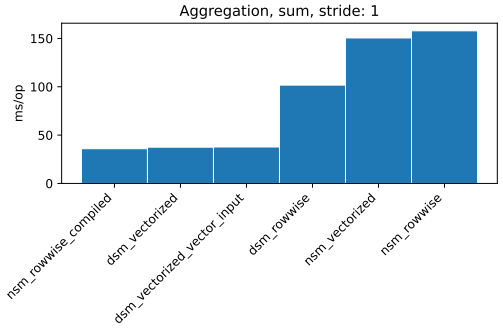
(b) Aggregation, max, stride = 8



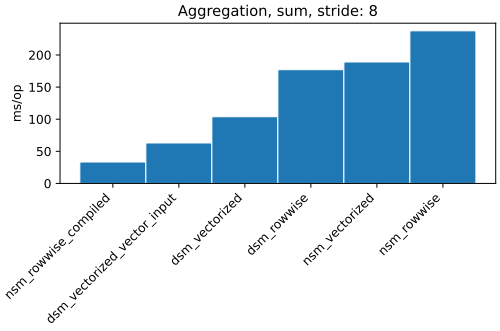
(c) Aggregation, min, stride = 1



(d) Aggregation, min, stride = 8



(e) Aggregation, sum, stride = 1



(f) Aggregation, sum, stride = 8

Figure 5.2: Results for aggregation operator

layout is in use, the *vectorized* keyword indicates a case using the new SIMD implementation and *rowwise* indicates usage of the pipelined runtime without fusing or compiling the plan.

The new vectorized implementations outperforms the compiled one in the case of finding maximum and minimum value for low stride and is roughly equal to the compiled in the case of summation. When stride is increased the compiled solution outperforms the vectorized solution. Note that all implementations except the compiled one perform worse when the stride is increased.

5.2.3 Column-wise copying between morsels

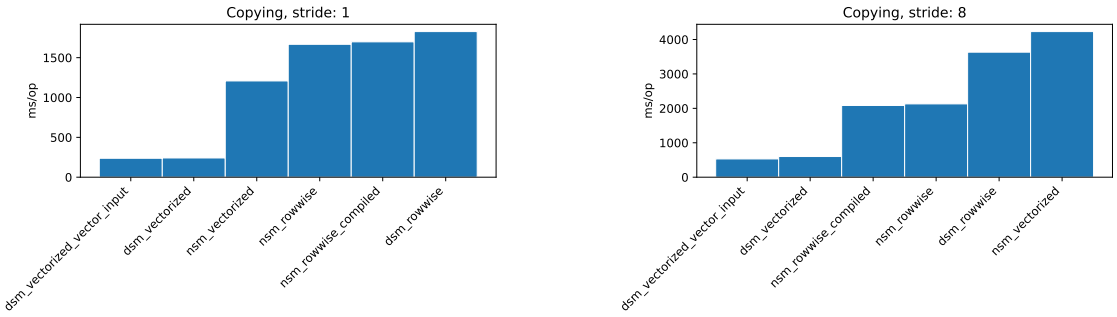
Benchmark design

The plan used for benchmarking copying between morsels consists of an input operator, followed by several unwinds which require materializing results, and finally producing results. Since the unwinds are very limited, the overhead work should be limited compared to the work of materializing and copying.

Benchmark results

In figure 5.3 the benchmark results for the copying between morsels are shown. The current state-of-the-art implementation is labeled *nsm_rowwise_compiled*. The *nsm* and *dsm* prefixes indicate what memory layout is in use, the *vectorized* keyword indicates a case using the new column-wise, vectorized implementation and *rowwise* indicates usage of the pipelined runtime without fusing or compiling the plan. The postfix *vector_input* indicates that a faster, column-wise and block-oriented method has been used for the input part of the plan.

Do note that this benchmark was designed to test copying between morsels, meaning that fusing of operators has been disabled.



