FACULTY OF INFORMATION TECHNOLOGY CTU IN PRAGUE

Assignment of master's thesis

Title:	Design and implementation of parallel processing of data flow in the Manta project
Student:	Bc. Tomáš Polačok
Supervisor:	Ing. Michal Valenta, Ph.D.
Study program:	Informatics
Branch / specialization:	Knowledge Engineering
Department:	Department of Applied Mathematics
Validity:	until the end of summer semester 2021/2022

Instructions

The Manta project will soon migrate to a new architecture as well as to a new graph database. This will require a reimplementation of current processes and introduces a possibility for optimization of the present data processing algorithms.

1. Get familiar with the Manta project, especially with the persistent data flow storage module in the graph database and its usage in Manta algorithms.
2. Investigate which operations and algorithms in the Manta project would be the right candidates for optimization.
3. Select suitable candidates for parallel processing.
4. Design optimized and parallel processing of the operations from the second and the third step with the focus on preserving the same results that would be yielded with the original approach.
5. Implement a prototype and carry out profound testing, including performance testing, to verify the positive performance effects of your proposed enhancements.

Electronically approved by Ing. Karel Klouda, Ph.D. on 9 September 2020 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Design and implementation of parallel processing of data flow in the Manta project

Bc. Tomáš Polačok

Department of Applied Mathematics
Supervisor: Ing. Michal Valenta, Ph.D.

May 5, 2021

Acknowledgements

I would like to express my gratitude to my supervisor Ing. Michal Valenta, Ph.D., and to all who helped and supported me throughout my work.

Declaration

I hereby declare that I have authored this thesis independently, and that all sources used are declared in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis (work) is subject to the rights and obligations arising from Act No. 121/2000 Coll., on Copyright and Rights Related to Copyright and on Amendments to Certain Laws (the Copyright Act), as amended, (hereinafter as the "Copyright Act"), in particular § 35, and § 60 of the Copyright Act governing the school work.

With respect to the computer programs that are part of my thesis (work) and with respect to all documentation related to the computer programs ("software"), in accordance with Article 2373 of the Act No. 89/2012 Coll., the Civil Code, I hereby grant a nonexclusive and irrevocable authorisation (license) to use this software, to any and all persons that wish to use the software. Such persons are entitled to use the software in any way without any limitations (including use for-profit purposes). This license is not limited in terms of time, location and quantity, is granted free of charge, and also covers the right to alter or modify the software, combine it with another work, and/or include the software in a collective work.

In Prague on May 5, 2021 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Polačok, Tomáš. *Design and implementation of parallel processing of data flow in the Manta project.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021.

Abstrakt

Táto práca sa zameriava na migráciu imperatívneho prístupu grafových algoritmov z grafovej databázy Titan do procedurálneho prístupu grafovej databázy Neo4j pomocou dotazovacieho jazyka Cypher. Cieľom je tiež optimalizovať tieto algoritmy a zaviesť paralelizmus, čo môže byť celkom náročná úloha z dôvodu uzamykania databázy, ako aj ďalších obmedzení pochádzajúcich z paralelizovaných algoritmov.

Klíčová slova grafová databáza, optimalizácia, paralelizácia, Neo4j, Titan

Abstract

This work focuses on the migration of the graph algorithms from the Titan graph database's imperative approach into the Neo4j graph database's procedural approach using the Cypher query language. The goal is also to optimize these algorithms and introduce parallelism, which can be a quite challenging task due to database locking, as well as the other constraints originating from the parallelized algorithms.

Keywords graph database, optimization, parallelization, Neo4j, Titan

Contents

List of Figures

List of Listings

Introduction

Modern systems usually work with a large amount of data and as the data is rapidly growing in volume and variance, the complexity of maintenance and change increases with it. This complexity further grows as the data usually originates from various sources and is interconnected in a way that change of one item typically reflects into cascade changes of many other items across the different sources.

