Avantstep

parallel query execution in graph databases

van der Looij, J.W.F.M.C.

Award date:
2021

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Database group
Graduation project

Thesis

Avantstep: parallel query execution in graph databases

Computer Science and Engineering
Quartile 3-4: 2020-2021

J.W.F.M.C. van der Looij j.w.f.m.c.v.d.looij@student.tue.nl

Supervisors
N. Yakovets n.yakovets@tue.nl
A.A.G van de Wall a.a.g.v.d.wall@tue.nl

Eindhoven, July 12, 2021

Abstract

Modern servers contain enough computing power and available memory that just a few decades ago was only available in large distributed computing clusters. Due to the increase in amount and speed of memory, the execution becomes limited by the computational bottleneck instead of the I/O bottleneck. This thesis presents a graph execution engine designed to solve this computational bottleneck by utilising the parallelism available in modern servers. The execution engine is based on the relational execution engine called Quickstep. We describe how the design of the relational engine is modified and extended for use in graph databases, which is then implemented in AvantGraph, an existing graph database. Furthermore, we present experimental results comparing the performance of our parallel graph execution engine to the existing sequential graph execution engine in AvantGraph, demonstrating that the parallel execution engine can both utilise the parallelism available in modern servers and outperform the sequential graph execution engine.

Table of Contents

1 Introduction

Graph databases have numerous applications spanning various domains. For example, graph databases are used to store and analyse different types of networks. These networks include social networks [34] and complex heterogeneous biological data [37]. Furthermore, graph databases are used to store and query graphs containing encyclopedic data, also known as knowledge graphs. Besides these applications, graph databases are used in other workloads that query the underlying graph structure of the data. Compared to traditional relational databases, graph databases provide significant performance advantages in these situations [16].

We are interested in the evaluation of *Union of Conjunctive Regular Path Queries (UCRPQ)*. These type of queries combine different types of graph evaluation namely, *subgraph matching* (also known as *Conjunctive queries (CQ)*) and *reachability* (also known as *Regular Path Queries (RPQ)*) [8]. Union of Conjunctive Regular Path Queries are often described using a high-level declarative language, such as SPARQL or RPGQ, in which the path predicates are defined. The output of these queries are all vertex binding which satisfy the path predicates.

There are various differences between relational and graph databases. For example, in relational databases the structure of the data must be know beforehand (schemas), whereas a graph database does not require a predetermined data structure [27]. Another difference is the type of queries which are executed. Although supported, recursive queries are not at the foundation of relation databases. Recursive queries are not supported by early versions of SQL, the query language used by most relational databases, nor *Relational algebra (RA)*, the formal query language SQL is based on [31]. Support for recursive queries was added in later versions of SQL and RA is often extended to include support for recursive queries. In contrast, recursive queries lie at the core of graph databases. One of the fundamental types of graph queries are reachability queries, which are based on recursion. Obtaining bounds on the output of queries is a heavily researched topic in the database community. A tight *Worst Case Optimal (WCO)* bound on the output size of conjunctive queries has been discovered by Atserias et al. [4]. One WCO join algorithm is the *Leap Frog Trie Join (LFTJ)* [36]. Although conjunctive queries can be found in relational databases, they are much more prevalent in graph databases as conjunctive queries are another fundamental type of graph query.

This thesis is primarily focused on the part of a *database management system (DBMS)*, which executes the queries, namely the *execution engine*. Before execution, a query must first be parsed and planned. The execution engine interprets the generated plan to execute. During interpretation, an operator tree is created in memory, and the execution engine navigates the operator tree executing the required code based on the operator type(s) in the tree.

Historically, I/O operations have been the dominating factor in execution time. However, with the introduction of SSDs, the cost of I/O operations have reduced significantly. Furthermore, with the ever increasing size of available main-memory, I/O operations are no longer necessarily the dominating factor. In addition, with sufficient main-memory, entire data sets can be stored in memory eliminating the I/O factor nearly completely. When data sets fit in memory, the dominating factor often becomes the execution of the operators themselves.

Extensive research has been done to ensure that the amount of work performed by these operators are as close to optimal as possible, which means that the dominating factor often cannot be solved by performing less work. The speed at which a single piece of work can be executed by modern hardware is often limited. However, with the introduction of concurrent execution, numerous different pieces of work can be executed simultaneously. Because the amount of work can often not be reduced, a possible solution to the current dominating factor is to perform the work concurrently.

Due to the change in dominating factor, different design considerations are required when building modern execution engines. A number of (relational) execution engines have already been designed with the change in dominating factor in mind, for example *Quickstep* [28]. However, due to the differences between relational and graph databases, solutions designed for relational databases cannot always be directly applied to graph databases.

Research Questions. Numerous modern execution engines have been designed for relational databases. However, due to the differences between relational and graph databases, these designs cannot always simply be applied in graph databases. Therefore, we try to answer the following: 1) How can the research in modern relational execution engine be adapted for graph databases? 2) How can the Transitive Closure and Leap Frog Trie Join be parallelised? 3) What are the effects of parallel execution on query performance?

As we answer these research questions, we furthermore contribute the following:

1. We describe and implement an adaptation of the

Quickstep design for use in Graph databases (Section 3.1),

2. We describe how the design for Quickstep can be extended to support cyclic queries (Section 3.2.2),

3. We describe how operators, such as Transitive Closure (Section 3.3) and Leap Frog Trie Join (Section 3.4), can be implemented in a parallel vectorised execution engine, which are then implemented,

4. We evaluate the effects of parallel vs sequential query execution on large real-world graphs and evaluate the parallel scaling capabilities in various different scenarios (Section 4).

The remainder of this thesis is organised as follows: Section 2 provides an overview of preliminaries and background information. In Section 3, we describe the design of the parallel execution engine along with the extensions required, highlight the similarities and differences with Quickstep's design, and finally, implement the design. Section 4 provides an evaluation of the parallel execution engine and a comparison with the original execution engine. We then compare our implementation to related work in Section 5. And finally, we conclude in Section 6.

2 Background

2.1 Query Semantics

Graph databases are designed to answer queries using the information encoded in graphs. In this thesis, we consider queries on property graphs. Property graphs are graphs where each vertex and edge are assigned a set of labels and key-value pairs.

These graphs are formally defined as follows. Let $\mathcal{O}$ be a set of *objects*, $\mathcal{L}$ be a finite set of *labels*, $\mathcal{K}$ be a set of *property keys*, and $\mathcal{N}$ be a set of *values*. Furthermore, we assume that these sets are pairwise disjoint. A property graph is given by the tuple (V, E, η, λ, v), where $V, E \subseteq \mathcal{O}$ are disjoint sets of vertices and edges respectively. The function $\eta : E \to V \times V$ assigns an ordered vertex pair — the source and target vertex — to each edge in the graph. The function $\lambda : V \cup E \to \mathcal{P}(\mathcal{L})$ assigns to every vertex and edge a (possibly empty) set of labels. Finally, the partial function $v : (V \cup E) \times \mathcal{K} \to \mathcal{N}$ assigns a value from $\mathcal{N}$ to properties in $\mathcal{K}$. For each vertex and edge in the graph, the number of assigned properties is finite.

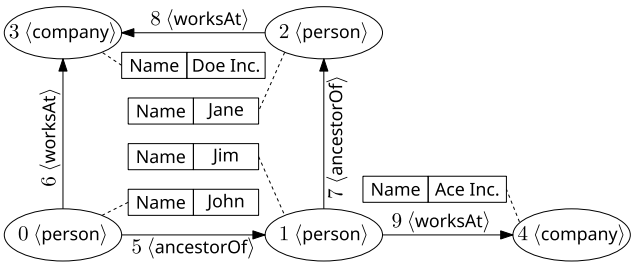


Figure 1: Example property graph, with numbers assigned to the vertices and edges.

An example graph is shown in Figure 1. The labels are wrapped in angle brackets and each vertex and edge is assigned a label. Furthermore, vertices and edge are assigned a number allowing easy referral to specific vertices and/or edges. Properties are shown in tables and the dashed line points to the associated vertex. The formal definition of the example graph would be:

$$
\begin{aligned}
V &= \{0,1,2,3,4\} \\
E &= \{5,6,7,8,9\} \\
\eta(5) &= (0,1) \\
\eta(6) &= (0,3) \\
\eta(7) &= (1,2) \\
\eta(8) &= (2,3) \\
\eta(9) &= (1,4) \\
\lambda(0) &= \{\langle person \rangle\} \\
\lambda(1) &= \{\langle person \rangle\} \\
\lambda(2) &= \{\langle person \rangle\} \\
\lambda(3) &= \{\langle company \rangle\} \\
\lambda(4) &= \{\langle company \rangle\} \\
\lambda(5) &= \{\langle ancestorOf \rangle\} \\
\lambda(6) &= \{\langle worksAt \rangle\} \\
\lambda(7) &= \{\langle ancestorOf \rangle\} \\
\lambda(8) &= \{\langle worksAt \rangle\} \\
\lambda(9) &= \{\langle worksAt \rangle\} \\
v(0, \text{"Name"}) &= \text{"John"} \\
v(1, \text{"Name"}) &= \text{"Jim"} \\
v(2, \text{"Name"}) &= \text{"Jane"} \\
v(3, \text{"Name"}) &= \text{"Doe Inc."} \\
v(4, \text{"Name"}) &= \text{"Ace Inc."}
\end{aligned}
$$

We can execute *Union of Conjunctive Regular Path Queries (UCRPQ)* on property graphs. Formally, such a query consist of a finite non-empty set of *Conjunctive Regular Path Queries (CRPQ)* each with the same number of output variables.A CRPQ consists of three parts. First, a set of vertex variables $\mathcal{V}$. Second, body predicates given by (v_1, v_2, p), where $v_1, v_2 \in \mathcal{V}$ define the source and target vertex of the edge (or path), p defines a predicate over the edge (or path) enforcing the

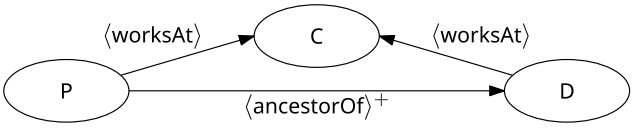


Figure 2: A graph pattern describing people(P) that work at the same company (C) as one of their descendants (D).

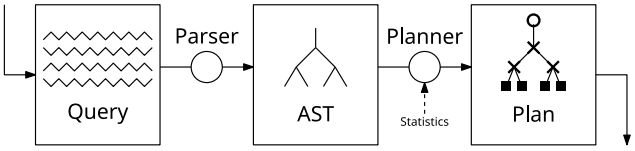


Figure 3: The different representation status of the query before execution [35].

existence thereof and/or placing additional constraints on the labels assigned to the edges. Third, body constraints, which can enforce certain requirements. These requirements can include that elements in the body predicates are assigned certain labels or place restrictions on the properties of vertices and edges.

The results of these type of queries are a subsets of vertices for which there exists an assignment $a : V(G') \rightarrow \mathcal{V}$ (where $V(G')$ are the vertices in the subset) such that all the vertex and edge predicates are fulfilled. For example, we might want to know the set of people that work at the same company as one of their descendants. Figure 2 shows a graph pattern encoding the query. If this example is executed on the example graph previously shown in Figure 1, the only vertex subset matching the pattern is the subset $\{0, 2, 3\}$. The vertices $1, 4$ are not part of the assignment and therefore not part of the result.

2.2 Query pipeline

In a typical database, a query goes through three different stages before the query is executed. These stages are depicted in Figure 3. First the query is parsed and translated into an intermediate representation, for example an *Abstract Syntax Tree (AST)*. In the following stage, the optimal method of execution — which query operators to use — is calculated by the planner using statistics and heuristics, resulting in a physical query plan. The difference between the physical query plan and the previous query representations, is that the query plan encodes how to answer the query, whereas the previous representations only contain the question to be answered. The physical query plan consists of operator primitives which describe operations that need to be performed. For example, merge-join and table-scan. Once the physical plan is created, it is send to the execution engine, which executes it.

2.3 Intermediate result pipelining

Every operator in the physical query plan produces results. The results produced by the last operator are the query results, while the results produced by other operators are intermediate results used as input for other operators.

If an operator must be fully executed before starting with the next one, all results produced by the operator must be buffered until the next operator can use them. Materialising the intermediate results by buffering can be expensive or even impossible due to memory constraints. The complete materialisation of intermediate results can be prevented by sending them to the next operator and processing them as soon as possible, which is called *pipelining*. Pipelining results from one operator to the next ensures that only a fraction of the intermediate results is materialised simultaneously. The downside to pipelining the results is the additional overhead introduced due to the repeated context switching.

The overall cost of pipelining can be reduced by sending results in blocks instead of individually. When results are sent in blocks, the cost of changing which operator is executed can be amortized over more tuples. For example, if 1024 tuples are pipelined together in a single block, then which operator is executed is switched only once every 1024 tuples. This method does, however, result in additional materialisation cost as more results are materialised at once — 1024 compared to only 1.