(a) Copying, stride = 1

(b) Copying, stride = 8

Figure 5.3: Results for copying between morsels

Chapter 6
Discussion

This section discusses the achieved results presented in the previous section, their meaning, their cause and potential improvements that could be executed or learned from. Section 6.1 discusses each operator implementation and their results in depth, section 6.2 discusses the unimplemented filter operator, ideas on how to implement it and the strengths and weaknesses of these implementation possibilities. Section 6.3 presents potential improvements that could be implemented to achieve greater results, section 6.4 discusses general limitations with the implementations as well as the results, and finally 7.2 outlines possible future work.

6.1 Operator performance

This section dives into each operator and considers the advantages and disadvantages that were achieved with them.

6.1.1 Project operator

The results for the aggregation operator show that a performance improvement compared to the state-of-the-art is achieved by using the vectorized solution in combination with a column-wise memory layout. The solution built to better utilize the column-wise access pattern also outperforms the compiled solution when used in combination with column-wise access. However, since this solution also showed improvement when using a row-wise memory layout compared to the uncompiled solution using row-wise access it is likely that the improvement brought on by this implementation was not simply derived from better memory alignment. Only changing the memory layout yields an improvement when compared to the uncompiled solution using row-wise access, although it is clearly slower than the compiled solution. Finally, the vectorized solution performs the worst out of all solutions when combined with the row-wise access pattern. Profiling reveals that a great amount of time is

spent getting and setting values to correct spots in the morsels, since the desired values are scattered all over.

Increasing the stride show that the compiled implementation outperforms the vectorized with column-wise memory layout unless the faster vectorized input method is also used. Profiling does reveal however, that with increased stride a fairly small amount of work time is spent during actual projection, and a greater time is spent for input and output.

Since the vectorized implementation relies upon SIMD and the hardware used to capture the results support up to AVX2, faster results could likely have been achieved on newer hardware with support for more lanes, such as AVX-512.

6.1.2 Aggregation operator

The results for the aggregation operator are not quite as clear as for the projection operator. When considering low stride the vectorized solution outperforms the compiled one while finding a maximum or minimum value, and performs evenly when summation is in question. Just changing the memory layout from row-wise to column-wise does show improvement compared to the uncompiled solution, but is definitely worse than the compiled one.

When the stride is increased the compiled solution is affected the least of all candidates and manages to outperform the vectorized one in all cases of aggregation function. Only when the vectorized version using the improved vector input is considered is it comparable to the state-of-the-art solution. In the case of maximum and minimum, this vectorized variant is almost on equal footing, but in the case of summation it still lags quite a bit behind. Profiling on these shows that similarly to the projection case, a greater deal of time is spent on just the input and output operators rather that the actual operator of interest, the aggregation one, which further explains why the vectorized input variant shows such improvement with larger stride.

The point regarding newer hardware, previously from section 6.1.1, also applies to the aggregation operator.

6.1.3 Column-wise copying

The results for the column-wise copying show a massive improvement when using the block-oriented column-wise approach in combination with a column-wise data layout in comparison to the state-of-the-art solution. Using the new approach also shows improvement in combination with a row-wise memory layout for low stride values. This gain, however, is severely smaller than the one achieved with the column-wise memory layout. Unsurprisingly, the row-wise approach in combination with the column-wise memory layout performs the worst overall since the change in memory layout only hinders the process.

With increased stride the new approach in combination with row-wise memory layout quickly deteriorates. This is likely due to increased tension using a column-wise access pattern with row-wise memory layout. Increasing stride also shows slowdown across the board, which is expected since there are more values to copy. The relative improvement achieved by the new approach compared to the compiled is reduced, decreasing from a speedup of roughly 7 times to only 3.5.

Since fusing is disabled for the benchmark – due to the desire to examine copying between morsels – one of the greatest advantages of the state-of-the-art compiled implementation is

eliminated: the ability to fuse operators removes the need to materialize results. The disabling of fusing also explains why the results marked *nsm_rowwise* and *nsm_rowwise_compiled* perform so evenly; they are essentially the same.

It is possible that the row-wise copying behaviour could stand to benefit from a block-wise approach instead of the current one tuple at a time version and that this might close the gap to the new column-wise behaviour.