MANTA Flow application helps with the maintenance problem as it can analyze the connections between the data of various sources and create unified data lineage. Using this lineage, customers can see how each item affects the other items. For example, in a banking database system, one would be able to see how the column amount of a table representing transactions is connected to the balance column of a table representing accounts as the change would be reflected in some database script which is analyzed by MANTA Flow.

The analyzed data comes from various sources and can be substantially large requiring a lot of processing time. On top of that, each stored element has a certain revision (version), which determines its validity.

After the analysis of the input data is done, it needs to be merged into the database. The database, in this case, is the bottleneck as the analysis can not only be done in parallel within a single technology but also parallel processing of different technologies is possible, as mostly the order of technologies processed is not set. There are cases, in which certain processing order is required, but those are the minority. After the initial analysis and insertion of analyzed data into the database, various post-processing algorithms, which might also modify the database, are invoked.

The main goal of this thesis is to choose the right candidate for optimization from within the graph algorithms used in the Manta Flow project. As the current database algorithms run in serial, the focus lays in finding the way to be able to run them in parallel, which could greatly increase the performance. Currently, the main storage technology used in the Manta is the Titan graph database, but it will be soon migrated to a new graph database. This the-

sis also deals with the reimplementation of the optimized versions of selected algorithms within the new graph technology.

In the first chapter, the MANTA Flow project, the current implementation of the chosen algorithm, as well as graph database specifications are described. The next chapter contains an analysis of the possible optimization steps along with the migration possibilities, followed by the chapter containing an implementation of the previously analyzed parts. The last chapter contains performance evaluation to determine the increase in performance in comparison to the current implementation.

Chapter 1

Background

In the first chapter, the MANTA Flow project as a whole is described. The focus of this chapter is to explain how Manta currently works, its architecture, mainly the database storage system, along the model used to represent internal structures.

Following the initial explanation, merger, one of the important database algorithms used within MANTA is introduced. Optimizing this algorithm is the main focus of this work and will be discussed in the following chapters.

1.1 MANTA

MANTA is a data lineage platform that automatically scans your data environment to build a powerful map of all data flows and deliver it through a native UI and other channels to both technical and non-technical users. With MANTA, everyone gets full visibility and control of their data pipeline. [3]

It is used to extract and analyze various type of technologies, where users can visualize the result of the analysis, which provides them with better overview of how the data in their systems is interconnected, therefore supporting better data quality and control.

1.1.1 Supported technologies

The technologies MANTA supports can be divided into 5 categories [4]. Each category has specific traits, which separates them, but there is one thing that they all have in common – metadata, which can be extracted and used to create dataflows. These technologies are grouped as follows:

- **Modeling tools** This group represents modeling tools used to build conceptual and logical models which provide a higher-level representation of the data used within a company. MANTA can be used to interpolate the dataflows extracted from physical data into these higher-level models.

- **Data integration tools** Data integration involves combining data residing in different sources and providing users with a unified view of them. The most widely used term for representing this group is ETL. All of the ETL tools are used to extract and process the data, so it can be loaded into another system.

- **Programming languages** Programming languages like C# or Java contain methods through which certain objects are passed, which means that data flows through them. This data can be extracted and used to provide a more detailed flow of the data.

- **Databases** Databases are the most frequent target of the data analysis as this is the technology that contains the most data and is used in almost every company.

- **Reporting & Analysis** These tools often work together with database and ETL tools to further aggregate and analyze the data to create various reports.

1.1.2 Architecture

The currently used architecture model of the MANTA application is the client-server model. This model works in a way, that the application is distributed into two separate modules which communicate together. Each of these modules contains several components.

- **MANTA Flow CLI** The client part of the model is a Java command-line application, which contains certain scenarios to extract and analyze the metadata (along with several supporting scenarios to create a new version, clean the repository, etc.).