2.4 Row-wise and column-wise layout

A collection of tuples can be stored row- or column-wise. Storing tuples row-wise means that each tuple is stored in a continuous block of memory and for each tuple, a pointer to the block of memory is kept. Storing tuples column-wise means that each column is stored in a continuous block of memory and for each column a pointer to the block of memory is stored. The cost of removing a tuple from a set of tuples stored row-wise is cheap, as only the pointers have to be moved. The cost of removing a tuple from a set of tuples stored column-wise can be quite expensive because, for each column, possibly all values must be moved. To prevent these expensive deletions, an additional column is stored, namely a selection column. The selection column stores

the indexes of the tuples which have not been removed. When a selection column is used, removing a tuple only requires modification to a single column, namely the selection column, in which deletions are performed similar to row-wise tuple deletion. If an operator does not remove tuples, the selection column can be omitted and the tuples can be access directly. Omitting the selection column can be seen as there being a default identity selection column. Figure 4 depicts two tuple blocks, one using a row-wise and one using a column-wise layout. In both depictions, arrows are used to indicate the values associated with the tuples. The row-wise block contains 4 tuples with pointers to their memory blocks. In order to access the first field of the second tuple, the memory location of the second tuple is retrieved from the index column and then the value can be read from the memory location. The column-wise block also contains 4 tuples, however, the selection column contains gaps in the indexes where tuples have been removed. In order to access the first field of the second tuple, the offset of the second tuple is retrieved from the selection column — which in this case is 3 — and then from the first column, the value located at the offset can be read.

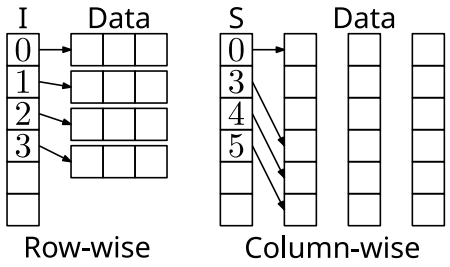


Figure 4: Examples of tuple blocks.

2.5 Transitive closure

One of the core functionalities of modern graph databases involves querying *reachability*.

When querying reachability, we look for all vertex pairs, for which there exists a path in the graph, such that the edge labels along the path satisfy the constrains in the query. Part of the example query shown in Figure 2 contains an example of reachability. In contrast to the left and right edge, the bottom edge is augmented with a "+". The addition of the "+" indicates that the two vertices P and D, do not have to be connected by a single edge, but can be connected by a path of one or more edges.

Reachability can be evaluated by various methods. One of these methods is the *Transitive Closure (TC)*. The transitive closure is defined as follows, let $G = (V, E, \eta, \lambda, v)$ be a property graph and $R \subseteq V \times V$ be a *binary* relation which contains pairs of vertices in G. R is called transitive on a set of vertices V is for all vertices $s, v, t \in V$ if it holds that $(s, v) \in R$ and $(v, t) \in R$ it implies that $(s, t) \in R$. The transitive closure R^+ of the relation R is the smallest binary relation on V that contains R and is transitive. [8].

The relation B is called the *base* relation over which the transitive closure B^+ is computed. Depending on how the relation B is defined over the graph G, B^+ can be used to represent various different reachability patterns. For example, if the relation B contains all vertex pairs which are connected by an edge with the label $ancestorOf$, B^+ would contain all pair of vertices (s, t) such that vertex t can be reached from vertex s by following only edges with the label $ancestorOf$. Using the example graph shown in Figure 1, the base relation B would be $\{(0, 1), (1, 2)\}$ and the relation B^+ would be $\{(0, 1), (1, 2), (0, 2)\}$. Figure 5 shows a visual representation of the base relation B depicted with black edges and the transitive closure B^+ depicted with red edges.

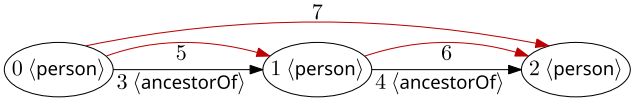


Figure 5: Transitive closure (shown as red edges) over the base relation (shown as black edges) defined by $ancestorOf$ in the example graph. The vertices and edges have been numbered to facilitate referral to vertices and/or edges.

2.6 Worst-case optimal

The join operator is one of the core query operators and forms the base of many query languages, such as SQL.

Therefore, minimising the work performed by a query often involves minimising the work performed by join operators. A lower bound on the minimal amount of work required to evaluate join operators is the output size as the algorithm has to enumerate all the output tuples. Obtaining a bound on the output size of complex join patters is often non-trivial. Furthermore, the obtained bound often cannot be achieved by conventional join operators, such as merge join, or nested loop join.

Consider the following join query, $q(x, y, z) = R_1(x, y), R_2(y, z)$. Here q denotes the query, x, y, z are the variables in the query, and R_1, R_2 are the relations in the query. For this query it is easy to see

that the largest possible output is $|R_1| \cdot |R_2|$, which occurs when all the tuples in the relations pair-wise join.

However, for more complex queries, obtaining a *worst-case optimal* (WCO) bound might not be trivial. Atserias et al. [4] proposed a method based on fractional edge covers for obtaining a WCO bound. The bound obtained by this method is also known as the AGM bound. The AGM bound can be obtained as follows.

Let $vars(q)$ denote all the variables of the query q and $vars(R_j)$ denote all the variables belonging to relation R_j.

A fractional edge cover of a conjunctive query q is a vector u, which assigns a weight u_j to the relation R_j, such that for every variable $x \in vars(q)$, we have that:

$$\sum_{j:x \in vars(R_j)} u_j \geq 1$$

Furthermore, let q be a full conjunctive query. Then, for every fractional edge cover u of q, we have that:

$$|q| \leq \prod_{j=1}^{|R|} |R_j|^{u_j} \tag{2.1}$$

The AGM bound is obtained by computing the cardinality of q (Equation (2.1)) using the minimal fractional edge cover, i.e. a fractional edge cover with minimal weight.

Consider the following triangle query, $q_t(x, y, z) = R(x, y), S(y, z), T(z, x)$. Figure 6 shows the query q_t encoded in a graph pattern. Furthermore, suppose $|R| = |S| = |T| = N$. Then the minimal fractional edge cover assigns a weight of 0.5 to each relation. The AGM bound for q_t would be $|q_t| \leq N^{3/2}$.

Conventional join algorithms perform pairwise joins, which would give us a bound of $\Omega(N^2)$. However, the AGM bound is tighter than the bound obtained by using conventional join algorithms.

In order to achieve the AGM bound, the selectivity of multiple joins can be leveraged. To understand how leveraging selectivity can help lets consider the following query $q_s = R(x), S(x), T(x)$. Figure 7 shows the selectivity of each relation with regards to x. For example, $R(x)$ selects the first million and third million values of x. The selectivity of any pairwise combination of relations results in a selectivity of 1/3, however, the selectivity of all relations combined results in a selectivity of 0. Thus to obtain the minimal selectivity, all three relations should be evaluated simultaneously.

A similar approach can be used to obtain a tighter bound for the join results, namely, performing multiple joins simultaneously. One such join algorithm is the *Leap From Trie Join (LFTJ)* proposed by Veldhuizen [36].

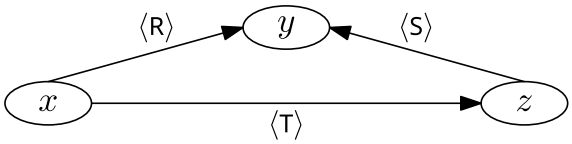


Figure 6: A graph pattern describing a triangle query.

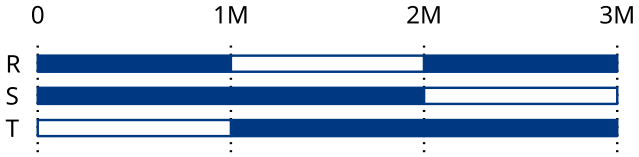


Figure 7: Selectivity of query q_s.

2.7 Parallelism

There are various different techniques to execute queries in parallel. To illustrate the different techniques, two example queries, as shown in Figure 8, are used.

The execution plans of the queries are represented by operator trees, where the nodes represent the operators and the edges represent the dependencies between operators. Two operators are *non-dependant* when neither operator depends on the other.

The simplest technique is to execute multiple sequential queries in parallel. Figure 9 shows the execution of the two examples queries using this technique. This technique can be classified as *inter-query* parallelism. A disadvantage is that this technique cannot be used to speed up a single query.

Another technique is to execute non-dependant query operators in parallel. Consider the example query 1 (Figure 8) consisting of a single join on two relations. The operators reading the relations are non-dependant and can be executed in parallel. Figure 10 shows the execution of the two examples queries using this technique.

The variation which executes *dependant* operators in parallel utilises result pipelining. Without parallelism, in order to process pipelined results, the executing query operator has to be switched. However, if parallelism is used, both query operators can be executed simultaneously, avoiding operator switching. Figure 11 shows the

execution of the two examples queries using this technique. Both techniques can be classified as *intra-query* or *inter-operator* parallelism. A disadvantage is that operators themselves are still executed sequentially, and therefore these techniques cannot be used to speed up queries, which are dominated by a single operator. The second example query is dominated by the Selection operator.

The last technique is to execute the work for a single operator in parallel. Consider the join operator in example query 1 (Figure 8). The join operator will pairwise compare tuples from both relation for possible joins. Rather than comparing these tuples sequentially, the multiple tuples can be compared in parallel. Figure 12 shows the execution of the two examples queries using this technique. This technique can be classified as *intra-operator* parallelism. A disadvantage of this technique is that it cannot be applied to all operators. For example, the Read X and Selection operators cannot be parallelised and the Read Y and Read Z operators have limited parallelism.

As can be seen in the execution examples, each technique has disadvantages and none of them can fully utilise all the parallelism available. Therefore a combination of all techniques is required to fully utilise the parallelism. An example in which all techniques are used is shown in Figure 13. When all techniques are used, all available parallelism is utilised.

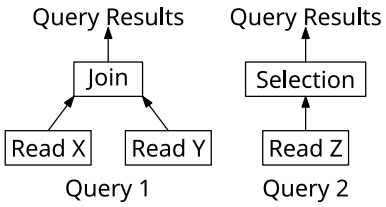


Figure 8: Two example queries.

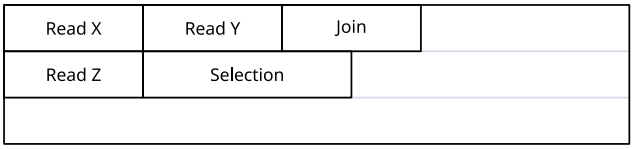


Figure 9: Two example queries executed using inter-query parallelism.

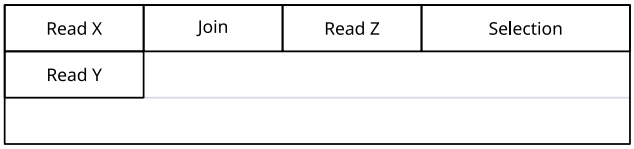


Figure 10: Two example queries executed using intra-query parallelism without pipelining.

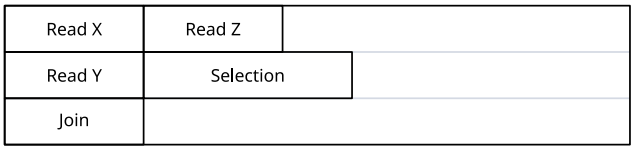


Figure 11: Two example queries executed using intra-query parallelism with pipelining.

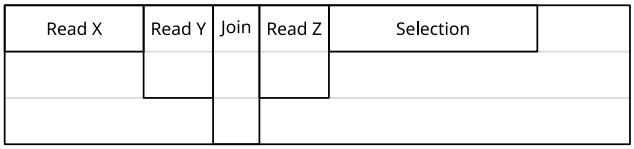


Figure 12: Two example queries executed using intra-query parallelism.

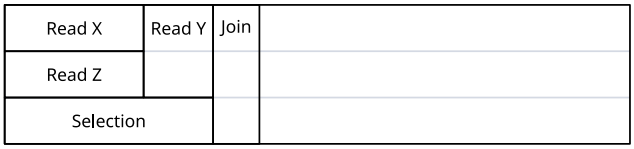


Figure 13: Two example queries executed using all forms of parallelism.

2.8 Quickstep execution model

Over the years, many database systems have been designed to utilise the parallelism available in modern hardware, one of which is Quickstep. In order to take full advantage of the parallelism available, Quickstep utilises several types of parallelism. The three most important of which are, intra-operator parallelism, inter-operator parallelism, and lastly, inter-query parallelism. To enable these different types of parallelism, several techniques are used.

Normally, execution management and the actual execution are interwoven. In contrast, in Quickstep, execution management and the actual execution has been separated and divided into a management thread and several worker threads. The management thread delegates the execution of the work to different worker

threads. Separating management and execution allows work to be divided across multiple threads and therefore utilise more of the available parallelism.

In order to divide the work across multiple execution threads, each query operator has been designed such that it can be divided into multiple sub-tasks. These sub-tasks are independent of one another, and can therefore be delegated to one of the execution threads. Splitting the query operators into multiple sub-tasks enables Quickstep to take advantage of intra-operator parallelism. Furthermore, to enable intra-operator parallelism, intermediate result pipelining is used. And lastly, multiple queries can be executed concurrently, which allows for inter-query parallelism.