6.2 The filter operator

Implementing a filter operator to use SIMD can be done since lane-wise comparisons can be executed. The results can then be stored as a mask which signifies what tuples to carry forward or not. Here a problem arises since in order to sort only the valid values from here a scalar pass through the mask is required and reallocation into new morsels has to be done. The alternative is to keep the mask and continue operations with morsels unaffected. This has the potential of also causing a loss in performance since the degree of valuable computation decreases to the selectivity of the predicate. This completely nullifies the gains of an early eager sort.

6.3 Potential improvement

A different approach to increase the performance of a query would be to combine a vectorized and compiled solution into one. If the compiled solution could carry a block through an entire pipeline one might achieve the benefits of both compilation and SIMD at once, although it requires the blocks to be small enough to avoid the need to materialize them. The *LongVector* type in the Vector API might be able to accomplish this.

6.4 Limitations

As shown the DSM memory layout performs better in combination with vectorization and the use of SIMD. Unfortunately the use of SIMD also limits the possible use cases since SIMD only works with value types. This means that any query concerning references (such as strings) or the usage of null will not be able to fully leverage the advantages of a different memory layout.

Another limitation is connected to the scope of the project, operators that increase the cardinality. Since increasing cardinality changes the memory layout and data locality it would greatly affect the performance but this has not been examined in this thesis. Due to cardinality-increasing operators being common in queries, this is possibly a major disadvantage for the DSM memory layout.

A major obstacle that stops the implemented operators from being possible candidates for usage inside of Neo4j today is that they rely upon an incubator module for SIMD code since they are implemented using the Vector API. This obstacle alone means that the operator implementations are not deployable until, and only if, the Vector API is adopted into Java.

The final limitation is in regard to the results presented. Since the implemented operators were selected since they were believed to be the ones that could benefit the most from the column-wise memory layout, it is likely that further operators will not profit to the same extent from this change.

Chapter 7
Conclusions and future work

7.1 Conclusions

In conclusion, it has been shown that using an alternate memory layout can result in improved performance when combined with vectorized code inside the query engine of the graph database Neo4j. This performance increase is however severely limited due to only being usable for value types and not applicable to all operators.

The experiments carried out also revealed that only changing the memory layout from NSM – row-wise – to DSM – column-wise – is not sufficient in order to achieve improvements when comparing to the state-of-the-art compiled strategy but does offer slightly increased performance compared to the pipelined runtime when fusing and compiling is not possible.

In a similar vein, for the operators tested in this thesis only changing to a vectorized approach without changing the memory layout to a column-wise ordering actually has a negative impact on performance, even compared to when fusing and compilation is disabled.

The largest advantage seen among the new implementations was the column-wise copying between morsels which implies that queries requiring materialization between pipelines many times could stand to benefit greatly.

Finally, the implementations created as part of this thesis are not suited for a commercial product since they rely upon an incubator module.

7.2 Future work

7.2.1 On-the-fly transition

As noted in section 1.2.3, Zukowski et al. suggest that on-the-fly transition between DSM and NSM is a valid approach for certain queries. In order to examine this inside of Neo4j

further investigation of possible speedup for more operators need to be conducted, together with examining the speed of changing the memory layout for morsels.

7.2.2 Combining compilation and vectorization

As noted in both section 1.2.2 and 6.3 combining compilation and vectorization can probably yield even greater results than just one of the strategies. The suggested approaches of doing this in both section vary however, with the one suggested Sompolski et al. being more akin to changing between the strategies in a clever way, and the one suggested in this thesis rather brings the vectorization inside of the compilation-loop.

References

[1] Alex Buckley Chris Hegarty. Jep 11: Incubator modules. `https://openjdk.java.net/jeps/11`.

[2] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[3] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems The Complete Book 2nd Edition*. Pearson Prentice Hall, 2009.

[4] Goetz Graefe. Encapsulation of parallelism in the Volcano query processing system. *Proceedings of the 1990 ACM SIGMOD International Conference: Management of Data*, pages 102–111, May 1990.

[5] ISO. Information Technology — Database Languages — GQL. `https://www.iso.org/standard/76120.html`.

[6] ISO. Information technology — Database languages — SQL — Part 1: Framework (SQL/Framework). `https://www.iso.org/standard/63555.html`.