- **MANTA Flow Server** The server part of the model is a Java application deployed on the Tomcat application server, which handles requests sent from the client. The MANTA Flow Server contains an embedded graph database Titan, which is persistent storage serving as a repository of the extracted metadata. The model stored on the database can be visualized by the viewer module, which is the last bigger component of the MANTA Flow Server.

Another important segment of the application is the **Admin UI**, which runs on a separate server than the MANTA Flow Server. It allows users to manage configuration and handle the updating of both client and server parts of the main application. The other use of this application is to view logs from all run scenarios in a sophisticated user-friendly way.

1.2 Data Model

This section contains a detailed description of how the data lineage within MANTA is represented. There are two main types of objects – vertices to identify various objects and edges used to connect certain vertices. Both object types are divided into several subtypes representing a specific part of the hierarchy or other supporting structures. Currently, there are 9 types of vertices and 10 types of edges used in MANTA.

1.2.1 Vertices

Most of the vertex types are used to represent structural objects retrieved from the metadata analysis, but some vertices are used for versioning purposes. Each of the vertices includes several properties that describe it or serve for indexing purposes for faster lookups.

There are three special vertices, which serve as control structures. Only one instance of each controlling structure can exist at any time. Each of these special vertices contains boolean property indicating the type of the root and is used solely for indexing. These vertices are used to separate the database into three distinct parts and serve as the roots of the respective subtrees created by connecting other vertices to them. All instances of other vertex types contain *vertexType* property, which represents the type of the vertex. The separation of the types is as follows:

- **source root** This vertex is the root of the source node subtree. The depth of the subtree of this vertex is only one as it only contains source nodes directly connected to the stated root. The indexing boolean property it contains is named *sourceRoot*.

- **source node** As mentioned previously, these vertices can only be connected to the source root. Each source node vertex contains *local name* property, which points to the source file location. Following properties are node *identifier*, which is a uniquely generated string. *Hash* property is a hash created from the source file contents. The last two properties are the *technology* and *connection* which are references to the technologies from which the source files were extracted.

- **revision root** Revision root serves as the root of the versioning subtree. This subtree contains the history of all versions represented by revision nodes. While the other root vertices only serve as control structures, in addition to indexing property *revisionRoot*, revision root also contains two other properties – *latestCommittedRevision* and *latestUncommittedRevision* which allow easier retrieval of revision numbers which are often used.

- **revision node** This vertex represent specific version. Revision node contains 5 properties – *revisionNumber*, which represents given version, *committed* flag denoting whether the revision has been committed, *commitTime*, *previousRevisionNumber*, and *nextRevisionNumber* for easier information retrieval. Each of these vertices is connected to revision root vertex.

- **super root** The super root is the root of the subtree containing all extracted metadata from the source files. The indexing boolean property it contains is named *superRoot*.

- **layer** Layer vertex is connected to vertex and serves to determine resource's layer affiliation. It contains additional *layerName* and *layerType* properties.

  The purpose of this vertex is to differentiate the logical level of various resources and underlying metadata. For example, there can be conceptual, logical, and physical levels, where each portrays the data from a different perspective.

- **resource** The resource is the highest level vertex connected directly to the super root vertex. It represents certain technology (Oracle) or subtype of the technology (Oracle DDL). Each resource vertex has to be connected to a specific layer vertex and contains three properties – *resourceName*, *resourceType* and *resourceDesc*. Each resource vertex is connected to super root vertex and exactly one layer vertex.

- **node** This is the most common vertex which can be linked to resource, node (flow-wise or parent-wise), or attribute vertices. Contains *nodeName*, *nodeType* properties. An example of this node vertex is Oracle's table or column. Each node belongs to a certain resource, either connected directly or indirectly through its parent nodes.

- **attribute** Each node can be linked to arbitrary amount of attribute vertices. Each attribute vertex contains *attributeKey* and *attributeValue* properties.

  These attributes are not an immediate part of node vertex due to versioning reasons (see subsection 1.2.4). The properties of the nodes extracted from the metadata might get slightly changed, so the new revision needs to reflect that.