3 Approach

In this thesis, we implement parallel query execution in the pre-existing graph database AvantGraph — a single-threaded main-memory graph-database developed at the TU/e [5]. We completely replace the existing execution engine to support parallel query execution.

3.1 Architecture

The design is based on the architecture used by Quickstep, borrowing the general ideas and components. The internal working of the components, however, is redesigned to accommodate the use in graph databases — as Quickstep is designed for relational databases — as well as allowing integration into AvantGraph. The internal architecture resembles that of a typical DBMS engine. The distinguishing aspect is the combination of the purpose designed query operators and scheduler, which allows the execution engine to fully utilise the parallelism available. A high level overview of the architecture of the execution engine is shown in Figure 14. The figure shows the different components and how each component interacts with the other components.

A *Command Line Interface (CLI)* is used, which is responsible for the interaction between the user and the execution engine. The CLI is responsible for sending queries to the execution engine and presenting the result of the queries to the user.

In order to facilitate communication between the different components of the execution engine a *Message bus* is used. The functionality provided by the Quickstep message bus was suitable for the functionality required in AvantGraph. Therefore, the message bus is nearly identical to the message bus in Quickstep. Only small modifications have been made to allow integration into AvantGraph.

Memory management of the (intermediate) query results is handled by the *Storage manager*. The design is based on a block-based architecture. To prevent continuously moving complete blocks of data, each block is assigned a unique id. This unique id is shared between the components, which can use the id to retrieve the block when required.

An execution plan is encoded as a directed graph, also known as a DiGraph, of relational operators. The execution plan is created by the optimizer and then sent to the *scheduler*, which is described in Section 3.2.

The *query operator library* contains implementations of various query operators. Currently, the library has implementations for the following operators: *select*, *project*, *join*, *rename*, and *union*. Furthermore, several data access operators have been implemented, such as, read edges, read vertices, and read properties among others.

The execution engine implements several join algorithms, such as *nested loop join*, *indexed nested loop join*, *merge join*, *hash join*, and *leap frog trie join*. The hash join consist of two operators, a build and probe operator.

Nearly all operators take advantage of block level parallelism. Some data access operators cannot take advantage of block level parallelism due to their sequential or random nature. Another example is the LFTJ operator, which depending on the version, may use a different form of parallelism, the details of which are given in Section 3.4. Operators, which cannot take advantage of block level parallelism, do return block based results and support pipelining of these results.

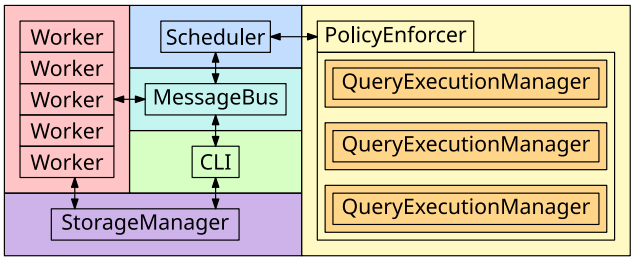


Figure 14: High level architecture overview and component interaction.

3.2 Scheduling and execution

In this section, we describe how the design of the execution engine achieves the three key objectives.

First, separating the control flow from the data flow during query execution allows for greater flexibility and

extensible of the execution engine. In order to achieve this objective, the execution engine separates these flows into different components. The control flow is handled by the scheduler and the data flow is handled by the workers (Section 3.2.1).

Second, to fully utilise the parallelism available in modern processors, the execution engine complements the block-based design with a *WorkUnit* based scheduling model (Section 3.2.2) allowing for high intra-query and intra-operator parallelism.

Finally, to obtain high inter-query parallelism, scheduling policies are used which govern resource sharing (such as CPU and memory) between concurrent queries (Section 3.2.3).

3.2.1 Threading model

The execution engine consists of a single *scheduler* thread, and a collection of *workers* threads.

The scheduler thread uses the query plan to generate and schedule work for the workers. When multiple queries are executed concurrently, the scheduler is responsible for enforcing resource allocation policies across concurrent queries and managing query admittance.

The workers are responsible for executing the query operation tasks that are scheduled. Each worker is a single execution thread, which is pinned to a CPU (or virtual) core. The workers are created when the execution engine starts and are kept alive across query executions, minimising query initialisation cost. The number of workers can vary and can be set (dynamically) when the engine starts.

3.2.2 WorkUnit based scheduler

The scheduler divides the work for a query into a collection of *WorkUnits*. First, we describe the WorkUnit abstraction and provide examples of different WorkUnit types. Next, we describe how WorkUnits are generated by the scheduler for different types of operators in a query plan, including pipelining and memory management during query execution.

The query plan received by the scheduler is represented as a DiGraph, in which each node represents a query operator and each edge represents a dependency between query operators. Figure 16 shows the query plan as a DiGraph for the query shown in Figure 15. For each edge in the DiGraph it is shown whether the edge allows results pipeling or is pipeline breaking.

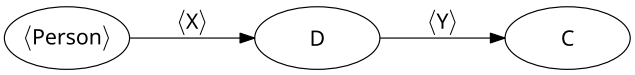


Figure 15: Example of a query.

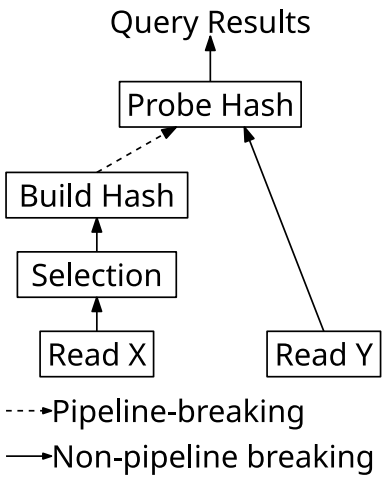


Figure 16: Example of a query plan DiGraph.

WorkUnit

A *WorkUnit* is a unit of intra-operator parallelism for a query operator. Every query operator in the execution engine encodes its work in a collection of WorkUnits, where each WorkUnit contains references to its input and all required parameters.

For example, a *Selection WorkUnit* contains a reference to the input block, a selection predicate, and method for obtaining a block into which the output can be written. A selection operator will generate one WorkUnit for each block in the input relation. Similarly, a *Build Hash WorkUnit* contains a reference to the input block, a collection of build keys, and a reference to a hash table.

WorkUnit generation and execution

The scheduler uses a simple DiGraph traversal algorithm to activate nodes in the graph for processing. The algorithm closely resembles a *Directed Acyclic Graph (DAG)* traversal algorithm, with the addition of a special sub-routine to resolve transitive cycles in the graph. The traversal algorithm starts at the leaf nodes and moves up along the graph. An active node in the DiGraph can generate *schedulable* WorkUnits. Once a node becomes active, the scheduler can fetch these WorkUnits.

In the example DiGraph (shown in Figure 16), the two Read, the Selection, and Build Hash operators are initially active. The Probe Hash is initially inactive

as the blocking dependency has not been met. Once the blocking dependency of the Probe Hash operator is met, the node will become active, and the scheduler will fetch WorkUnits from the operator.

The scheduler will assign the execution of the fetched WorkUnits to available workers, which will in turn execute the WorkUnits. The results produced by these WorkUnits are written to temporary storage blocks, which are used as input for the next operators. For example, the results of the Read X operator is used as input for the Selection operator. After a worker has executed a WorkUnit, the worker sends a completion message to the scheduler.

Cyclic plans

The traversal algorithm used is based on a DAG traversal algorithm.

An operator is finished once all their WorkUnits have been generated and executed. Before an operator can generate all WorkUnits, all input must be received first. In order for the scheduler to know that all input has been send to the operator, the child operator(s) generating the input must first be finished. The requirement of child operator completion is transitive and will cascade down the operator graph.

If there are no cycles in the graph, the transitive check will eventually reach the leaf operators and is trivially satisfied as per definition the leaf operators do not have child operators and thus all possible input has been received.

However, if there are cycles in the graph, the transitive nature of the check will prevent any operator in the cycle from being completed. The reason for this is that an operator in the cycle (via transitivity) depends on itself, therefore the operator must first be completed before it can be completed, which is of course impossible. Because of this, an alternative completion check is used for cyclic operators. Rather than checking each operator individually, all operators in the cycle are checked simultaneously. In order for the cycle to be completed, three conditions must hold. The first condition is that for all operators in the cycle, all child operators not part of the cycle must be completed. Second, no operator in the cycle can produce any new WorkUnits. And third, all produced WorkUnits have been executed. If these conditions hold, all operators in the cycle can be completed, since all operators in the cycle received all their inputs and all WorkUnits have been generated and executed.

Implementation of pipelining

In the example plan DiGraph (Figure 16) the edge for the Selection operator to the Build Hash operator supports pipelining results. As described before, the output of each Selection WorkUnit is written to temporary blocks. Once such a block is full[1], the blocks are send to the Build Hash operator. The Build Hash operator can create WorkUnits based on the received input blocks, thus achieving pipelining.

The design of the scheduler separates the control flow from the data flow. The control flow decisions are encapsulated in the scheduling policy. The scheduling policy can be changed to attain various objectives, such as high performance, or various resource sharing strategies among others. The current scheduler implementation schedules WorkUnits as soon as they are available.

Intermediate result management

During query execution, intermediate results are stored in temporary blocks. When possible, blocks belonging to the same operator are reused and additional results are appended until the blocks becomes full. Reusing blocks is done to minimise the number of sparsely populated blocks, reduce the number of created blocks and with it the allocation overhead. For example, when a join WorkUnit does not generate enough results to fully fill the block, the block is reused by a different join WorkUnit and additional results are appended.

Furthermore, to reduce the memory requirements caused by storing intermediate results, these results and the corresponding blocks are freed as soon as possible. Once all operators depending on the block do not require the block anymore, and all WorkUnits requiring the block as input have been executed, the block is removed. When an operator is finished with the block, varies across operators. For example, a Selection operator generates one WorkUnit for each input block, and thus is finished with the block once the WorkUnit has been generated — note that the generated WorkUnit has not been executed and therefore the block is not yet removed — whereas a Nested Loop Join operator will generate many WorkUnits for each input block and therefore requires the block until all WorkUnits have been generated.

3.2.3 Separation of Policy and Mechanism

The scheduler supports concurrent query execution. The execution of a single query is governed by a *Query*

[1] Or based on a different configurable criteria.

execution manager. Recall that a query is decomposed into query operators which are in turn are decomposed into WorkUnits, which are executed. For each query, all the WorkUnits belonging to the query are stored and organised into a data structure called a *WorkUnit Container*.

During scheduling a *single decision* consists of: selection of a query, selection of a WorkUnit from the WorkUnit container, and dispatch of the selected WorkUnit to a worker.

When multiple queries are executed concurrently, one key aspect of the scheduling decision is selecting a query from the collection of concurrent queries. Various policies for query selection can be used, such as priority based, or expected duration.

In the current implementation, queries are selected using a round robin approach. Once a query has been selected a single WorkUnit is selected. Should a query be selected multiple times, multiple WorkUnits are selected. For example, if there are two queries executing, and four WorkUnits have to be selected, two WorkUnits are selected from each query.

3.3 Transitive closure

The transitive closure operator can be implemented using various methods. Two of these methods will be explained below. The first method uses a technique of repeated joining, which is used in the original Avant-Graph execution engine. The second method uses a graph traversal algorithm, which is used in the parallel execution engine.

Recall the following definitions: the relation B over which the transitive closure B^+ is computed is called the base relation; the relation B^+ is the smallest binary relation which is transitive.

In order to successfully construct the transitive closure B^+, we have to ensure two things. First, we have to ensure that the relation is transitive, i.e. contains all transitive edges. Second, we have to ensure that the relation is minimal, i.e. contains no duplicates.

3.3.1 Transitive closure via repeated joining

The first method for constructing the transitive closure relies on repeated joining to discover all edges and a uniqueness check to remove all duplicates.

The base relation contains all single length edges. In order to obtain all edges of length 2, single length edges can be combined creating edges of length 2. Similarly, edges of length 3 can be obtained by combining edges of length 2 with edges of length 1. In order to obtain edges of length n, edges of length $n-1$ are combined with edges of length 1.

Edges are always combined with edges of length 1. Therefore, the same join operator can be used to combine edges of arbitrary length with edges of length 1. To combine edges, a hash join is used.

To compute the transitive closure, the following steps are taken. First, using the base relation, a hash table is constructed containing the length 1 edges. Next, the base relation is checked for uniqueness and returned as output, because the base relation contains all edges of length 1. The output of the unique check is used as the second input for the hash join. The following steps are performed repeatedly until no more unique results are produced. Lastly, the output of the hash join is checked for uniqueness, returned as output, and used as input for the hash join. The uniqueness check is required to ensure that the returned output is minimal, but also to ensure termination. Should the uniqueness check not be used, a cyclic path can be traversed infinitely many times. The uniqueness check prevents repeated traversal of paths. An example of the query plan associated with this method can be seen in Figure 17.