[7] Donald Kossmann and Konrad Stocker. Iterative Dynamic Programming: A New Class of Query Optimization Algorithms. *ACM Trans. Database Syst.*, 25(1):43–82, mar 2000.

[8] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. Morsel-driven parallelism. *Proceedings of the 2014 ACM SIGMOD International Conference Management of Data*, pages 743–754, June 2014.

[9] Neo4j. Neo4j Developer Guide. `https://neo4j.com/developer/graph-database/`.

[10] openCypher. openCypher. `https://opencypher.org`.

[11] openJDK. JMH. `https://openjdk.java.net/projects/code-tools/jmh/`.

[12] Oracle. Vector API Documentation. `https://docs.oracle.com/en/java/javase/17/docs/api/jdk.incubator.vector/jdk/incubator/vector/package-summary.html`.

[13] Alex Petrov. *Database Internals*. O'Reilly Media, Incorporated, 2019.

[14] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph Databases*. O'Reilly Media, Incorporated, 2013.

[15] Paul Sandoz. JEP 417: Vector API (Third Incubator). `https://openjdk.java.net/jeps/417`.

[16] Juliusz Sompolski, Marcin Zukowski, and Peter Boncz. Vectorization vs. compilation in query execution. *Proceedings of the Seventh International Workshop: Data Management on New Hardware*, pages 33–40, June 2011.

[17] M. Zukowski, N. Nes, and P. Boncz. DSM vs. NSM: CPU performance tradeoffs in block-oriented query processing. In *Proceedings of the 4th International Workshop on Data Management on New Hardware, DaMoN'08 - In conjuction with ACM SIGMOD/PODS Conference*, pages 47–54, CWI, 2008. Proceedings of the 4th International Workshop on Data Management on New Hardware, DaMoN'08 - In conjuction with ACM SIGMOD/PODS Conference.

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2022-06-20

EXAMENSARBETE Row vs. column data layout in a graph database query engine
STUDENT Eric Sporre
HANDLEDARE Henrik Nyman (Neo4j), Luigi Nardi (LTH)
EXAMINATOR Jacek Malec (LTH)

Alternativ minnesrepresentation för snabbare grafdatabaser

POPULÄRVETENSKAPLIG SAMMANFATTNING **Eric Sporre**

Grafdatabaser blir allt vanligare och är essentiella för allt mellan snabba rekommendationssystem till att tolka de ökända Panamadokumenten. I det här arbete undersöks en alternativ representation inom datorminnet som kan leda till snabbare resultat.

På moderna processorer sitter det flera temporära minnen i olika storlekar. De minsta är de absolut snabbaste medan de större är långsammare. Då de temporära minnena tenderar att vara små är det viktigt att anpassa den data som passerar genom processorn så att den passar om man vill uppnå hög prestanda. Om processorn är ute efter data som inte finns i sitt temporära minne tvingas den hämta den från antingen primärminnet (RAM) eller i värsta fall disk vilka båda är magnituder långsammare. Det kan jämföras med ifall man har alla ingredienser inför matlagning i kylen eller ifall man behöver åka till affären och handla.

Jag har arbetat med en alternativ sortering av block av data, tänk en tabell med rader och kolumner som nu behöver representeras på bara en rad. Normalt sorteras dessa block radvis men istället har jag försökt sortera dem utefter kolumnerna. Bilden visar ett mindre färgkodat exempel. Den bakomliggande tanken är att detta skulle kunna leda till att mer intressant data för processorn hamnar nära varandra då alla värden inom varje kolumn hör samman.

Den nya sorteringen testades för vissa mindre delar inom grafdatabasen och visade på lovande resultat ifall den kombinerades med nya metoder som kan dra ytterligare nytta av sorteringen. Dessa nya metoder kan utnyttja processorn bättre och föredrar den nya sorteringen. Det krävs dock mer undersökning innan man definitivt kan övergå till den nya sorteringen då den bara testats inom en mindre del av grafdatabasen.

1	2	3	4
5	6	7	8

Tabell 1: En tabell med data som ska representeras på en rad istället för två

1	2	3	4	5	6	7	8

Tabell 2: Radvis sortering av datatabellen

1	5	2	6	3	7	4	8

Tabell 3: Kolumnvis sortering av datatabellen