1.2.2 Edges

Each of the vertices can be connected to other vertices by edges. There are several types of edges, where each type can only connect specific types of vertices and is of a certain direction.

Every edge contains *tranStart* and *tranEnd* properties which serve as revision boundaries. For any type of vertex (save the root vertices), there has to exist exactly one control edge which dictates its revision using the mentioned properties.

- **hasSource** This type connects the source root vertex and the underlying source node vertices. There can exist only hasSource edge between a specific source node and source root vertices as it serves as the control edge of the source node vertices.
- **hasRevision** This type connects revision root vertex and underlying revision vertices. There can exist only a hasRevision edge between specific revision and revision root vertices as it serves as the control edge of the revision node vertices.
- **hasResource** This edge connects super root vertex to every resource vertex but also connects node vertices to specific resource vertices to indicate their resource. If a node vertex has no direct connection to resource vertex, its resource is determined by the parent's resource recursively. The edge is incoming to super root or resource respectively. This edge is the control edge for resource vertices and can be the control edge for node vertices in the case they are not connected to other nodes by the hasParent edge.
- **inLayer** This type is used to connect a resource vertex to its layer vertex, where it is incoming to the layer vertex. The edge is a control edge for the layer.
- **hasParent** Used to determine parenthood of the nodes, hasParent edge also contains one additional *childName* property, containing the name of the child of a given relationship and is used for faster lookup. The edge is incoming to the parent node vertex. Each node can have an arbitrary number of child nodes, essentially forming a subtree. One given node vertex can have up to exactly one parent node vertex. The edge always serves as a control edge for the nodes as there are cases when node vertex has both hasParent and hasResource edges.
- **hasAttribute** HasAttribute edge connects node vertex to its attribute vertices and is used solely for versioning purposes. The edge is incoming to the attribute vertex. As it is the only edge connecting the attribute vertex it also serves as its control edge.
- **directFlow** This type of edge is used to represent the direct data flow of the node vertices as extracted from the metadata. It can only connect the node vertices and it contains an additional *targetId* property, which represents the identifier of the targeted node of a given flow and is used

for faster lookups, and a *interpolated* property indicating whether the edge was created by the interpolation[1].

- **filterFlow** As the previous type, this type is used to represent the data flows, but in this case, the conditional data flows. For example, if the source file contains the `IF` statement, filter flow is extracted. This type also contains the same properties as the directFlow and can only be used to connect the node vertices.

- **mapsTo** Represents mapping of a node vertex belonging to one layer to a node vertex belonging to another. Same properties and vertex requirements as on the directFlow edge are applied.

- **perspective** This type of edge is used to map node vertex belonging to the physical layer so that it would appear as a child of the node (belonging to a different layer) it is mapped to. When the node vertex is mapped to another node vertex by perspective, its entire children hierarchy is mapped to the perspective as well. Same properties and vertex requirements as on the directFlow edge are applied.

1.2.3 Hierarchy

As mentioned in the previous subsection, there are 3 special types of vertices that serve as roots for the underlying subtrees. The source root and the resource root subtrees are just subtrees of depth one containing only elements of one vertex type (see figures 1.1 and 1.2 depicting respective subtree's hierarchy).

The more complex subtree is the one created under the super root vertex. The depth of this subtree is only bound by the extracted technology's restrictions and it contains various types of the vertices (see figures 1.3 and 1.4 depicting super root subtree hierarchy).

1.2.4 Versioning

Each vertex and edge in the MANTA hierarchy is versioned allowing users to view dataflows of the previous analyses. The specific versions are called revisions in MANTA terminology. Each revision is represented by the revision node vertex connected to the revision root vertex. This temporality was implemented thanks to the work of Petr Holeček [5].

Initially, when creating a new revision all the data had to be processed from scratch and then merged into the database which was not very effective as the amount of data changed was usually not that significant. Thanks to the work of Jan Sýkora [6], incremental updates were introduced allowing for the

[1] Interpolation is used to derive data flow from one layer to another.