Consider the transitive closure of the $ancestorOf$ relation shown in Figure 5. The black edges form the base relation and the red edges the transitive closure. In this example, in combination with the base relation, two joins are required to discover all transitive edges. The base relation contains all edges of length 1, the two red edges $5, 6$, which are checked for uniqueness, returned as output, and used as input for the first join. The first join discovers all edges of length 2, namely the edge 7, which is checked for uniqueness, returned as output, and used as input for the second join. The second join does not discover any new edges. As no edges have been discovered, no unique edges have been discovered, indicating that all edges have been discovered and that the transitive closure of the $ancestorOf$ relation has been computed.

This method contains a cycle of different operators, such as hash join, selection unique, and union among others. The cyclic sub-routine is used to resolve operator completeness in the cycle. Because this transitive closure implementation consist of other (basic) operators, parallelisation is achieved by leveraging the parallelism of the base operators.

To illustrate how the execution of the transitive closure can be parallelised, three example execution are shown in Figure 18. The execution plan shown in Figure 18a

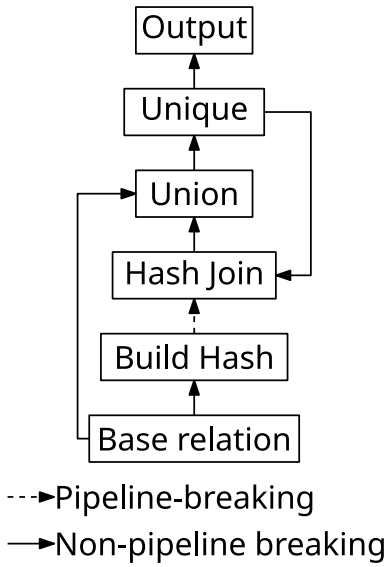


Figure 17: Example of a query plan DiGraph of a transitive closure using repeated joining.

shows how the transitive closure is executed when no parallelism is used. Each step in the execution has been annotated, indicating what the step is used for, and coloured to distinguish steps across different execution plans. Figure 18b and Figure 18c show how the sequential execution plan can be executed in parallel using 2 and 3 processes respectively.

3.3.2 Transitive closure via depth first search

The second method for constructing the transitive closure relies on a graph traversal algorithm to discover all edges and uniqueness checks to prevent duplicate edge discovery. The graph traversal algorithm is a *Depth First Search (DFS)* and in contrast to the first method, does not introduce any cycles in the query plan. Similarly to the first method, a hash table is constructed from the base relation. However, rather than using the hash table as input for the hash join, the hash table is used as an adjacency index.

From the base relation, all unique source vertices are extracted. The source vertices are used as starting points for the DFS. The DFS will, using the adjacency index, traverse all possible paths starting at a vertex. During path traversal, a uniqueness check is used to prevent duplicate traversal. Because all nodes discovered along the traverse path are reachable from the source vertex, edges are created between the source and the discovered nodes, which are returned as output. A DFS is performed from each of the unique source vertices. An example of the query plan associated with this method can be seen in Figure 19. All the source target vertex pairs generated by the DFS are unique[2] and are returned as output of the transitive closure. Once the DFS has been performed for each source vertex, the transitive closure is complete. This implementation of the transitive closure consist of other (base) operator and one DFS operator. Similarly to how parallelism is achieved in the base operators, the DFS operator uses a block-based WorkUnit approach. The DFS WorkUnit consist of a collection of source vertices from which to start the DFS and the hash table to use as adjacency index.

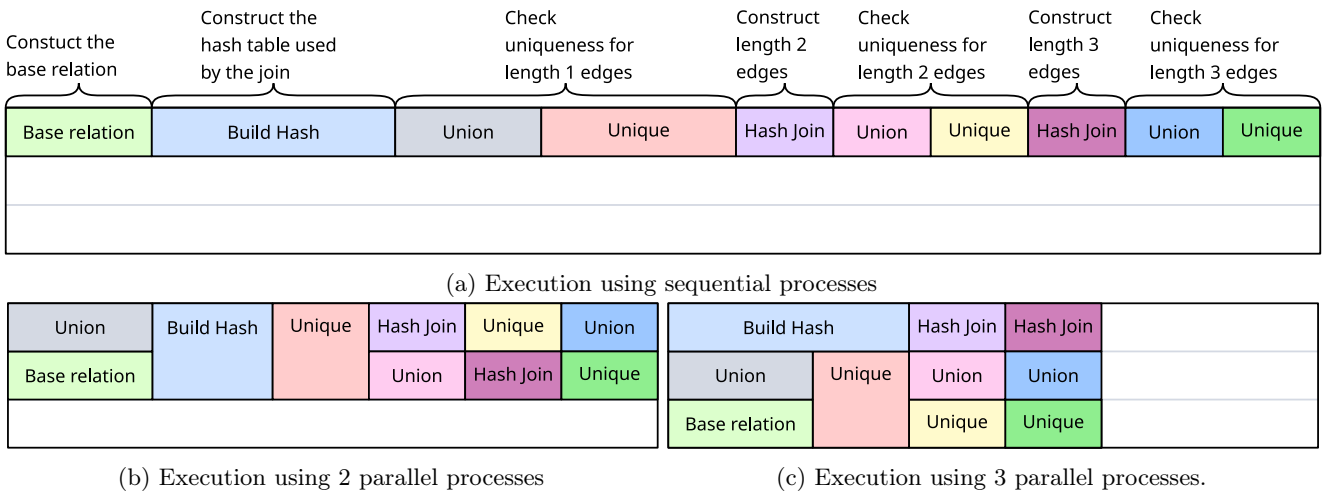
Consider the transitive closure of the $ancestorOf$ relation shown in Figure 5. The black edges form the base relation and the red edges the transitive closure. First, the adjacency index is constructed from the base relation. Next, all unique source vertices are extracted from the base relation, resulting in two source vertices, namely 0 and 1. A DFS is started from each of these source vertices. The DFS starting from the vertex 1 terminates after traversing one edge, namely 4. The output of this DFS is the red edge 6. The DFS starting from vertex 0 terminates after traversing two edges, namely 3 and 4. The output of this DFS are the red edges 5 and 7. Because all searches terminated, the transitive closure of the $ancestorOf$ relation has been computed.

To illustrate how the execution of the DFS transitive closure can be parallelised, three example executions are shown in Figure 20. The execution plan shown in Figure 20a shows how the transitive closure is executed when no parallelism is used. Each step in the execution has been annotated, indicating what the step is used for, and coloured to distinguish steps across different execution plans. Note, there are two DFS steps in the execution plan, one for each source. Because one DFS traverses twice as many edges as the other, the duration is also twice as long. Figure 18b and Figure 18c show how the sequential execution plan can be executed in parallel using 2 and 3 processes respectively.

3.4 Leap frog trie join

There are two versions of the LFTJ operator in AvantGraph. The first version is a conventional LFTJ requiring sorted inputs. The second version allows for one of the input to be in arbitrary order.

[2]Note, the vertex pairs $(1, 2)$ and $(2, 1)$ are not counted as duplicate as they denote different edges or paths in the graph.



(a) Execution using sequential processes

(b) Execution using 2 parallel processes

(c) Execution using 3 parallel processes.

Figure 18: Examples of possible execution plans of the transitive closure using repeated joining.

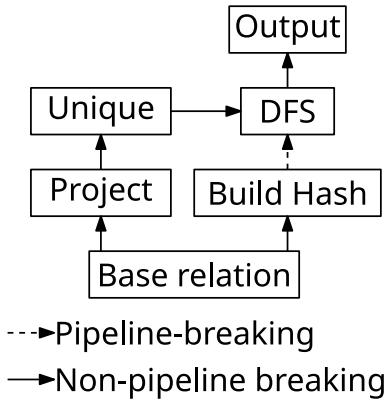


Figure 19: Example of a query plan DiGraph of a transitive closure using DFS.

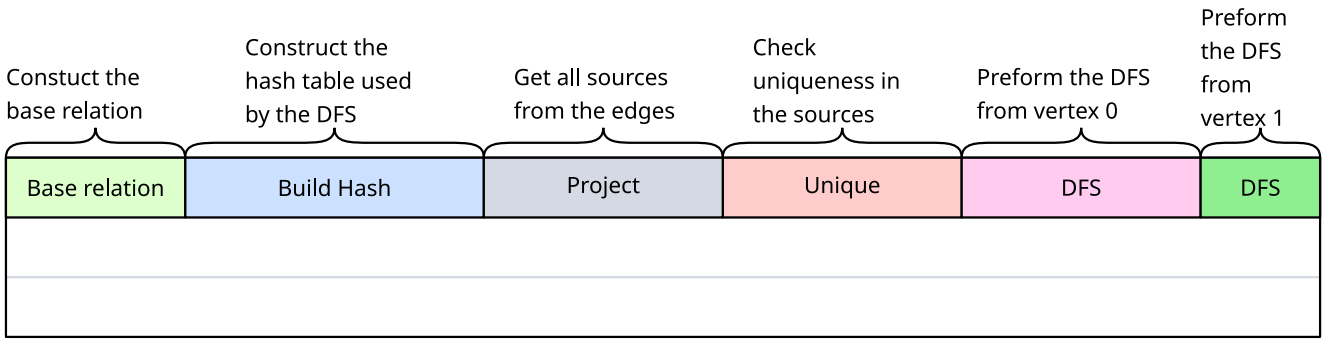
3.4.1 Conventional leap frog trie join

The first version is a conventional LFTJ operator following the ideas presented in [36] and works completely with *variables* and iterators. The conventional LFTJ operator is a leaf operator and does not have any input. Furthermore, the LFTJ is designed to only output the final results and not materialise any intermediate results. For these reasons, the LFTJ cannot trivially be split into block-based WorkUnits. Because there are no obvious splitting points in neither the input nor the output, an alternative splitting point is required. Recall that each WorkUnit should be independent of other WorkUnits.

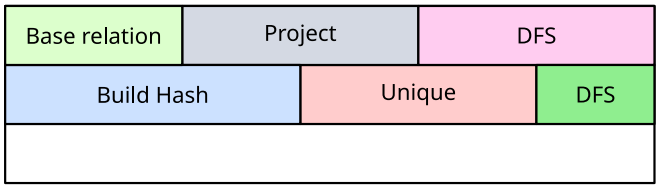
The binding of variables can be used as an alternative splitting point. For example, when there are three variables to bind, the binding of the second and third variables depend on a specific binding of the first variable, but are, however, independent of other bindings of the first variable. Therefore, given certain variable bindings, WorkUnits can be generated to bind the remaining variables. In the current implementation, the binding of the first variable is done by the scheduler, and for each binding found, a WorkUnit is created to find the bindings of the remaining variables.

Consider the LFTJ query shown in Figure 21 of the example graph shown in Figure 1. The manager will find bindings for the first variable, a in our example. Two possible bindings for the variable exist, namely vertices 0 and 1. Two different WorkUnits will be created one for each binding. The WorkUnit for the binding with vertex 0 is able to bind the other variables b, c, d with vertices $1, 2, 3$, respectively. The complete binding results in the output $1, 2, 3, 4$. In contrast, the WorkUnit for the binding with vertex 1 is not able to find a binding for the other variables, resulting in no output.

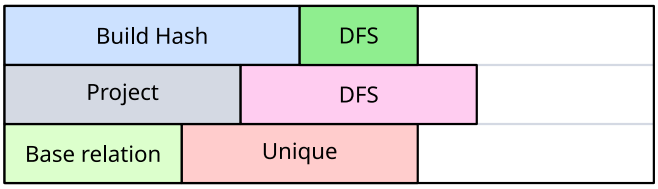
To illustrate how the execution of the LFTJ can be parallelised, two example execution are shown in Figure 22. The execution plan shown in Figure 22a shows how the LFTJ is executed when no parallelism is used. Each step in the execution has been annotated, indicating what the step is used for, and coloured to distinguish steps across different execution plans. Note, there are two LFTJ WorkUnits in the execution plan, one for each initial binding. The duration of the WorkUnit able to bind all variables is longer than the WorkUnit which is not able to bind all variables. Figure 22b shows how the sequential execution plan can be executed in parallel using 2 processes. Because there are only two bindings for the first variable, no example with three processes is given.



(a) Execution using sequential processes



(b) Execution using 2 parallel processes



(c) Execution using 3 parallel processes.

Figure 20: Examples of possible execution plans of the transitive closure using DFS.

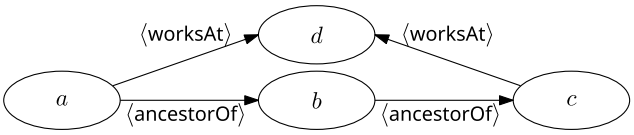
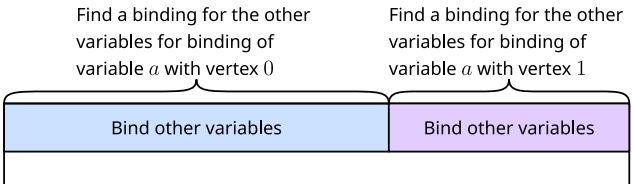
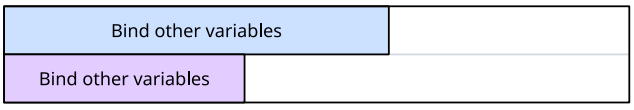


Figure 21: Example of a LFTJ query.



(a) Execution using sequential processes



(b) Execution using 2 parallel processes.

Figure 22: Examples of possible execution plans of the LFTJ.

There are several possible downsides to the current approach. First, if there is only a single binding of the first variable, only a single WorkUnit will be created thus limiting parallelism. Second, if there are many bindings of the first variable which result in few subsequent bindings of the second and third variable, a large number of WorkUnits will be created which output few results, and thus result in a large performance overhead. Therefore, the order of the variables can have an influence on the performance overhead and possible parallelism.

3.4.2 Leap frog trie join with one arbitrary input

The second version is a modified LFTJ, which replaces one of the iterators with an arbitrary input. This modification allows the LFTJ operator to be used on non-base relations in the graph. Similar as before, the modified LFTJ is designed to only output the final results and not materialize any intermediate results. However, due to the modification, this version can easily be split into multiple block-based WorkUnits. WorkUnits can be created for each of the input blocks.

The same example used for the unmodified LFTJ can be used here as the execution and parallelisation are nearly identical. The only difference is how the WorkUnits are generated.

4 Experimental evaluation

In this section, we first describe how the parallel execution engine is evaluated and then we present the results.

4.1 Evaluation criteria

The parallel execution engine will be evaluated on three criteria.

The first criteria is the performance difference (overhead) induced by the separation of the control and data flow. This criteria is measured by limiting the parallel execution engine to a single core and a single worker, essentially turning it into a sequential execution engine. The results are then compared to the original sequential execution engine. The difference in performance gives us an indication of how the two different types of execution engines compare to one another.

The second criteria is the performance difference introduced by distributing the various parts of the execution engine over multiple CPU cores. This criteria is measured by constraining the engine to a single worker but allowing the various parts of the execution engine to be executed on different CPU cores. The results are compared to the results when the engine is constrained to a single worker and only a single CPU core.

The third criteria is to what degree the parallel execution engine can take advantage of the available parallelism in different scenarios. We consider two different scenarios. In the first scenario, queries are executed sequentially, measuring the intra-operator and inter-operator parallelism. In the second scenario, multiple queries are executed concurrently, measuring the intra-operator, inter-operator, and intra-query parallelism. Because multiple machines are used for evaluation, the details of which will be discussed later, the number of concurrently executing queries depends on the machine used for the evaluation. The results of both scenarios are compared to the results of the parallel execution engine running on multiple CPU cores, but limited to a single worker.

4.2 System configuration

The experiments described above are preformed on two virtual machines as each serves a different purpose. The first virtual machine possesses less main memory than the second virtual machine, but can support nearly double the number of workers, giving a better representation of how more workers affect performance. However, even though the second virtual machine can support fewer workers, it possesses 12 times the available main memory, giving a better representation of how larger graphs affect performance.

The first virtual machine is configured with eight non-hyper-threading cores, eight threads, and 16GB of main memory. Because there are no *Non-Uniform Memory Access (NUMA)* nodes on this machine, only the worker threads have to be pinned. Seven workers are created, leaving one core free for either the scheduler, CLI, or other OS threads.

For the first criteria the CLI, scheduler, and worker are all assigned to the same core. For the second criteria the single worker is assigned to a core, and the CLI and scheduler can be executed on any core. For the third criteria the seven workers are all assigned one core each, and the CLI and scheduler can be executed on any core.

The second virtual machine is configured with two CPUs, each with six cores and 12 threads for a total of 24 threads and 192GB of main memory. As there is additional performance overhead associated with cross NUMA-node memory access, care is taken to ensure that all threads are assigned to the same NUMA-node. Furthermore, we do not take advantage of hyper-threading and assign a single thread to each physical core, thus giving us a total of six usable threads. Note, all six of these threads can be used, as there are still 18 other available threads for the OS.

For the first criteria the CLI, scheduler, and worker are all assigned to the same core. For the second criteria the CLI, scheduler, and single worker are all assigned one core each. For the third criteria the CLI, scheduler and four workers are all assigned one core each.

4.3 Dataset

In order to get representative results of the performance, the execution engine is evaluated by querying knowledge graphs, which is one of the applications of graph databases. All experiments are performed using the YAGO2S dataset [29] — a knowledge-graph mined from Wikipedia. Due to the memory capacity of the first virtual machine, two versions of the graph are used, the full graph and a subset consisting of the first ten million tuples, which are called yago2s-full and yago2s-10M respectively.

4.4 Workload

The execution engines are evaluated on seven types of queries. Four of which are pre-existing workloads used to benchmark the performance of the original execution

engine and three workloads are specially generated to test critical parts of the execution engine not tested by the other four workloads.

The four workloads, which will be executed on the first virtual machine with the yago2s-10M dataset, are:

1. **Chain-2**: A collection of three nodes and two edges, connected in such a way that a chain is formed through the nodes.
2. **Chain-3**: A collection of four nodes and three edges, connected in such a way that a chain is formed through the nodes.
3. **Chain-4**: A collection of five nodes and four edges, connected in such a way that a chain is formed through the nodes.
4. **Edge**: All the edges in the graph with a certain label.

The three workloads, which will be executed on the second virtual machine with the yago2s-full dataset, are:

1. **Kleene-1**: A collection of two nodes connected by one or more edges with each edge having the same label.
2. **Kleene-2**: A collection of two nodes connected by any non-zero multiple of two number of edges, such that consecutive edges have alternating labels.
3. **LFTJ-3**: A collection of four nodes and three edges, connected in such a way that a chain is formed through the nodes. The difference between this workload and the 3-chain is how the joins are performed. In the 3-chain conventional joins are used (merge and index nested loop join) whereas here the LFTJ is used.

Each workload consists of 20 queries, expect for Kleene-1, which consists of 10 queries. In order to obtain the 20 queries, a query miner is used, which, given a query pattern, finds all possible queries matching a given pattern. A random selection of 20 is taken, which are the 20 queries used in the workload. Because only ten single label closures exist within yago2s-full, the Kleene-1 workload consists of fewer queries.

The seven workloads cover many different traditional query operators and different versions of these operators, such as multiple join implementations. The Kleene-1 and Kleene-2 workloads cover the transitive closure, and the LFTJ-3 workload covers the WCO and LFTJ operator.

The average results of all queries in a workload are used as the final result of that workload.

4.5 Evaluation method

Even though AvantGraph is a main memory database, the datasets, initially located on disk, must first be loaded into memory. Furthermore, there are various indexes in AvantGraph, which also must be loaded into memory.

Due to the large volume of data and accompanying indexes, certain parts are only loaded into memory when actually needed. For example, only the part of the adjacency index which is required is loaded into memory. This lazy loading technique can significantly reduce the memory requirements but can also result in significant I/O overhead when accessing the data for the first time. Furthermore, the virtual machines are not dedicated systems, some system noise is present and can negatively influence the results. To minimise both the I/O overhead and the system noise, all work-loads are executed three times in quick succession. The best result is used for the comparisons.

4.6 Measurement points

In order to analyse the various aspects of the parallel execution engine, three different data points are collected from each workload. Using these three data points, various other data points are calculated.

The different data points cover various aspects of the execution engine and will be compared to get a holistic overview of the engine. In contrast to the various data points collected and calculated from the parallel execution engine, only a single data point is collected from the sequential execution engine, namely the total execution time.

The data points collected from the parallel execution engine are:

1. **CPU time**: The total duration to execute all WorkUnits, excluding generation and scheduling. This metric represents the time required for the data manipulation (i.e. query operator execution) for the queries.
2. **Wall time**: The real world time spend waiting on query results during query execution. Consists of the time between the start of the first WorkUnit of the first query until the completion of the last WorkUnit of the last query. Periods when no queries are executed are discarded. For example, when queries are executed sequentially, the time

between finishing a query and receiving the next query for execution is discarded.

3. **Number of workers**: The number of available workers in the current configuration.

Using the three collected data points, the additional data points are calculated as follows:

1. **Effective concurrency**: The number of WorkUnits which are, on average, executed concurrently. Calculated by the fraction of CPU time and Wall time.
2. **Available concurrency**: The amount of concurrency available in the current configuration. Equal to the number of workers.
3. **Concurrency utilised**: The percentage of the available concurrency utilised. Calculated by the fraction of effective concurrency and available concurrency.

4.7 Results

4.7.1 Inter-engine performance difference

The results of the first criteria, the inter-engine performance difference, can be seen in Figures 23 and 24. Here and in all following figures, when results are compared to results of a different configuration, the results are scaled to the results of the other configuration. As can be seen in Figure 23, the work performed by the parallel execution engine is comparable to the original execution engine, differing with at most a factor of 1.08. In addition, many workloads see a significant reduction in the work performed, with the lowest requiring only a factor of 0.36 CPU time. Figure 24 shows that, with the exception of the LFTJ workload, similar conclusions can be drawn with regards to Wall time for the remaining workloads, namely the execution engine is on par or reduces the Wall time compared to the original. The LFTJ workload sees an increase in Wall time with a factor of 1.18. The reason for the difference is due to the additional overhead during WorkUnit generation for the LFTJ operator as discussed earlier. All components of the parallel execution engine are executing on the same core, therefore the control flow and data flow are interleaved, i.e. WorkUnit generation and scheduling occurs interleaved with WorkUnit execution. Because, both the CPU and Wall time are known, an estimation of the control flow overhead can be made, which is shown in Figure 25. The LFTJ workload sees a noteworthy scheduling overhead with a factor of 1.14. However, the overhead in the other workloads in less than a factor of 0.01.

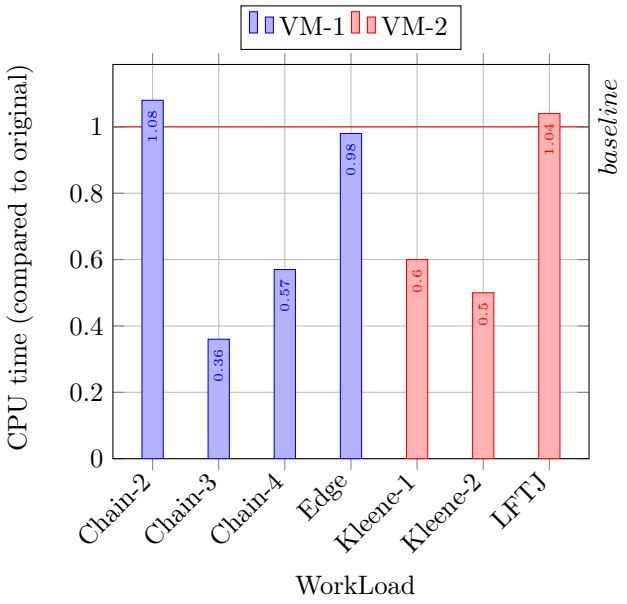


Figure 23: The CPU time of the different workloads of the parallel execution engine using a single CPU core compared to the original execution engine.

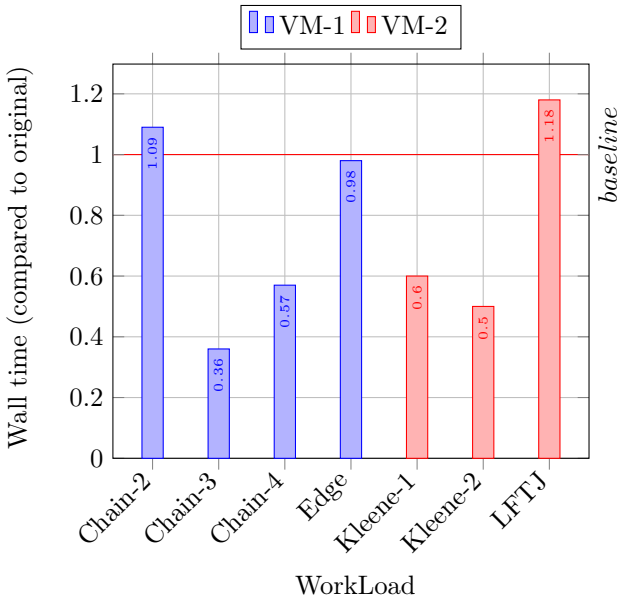


Figure 24: The Wall time of the different workloads of the parallel execution engine using a single CPU core compared to the original execution engine.

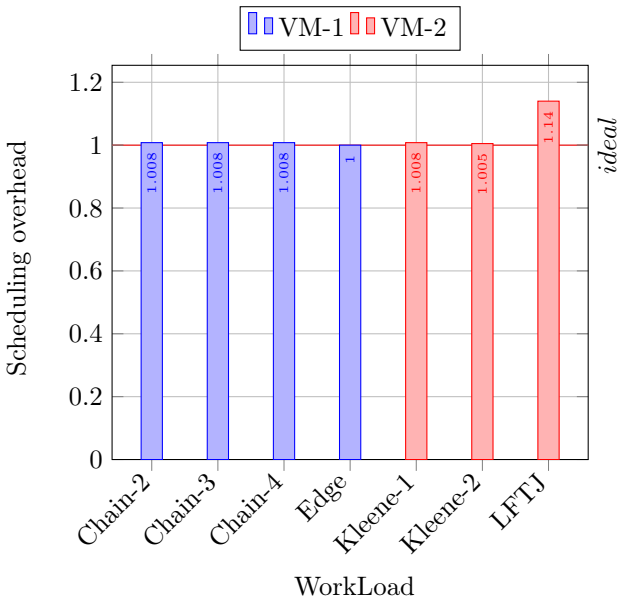


Figure 25: An estimation of the scheduling overhead in the parallel execution engine.

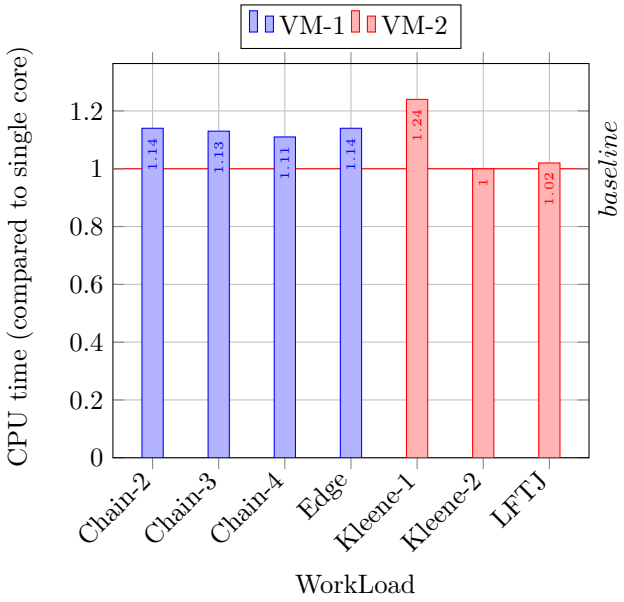


Figure 26: The CPU time of the different workloads of the parallel execution engine with multiple CPU cores but only a single worker compared to the parallel execution engine with a single CPU core.

4.7.2 Multiple threads difference

The results of the second criteria, the performance difference when distributing the execution engine over multiple CPU cores compared to a single core, can be seen in Figures 26 and 27.

The results used as baseline include the scheduling overhead. In contrast, the results produced by distributing the execution engine over multiple CPU cores do not include the scheduling overhead. Because, the scheduling overhead induced by the LFTJ operator is not present in these results, a decrease in Wall time would be expected. Therefore, the seemingly stable CPU and Wall time of the LFTJ are indicative of an increase in Wall time. Thus, all workloads, with the exception of Kleene-2 and LFTJ, see an increase in both CPU and Wall time between 1.11 and 1.24. The Kleene-2 workload does not see an increase in CPU and Wall time. The LFTJ only sees an increase in Wall time, with the CPU time remaining unchanged. A possible explanation for the increase in both CPU and Wall time is related to data locality. Even though AvantGraph is a main memory database and all data is located in memory, there are additional memory layers above main memory, e.g. the different layers of CPU cache. We speculate that by using different CPU cores, some performance benefits related to the CPU caches are lost and therefore results in the additional overhead. However, additional research is need to confirm these speculations.

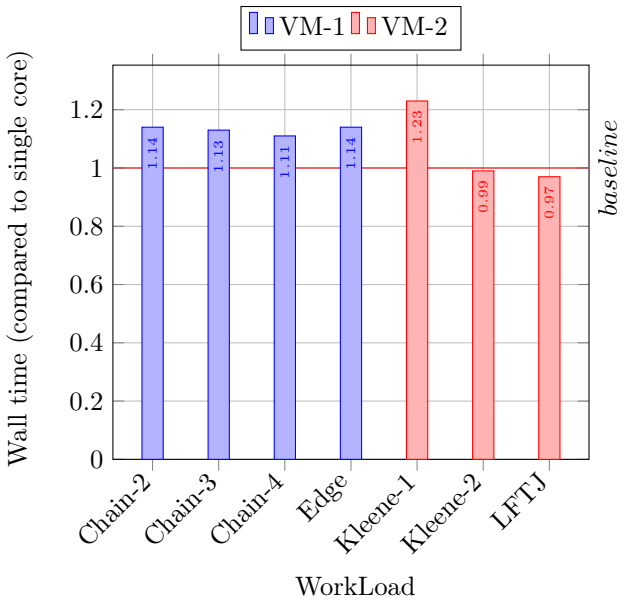


Figure 27: The Wall time of the different workloads of the parallel execution engine with multiple CPU cores but a single worker compared to the parallel execution engine with a single CPU core.

4.7.3 Parallel scalability

The third criteria, to what degree the execution engine can utilise the available parallelism in different scenarios, is evaluated using two different scenarios. The details of which can be found in Section 4.1. The results are grouped per virtual machine due to the difference in the parallelism available. First, the results of the first virtual machine will be presented, followed by the results of the second virtual machine.

First virtual machine

As can be seen in Figure 28, the CPU time of most workloads in both scenarios is comparable to the CPU time of a single worker, differing with a factor ranging from 0.92 to 1.05, or slightly increased with a factor of 1.12 for Chain-4.

Figure 29 shows that the Wall time decreases significantly in all workloads in both scenarios. There is a difference in the amount decreased, with a greater decrease in the parallel query scenario. In the sequential query scenario the decrease is less than the predicted maximum achievable, whereas the decrease in the parallel query scenario achieves (close to) the predicted maximum in the majority of workloads. There can be multiple reasons for not achieving the maximum decrease. The most common are sequential queries/-operators limiting intra-query parallelism, or the fewer results than block size limiting block level parallelism. The Edges workload is an example of fully sequential queries, which can be seen in the minimal decrease in the sequential scenario. However, despite the queries being sequential, a significant decrease in Wall time was still achieved in the parallel query scenario due to leveraging the inter-query parallelism. Similar decreases between the different scenarios in the other workloads can be attributed to leveraging inter-query parallelism.

The amount of available parallelism utilised is shown in Figure 30. As can be seen, none of the workloads in the sequential scenario are able to utilise more than 0.6 of the available parallelism, with the Edge workload not able to utilise any parallelism as the queries are entirely sequential[3]. However, in the second scenario, where multiple queries are executed concurrently, significantly more of the available parallelism is utilised by each workload. The parallelism utilised by the Edge workload more than triples. Furthermore, the Chain-2 workload is able to utilise 0.9 of the available parallelism and lastly, the remaining two workloads are able to utilise over 0.95 of the available parallelism.

[3]The first virtual machine is configured with seven workers, therefore, 0.14 corresponds approximately to single worker.

Second virtual machine

As can be seen in Figure 31, the CPU time in both scenarios is either comparable to the CPU time of a single worker or increases with a factor of up to 1.23.

Figure 32 shows a decrease in Wall time for all workloads in both scenarios. The decrease ranges between a factor of 0.38 up to the predicted minimum of 0.25. In the sequential query scenario, all three workloads achieved significant reductions, with LFTJ even achieving the theoretical minimum Wall time. In the parallel query scenario, both the Kleene-1 and Kleene-2 work see even further reductions in Wall time, while the LFTJ workload sees an increase in Wall time. We suspect that this increase can be attributed to CPU cache misses. With multiple queries and a round robin WorkUnit selection policy, the execution of each individual query increases compared to the sequential scenario. The increase in execution time might result in additional CPU cache misses.

The amount of available parallelism utilised in both scenarios is shown in Figure 33. Recall that the workloads in the sequential scenario on the first virtual machine could nor utilise more than a factor of 0.6 of the available parallelism. All the workloads in the sequential scenario on the second virtual machine are able to utilise more than a factor of 0.6 of the available parallelism, ranging between 0.79 and 0.91. For the parallel scenario, the different workloads are all able to utilise a significant portion of the available parallelism ranging from a factor between 0.8 to 0.96.

The sequential scenario measures intra-query and intra-operator parallelism, which, depending on the workload, might not be enough to fully utilise the available parallelism. The parallel scenario, in addition to intra-query and intra-operator parallelism, also measures inter-query parallelism. This is often enough to fully utilise the available parallelism. The increases in parallelism utilised between the two scenarios can therefore largely be attributed to inter-query parallelism. However, the addition of inter-query parallelism might even be counter productive, as is shown in Figure 30 in the LFTJ workload, which sees a decrease in parallelism utilised.

4.7.4 Speedup over original

We will compare the parallel execution engine with all configurations used in the three criteria, the full details of which are given in Section 4.2, to the original execution engine. The configurations used are: limited to a single CPU core and single worker, multiple CPU

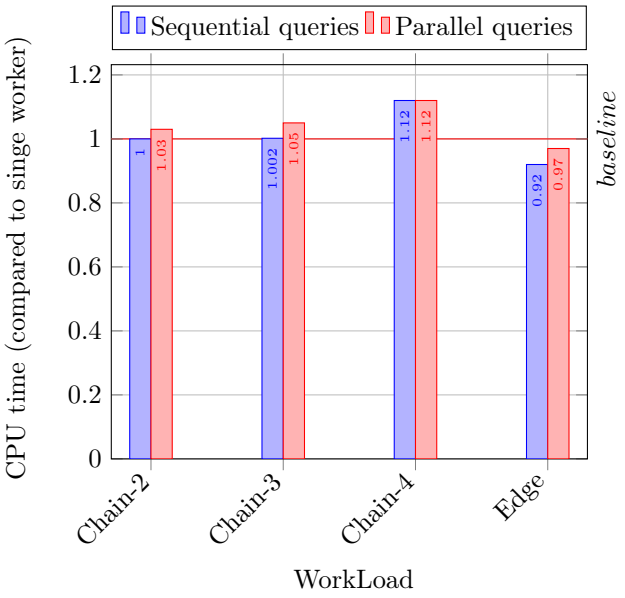


Figure 28: The CPU time of the different workloads of the parallel execution engine with seven workers compared to the parallel execution engine with one worker.

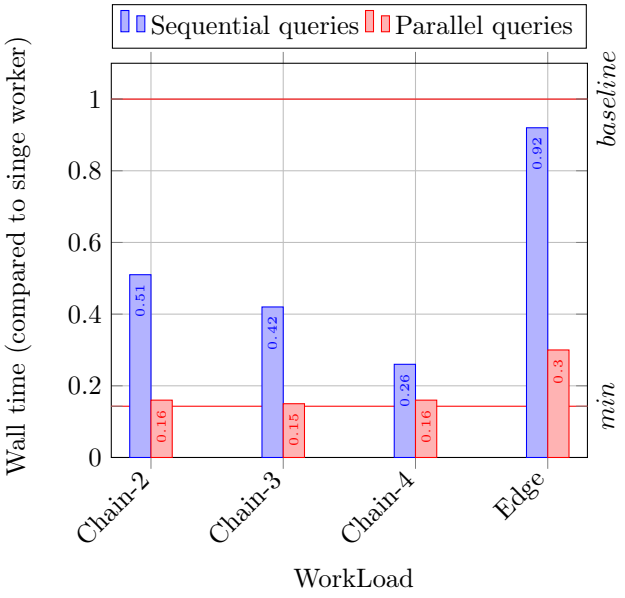


Figure 29: The Wall time of the different workloads of the parallel execution engine with seven workers compared to the parallel execution engine with one worker.

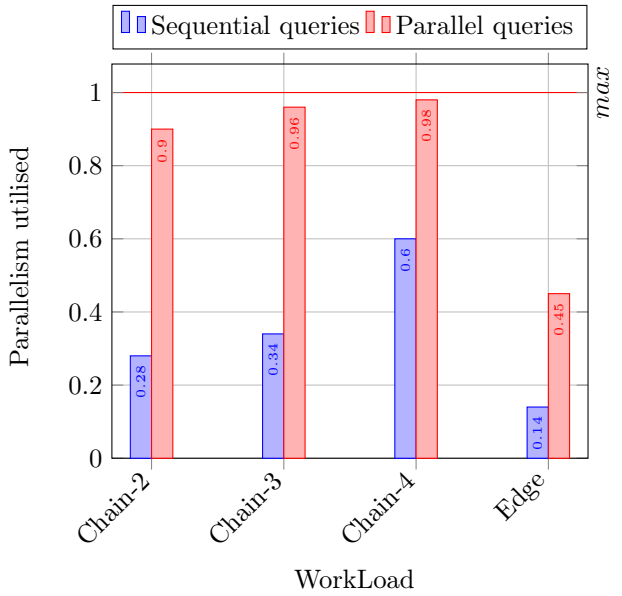


Figure 30: The amount of available parallelism utilised by the parallel execution engine with seven workers.

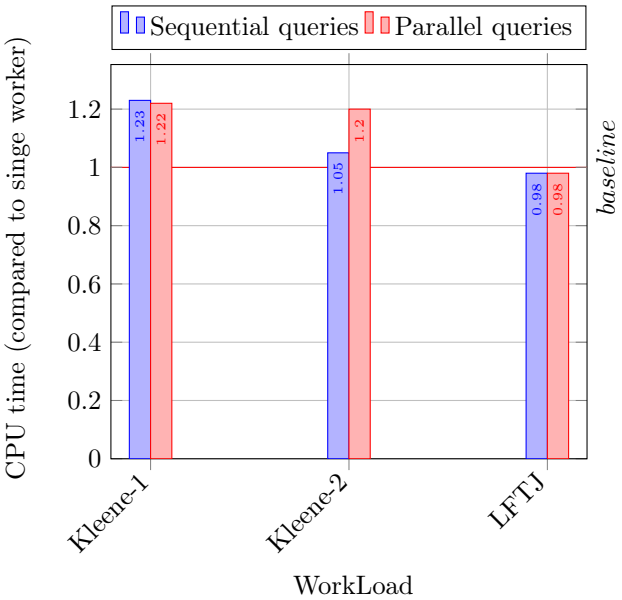


Figure 31: The CPU time of the different workloads of the parallel execution engine with four workers compared to the parallel execution engine with one worker.

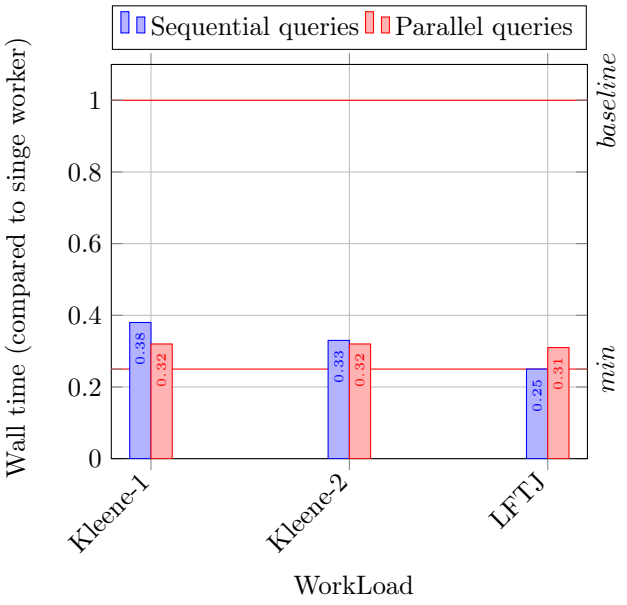


Figure 32: The Wall time of the different workloads of the parallel execution engine with four workers compared to the parallel execution engine with one worker.

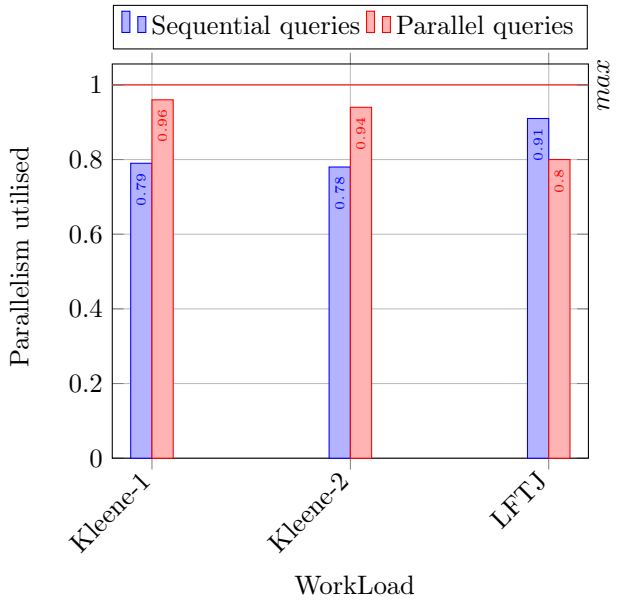


Figure 33: The amount of available parallelism utilised by the parallel execution engine with four workers.

cores and limited to a single worker, multiple cores and workers but sequential query execution, and lastly, multiple cores, multiple workers, and parallel query execution.

Similarly as before, the results are grouped per virtual machine due to the parallelism differences.

First virtual machine

First we present the results of the workloads executed on the first virtual machine compared to the original execution engine. Figure 34 shows the CPU times of the different configurations compared to the original execution engine. As can be seen, for some workloads, the CPU time increases regardless of the configuration. For other workloads, the CPU time is comparable to the original execution engine. Lastly, for some configurations the CPU time always decreases, with the decrease diminishing the more parallelism is exposed by the different configurations

Figure 35 shows the Wall times of the different configurations compared to the original execution engine. As can be seen, all workloads show a significant reduction in Wall time in at least one of the configurations, with some workloads showing a significant reduction in all configurations. The single worker configuration shows the worst reduction in Wall time compared to the other configurations. The best reduction on all workloads is achieved by the parallel queries configuration with a reduction between 0.33 and 0.06.

Second virtual machine

Next, we present the results of the workloads executed on the second virtual machine compared to the original execution engine.

Figure 36 shows the CPU times of the different configurations compared to the original execution engine. As can be seen, two of the workloads always shows a decrease in the CPU time, with the decrease diminishing the more parallelism is exposed by the different configurations. The last workloads shows a negligible to slight increase in CPU time.

Figure 37 show the Wall times of the different configurations compared to the original execution engine. One workload shows a slight increase in Wall time when only one CPU core or worker is used. However, the Wall time decreases in the configurations with multiple workers, ranging between 0.36 to 0.29. The other two workloads show a significant reduction in wall time in all configurations, ranging from 0.74 to 0.16.

Regardless of the virtual machine or workload, the parallel execution engine is able to achieve significant

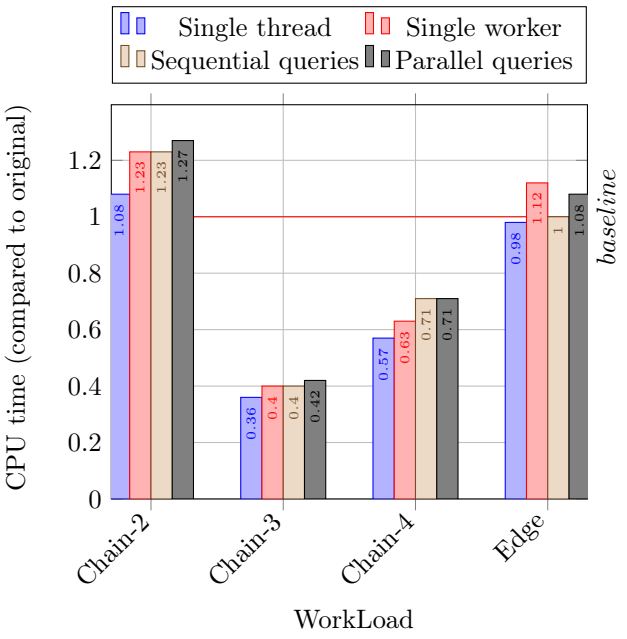


Figure 34: The CPU time of the different workloads of the parallel execution engine in different configurations compared to the original execution engine.

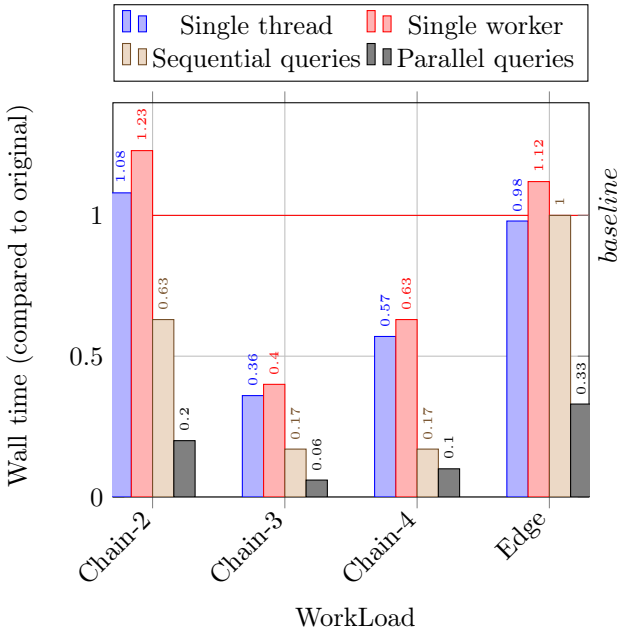


Figure 35: The Wall time of the different workloads of the parallel execution engine in different configurations compared to the original execution engine.

reductions in Wall time for all workloads on at least one configuration and for some workloads on all configurations.

5 Related work

We noted related work throughout the thesis, and highlight some of the overlapping areas of research here.

There are two simultaneous shifts in the database field. With the shift in how data can be represented, graph databases have become a topic of interest and various graph databases have been developed, including [33, 6, 15, 21, 32]. In addition, with the shift in execution bottleneck, there is a tremendous interest in the area of main-memory databases and a number of systems have been developed, including [2, 3, 7, 12, 28, 20].

Many of the graph processing systems are based on a scale-out approach, such as [33, 6, 32, 10, 30, 24, 14, 19, 25]. These systems use numerous severs to store and process the graphs. In contrast to scale-out, several systems have been designed using a scale-up approach, such as [21, 15, 22]. These systems use a single machine to store and process the graphs. These systems use different method to achieve parallel processing.

The scale-out systems can be subdivided into synchronous and asynchronous systems.

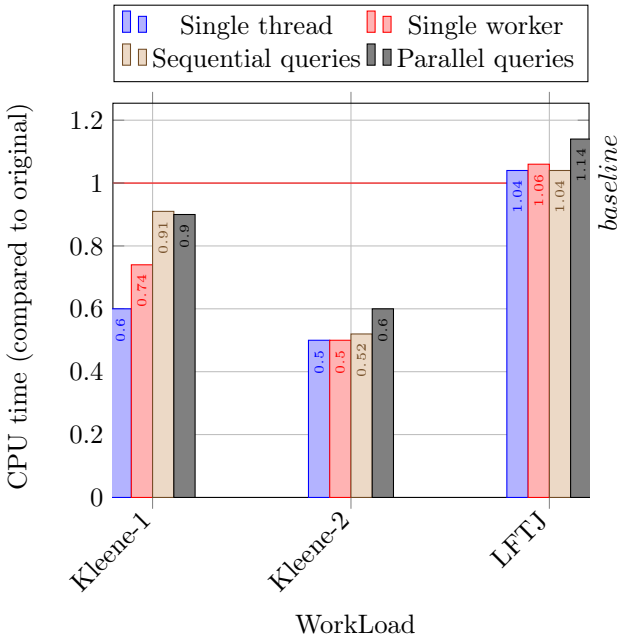


Figure 36: The CPU time of the different workloads of the parallel execution engine in different configurations compared to the original execution engine.

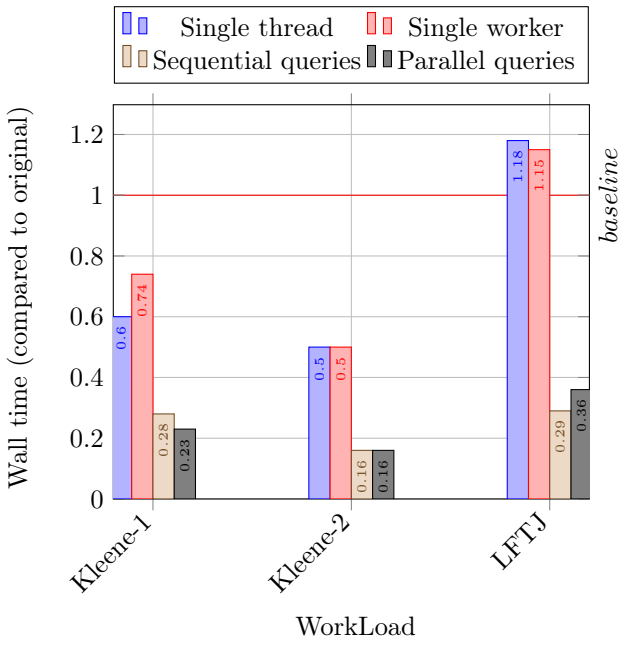


Figure 37: The Wall time of the different workloads of the parallel execution engine in different configurations compared to the original execution engine.

Synchronous scale-out GBase [19] and Pegasus [18] are based on MapReduce and utilise matrix-vector multiplications using compressed matrices. In contrast, Pregel [25] is based on the vertex-centric programming model where a vertex kernel is executed in parallel on each vertex.

Pregel uses the *Bulk-Synchronous Parallel (BPS)* message passing model. In this model, all vertex kernels run concurrently in a sequence of super-steps. In each super-step, each vertex kernel receives all messages from the previous super-step and sends messages to its neighbours in the subsequent super-step. To ensure that all kernels finished processing their messages, a barrier is created between super-steps.

Due to the synchronisation required, the synchronous approaches could suffer from expensive performance penalties because the runtime for each step is dictated by the slowest machine. The imbalance in runtime across machines can be the result of various factors, such as hardware imbalance, network imbalance, an imbalance within the graph itself, or a combination of various factors.

Asynchronous scale-out GraphLab [24], similarly to Pregel, is also based on the vertex-centric programming model. However, vertex kernels are asynchronously executed in parallel on each vertex. Rather than sending messages, each vertex reads and writes data on adjacent vertices and edges using shared-memory.

PowerGraph [13] is similar to GraphLab. However, PowerGraph accounts for the possible imbalance in the graph by partitioning the graph based on highly skewed power-law degree distributions.

Partitioning graphs effectively for all types of graph operators in a distributed environment is an inherently difficult problem [22]. In addition, the user must be able to manage and fine tune a distributed system, which is nontrivial for the ordinary user.

Scale-up GraphChi [22] is a disk-based single machine system using the asynchronous vertex-centric programming model. GraphChi uses a novel concept called *Parallel Sliding Windows (PSW)* for handling large scale graphs. As GraphChi is not a distributed system but a disk-based system, message passing is implemented by updating values to the edges. PSW partitions the vertices into P execution intervals. Each execution interval contains a shard file, which stores all the edges which have their target vertex in the interval. The edges in the shard file are sorted by their source vertex. PSW processes one shard file at a time, preforming the required work for the shard.

McSherry et al. [26] showed that a well designed single-threaded PageRank implementation could match or even significantly outperform several state of the art systems. These systems include, parallel single machine systems or distributed systems with up to 128 cores. Their implementation outperformed GraphChi [22], Stratosphere [10], X-Stream [30], on both data sets tested, and outperformed, GraphLab [24] and GraphX [14], on one of the data sets tested. None of the systems tested was able to outperform their implementation on all data sets.

Therefore, careful consideration should be taken when designing parallel systems to ensure the overhead introduced by parallelisation does not overshadow the performance benefits. Furthermore, Lin [23] articulated the importance of understanding the difference between scale-out and scale-up approaches. Lin suggested to use a scale-up approach where possible, as scale-out approaches often introduce significant overhead.

Therefore, we focused on a scale-up approach and verify that the overhead introduced by the scale-up does not overshadow the performance benefits.

We looked for a suitable scale-up graph system to base our solution on. However, graph processing systems based on a scale-up approach, such as [21], suffered from limitations (such as a lack of parallelism), which TurboGraph [15] aims to address. However, the proposed

solution relies on a general disk-based graph engine, and is therefore not a main-memory database.

Similar limitations exist in other scale-up graph systems.

Therefore we decided to base our solution on a relational database execution engine. Quickstep [28] is a single machine main memory relational database execution engine. Quickstep divides the work for each query into several independent WorkOrders which are executed by execution threads. Even though Quickstep [28] is designed for relational databases, the general design fulfilled many of the requirements for a main memory parallel graph database execution engine. The design of quickstep is, however, incompatible in some critical areas intrinsic to graph query processing, such as cyclic queries and transitive closures. These incompatibilities, however could be resolved by redesigning key areas and extending the design of Quickstep. Therefore, Quickstep was used as the base for the parallel graph execution engine and adapted and extended as required.

Our execution engine employs a distinct block-based architecture for query processing in combination with fast query processing techniques for in memory processing.

The vectorised block execution used in both AvantGraph's original execution engine and our new parallel execution engine has similarities to the work on columnar execution methods, such as [1, 11, 17, 35].

Several operators in AvantGraph use template meta programming which relies on the compiler's optimisations to make automatic use of *Single Instruction, Multiple Data (SIMD)* instructions. This technique is described in [38]. SIMD instructions operator on a vector or tuples and perform operators data-parallel.

The block-based scheduling that is used by the execution engine is similar to the MapReduce style query execution [9]. A key difference between MapReduce and our execution engine is that there is no notion of pipelining in the original MapReduce framework. In contrast, our execution engine does support pipelined parallelism. In addition, our execution engine supports sharing common data structures (e.g. hash table) between multiple tasks belonging to the same operator.

In this thesis we articulate the growing need for the scaling-up approach in graph databases and present a design for a parallel query execution engine for use in graph databases. The execution engine presented is designed for a very high-level of intra-operator parallelism to address this need.

6 Conclusion

The compute and memory densities inside individual servers continues to increase at an astonishing pace. As the parallel capabilities continue to grow, in order to utilise the full compute capabilities of these machines, there is a clear need to fully exploit the parallelism available. In this paper we presented a design for a parallel execution engine, bases on Quickstep, for use in Graph databases and implemented the design. The design emphasizes a scale-up approach and targets in-memory query processing on servers with multiple cores. We present a novel extension to the Quickstep design to resolve cyclic queries, and parallelise two non-trivial query operators, Transitive closure and Leap Frog Trie Join, which are intrinsic to graph processing. The obtained results show that the parallel execution engine is faster than the original execution engine in AvantGraph and is able to utilise nearly all of the parallelism available. Furthermore, the execution engine is able to scale to a large number of workers and handle large graphs and queries effectively. A future iteration of the parallel engine might extend the number of parallelisable operators, add dynamic operators for index creation, augment the query planner to take parallelism into account, or enable sub-plan sharing.

References

[1] D. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, page 671–682, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595934340. doi: 10.1145/1142473.1142548. URL https://doi.org/10.1145/1142473.1142548.

[2] L. Abraham, J. Allen, O. Barykin, V. Borkar, B. Chopra, C. Gerea, D. Merl, J. Metzler, D. Reiss, S. Subramanian, J. L. Wiener, and O. Zed. Scuba: Diving into data at facebook. *Proc. VLDB Endow.*, 6(11):1057–1067, 8 2013. ISSN 2150-8097. doi: 10.14778/2536222.2536231. URL https://doi.org/10.14778/2536222.2536231.

[3] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 1383–1394, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450327589. doi: 10.1145/2723372.2742797. URL https://doi.org/10.1145/2723372.2742797.

[4] A. Atserias, M. Grohe, and D. Marx. Size bounds and query plans for relational joins. In *Proceedings of the 2008 49th Annual IEEE Symposium on Foundations of Computer Science*, FOCS '08, page 739–748, USA, 2008. IEEE Computer Society. ISBN 9780769534367. doi: 10.1109/FOCS.2008.43. URL https://doi.org/10.1109/FOCS.2008.43.

[5] Avantgraph. Avantgraph website. http://avantgraph.io/, 2021. Accessed: 2021-05-27.

[6] L. Barguñó, V. Muntés-Mulero, D. Dominguez-Sal, and P. Valduriez. Parallelgdb: A parallel graph database based on cache specialization. In *Proceedings of the 15th Symposium on International Database Engineering&Applications*, IDEAS '11, page 162–169, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450306270. doi: 10.1145/2076623.2076643. URL https://doi.org/10.1145/2076623.2076643.

[7] P. A. Boncz, M. L. Kersten, and S. Manegold. Breaking the memory wall in monetdb. *Commun. ACM*, 51(12):77–85, 12 2008. ISSN 0001-0782. doi: 10.1145/1409360.1409380. URL https://doi.org/10.1145/1409360.1409380.

[8] A. Bonifati, G. Fletcher, H. Voigt, and N. Yakovets. Querying graphs. *Synthesis Lectures on Data Management*, 10(3):1–184, 2018. doi: 10.2200/S00873ED1V01Y201808DTM051. URL https://doi.org/10.2200/S00873ED1V01Y201808DTM051.

[9] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 1 2008. ISSN 0001-0782. doi: 10.1145/1327452.1327492. URL https://doi.org/10.1145/1327452.1327492.

[10] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl. Spinning fast iterative data flows. *Proc. VLDB Endow.*, 5(11):1268–1279, 7 2012. ISSN 2150-8097. doi: 10.14778/2350229.2350245. URL https://doi.org/10.14778/2350229.2350245.

[11] Z. Feng, E. Lo, B. Kao, and W. Xu. Byteslice: Pushing the envelop of main memory data processing with a new storage layout. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 31–46, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450327589. doi: 10.1145/2723372.2747642. URL https://doi.org/10.1145/2723372.2747642.

[12] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. The sap hana database - an architecture overview. *IEEE Data Eng. Bull.*, 35:28–33, 03 2012.

[13] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, page 17–30, USA, 2012. USENIX Association. ISBN 9781931971966.

[14] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, page 599–613, USA, 2014. USENIX Association. ISBN 9781931971164.

[15] W.-S. Han, S. Lee, K. Park, J.-H. Lee, M.-S. Kim, J. Kim, and H. Yu. Turbograph: A fast parallel graph engine handling billion-scale graphs in a single pc. In *Proceedings of the 19th ACM*

SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '13, page 77–85, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450321747. doi: 10.1145/2487575.2487581. URL https://doi.org/10.1145/2487575.2487581.

[16] C. Have and L. Jensen. Are graph databases ready for bioinformatics? *Bioinformatics (Online)*, 10 2013. ISSN 1367-4811. doi: 10.1093/bioinformatics/btt549.

[17] R. Johnson, V. Raman, R. Sidle, and G. Swart. Row-wise parallel predicate evaluation. *Proc. VLDB Endow.*, 1(1), 8 2008. ISSN 2150-8097. doi: 10.14778/1453856.1453925. URL https://doi.org/10.14778/1453856.1453925.

[18] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *2009 Ninth IEEE International Conference on Data Mining*, pages 229–238. IEEE, 2009.

[19] U. Kang, H. Tong, J. Sun, C.-Y. Lin, and C. Faloutsos. Gbase: A scalable and general graph management system. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, page 1091–1099, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450308137. doi: 10.1145/2020408.2020580. URL https://doi.org/10.1145/2020408.2020580.

[20] A. Kemper and T. Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, page 195–206, USA, 2011. IEEE Computer Society. ISBN 9781424489596. doi: 10.1109/ICDE.2011.5767867. URL https://doi.org/10.1109/ICDE.2011.5767867.

[21] A. Kyrola and C. Guestrin. Graphchi-db: Simple design for a scalable graph database system - on just a PC. *CoRR*, abs/1403.0701, 2014. URL http://arxiv.org/abs/1403.0701.

[22] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, page 31–46, USA, 2012. USENIX Association. ISBN 9781931971966.

[23] J. Lin. Scale up or scale out for graph processing? *IEEE Internet Computing*, 22(3):72–78, 2018. doi: 10.1109/MIC.2018.032501520.

[24] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. Hellerstein. Graphlab: A new framework for parallel machine learning. In *Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence*, UAI'10, page 340–349, Arlington, Virginia, USA, 2010. AUAI Press. ISBN 9780974903965.

[25] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, page 135–146, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450300322. doi: 10.1145/1807167.1807184. URL https://doi.org/10.1145/1807167.1807184.

[26] F. McSherry, M. Isard, and D. G. Murray. Scalability! but at what COST? In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, May 2015. USENIX Association. URL https://www.usenix.org/conference/hotos15/workshop-program/presentation/mcsherry.

[27] neo4j. Concepts: Relational to Graph. https://neo4j.com/developer/graph-db-vs-rdbms/, 2021. Accessed: 2021-06-29.

[28] J. M. Patel, H. Deshmukh, J. Zhu, N. Potti, Z. Zhang, M. Spehlmann, H. Memisoglu, and S. Saurabh. Quickstep: A data platform based on the scaling-up approach. *Proc. VLDB Endow.*, 11(6):663–676, Feb. 2018. ISSN 2150-8097. doi: 10.14778/3199517.3199518. URL https://doi.org/10.14778/3199517.3199518.

[29] Y. Project. Yago2s. https://yago-knowledge.org/downloads/yago-2s, 2021. Accessed: 2021-06-22.

[30] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 472–488, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450323888. doi: 10.1145/2517349.2522740. URL https://doi.org/10.1145/2517349.2522740.

[31] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts, Sixth Edition.*

McGraw-Hill Book Company, 2011. ISBN 978-0-07-352332-3. URL https://www.db-book.com/db6/index.html.

[32] S. R. Spillane, D. Bokser, D. Kemp, J.-H. Hwang, J. Birnbaum, A. Labouseur, P. W. Olsen, J. Vijayan, and J.-W. Yoon. A demonstration of the g* graph database system. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, ICDE '13, page 1356–1359, USA, 2013. IEEE Computer Society. ISBN 9781467349093. doi: 10.1109/ICDE.2013.6544943. URL https://doi.org/10.1109/ICDE.2013.6544943.

[33] TigerGraph. TigerGraph. https://www.tigergraph.com/, 2021. Accessed: 2021-01-18.

[34] D. Truong, Q. Truong, and T. Dkaki. Graph Methods for Social Network Analysis. In P. Vinh and L. Barolli, editors, *Nature of Computation and Communication*, pages 276–286, Cham, 2016. Springer International Publishing. ISBN 978-3-319-46909-6.

[35] A. A. G. van de Wall. Fully compiled execution of conjunctive graph queries. Master's thesis, Eindhoven University of Technology, 5612 AZ Eindhoven, 03 2020.

[36] T. L. Veldhuizen. Leapfrog triejoin: a worst-case optimal join algorithm, 2013.

[37] B. Yoon, S. Kim, and S. Kim. Use of graph database for the integration of heterogeneous biological data. *Genomics & Informatics*, 15:19 – 27, 03 2017. doi: 10.5808/GI.2017.15.1.19.

[38] J. Zhou and K. A. Ross. Implementing database operations using simd instructions. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, page 145–156, New York, NY, USA, 2002. Association for Computing Machinery. ISBN 1581134975. doi: 10.1145/564691.564709. URL https://doi.org/10.1145/564691.564709.