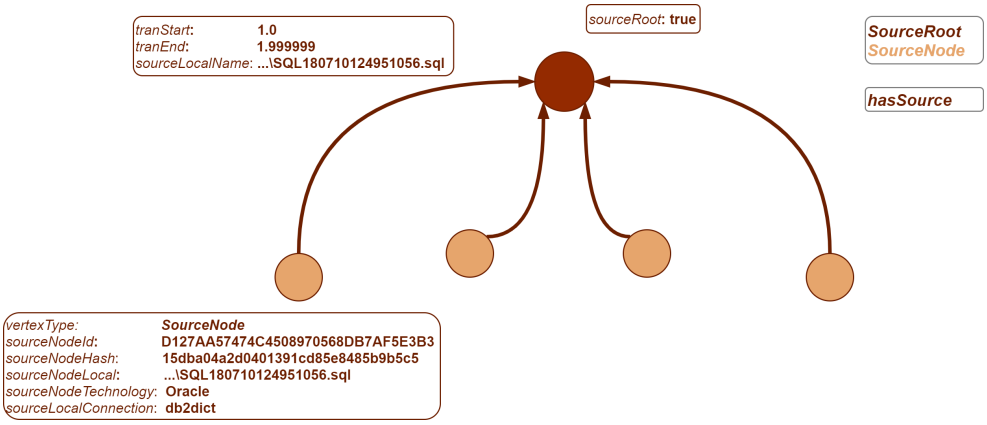


Figure 1.1: Source root subtree vertex hierarchy

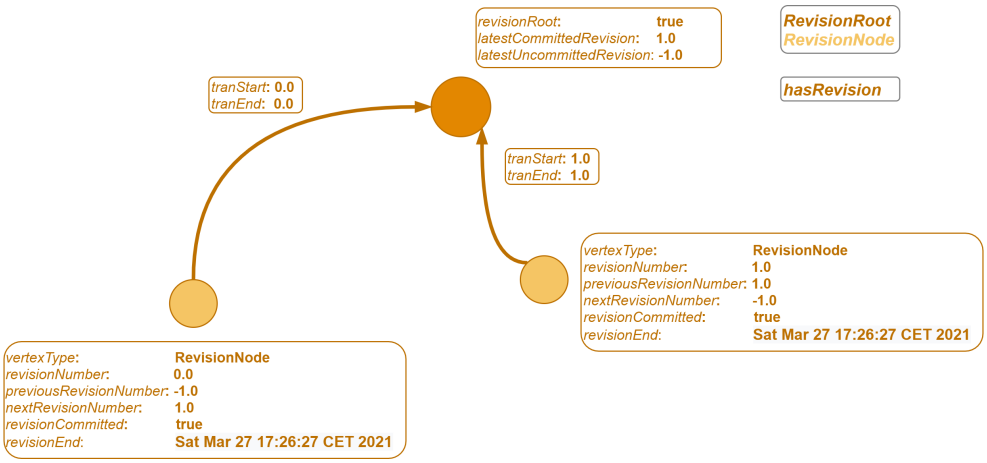


Figure 1.2: Revision root subtree vertex hierarchy

creation of minor revisions containing only smaller changes, hence complete analysis and change of objects' versions was no longer required in such cases.

The versioning of the vertices is done using control edges (which must exist for all vertices excluding the root vertices), by their *tranStart* and *tranEnd* properties which indicate the starting and ending revision. The value for each of these properties is the floating-point number, which can be split into an integral and decimal part, where the integral part represents a major revision and the decimal part represents a minor revision. The versioning of the edges is also done by the same properties. If the vertex or the edge is valid in the latest revision, the decimal part of the *tranEnd* is *.999999* and the integer part reflects the value of the property *revision number* belonging to the latest committed revision node vertex.

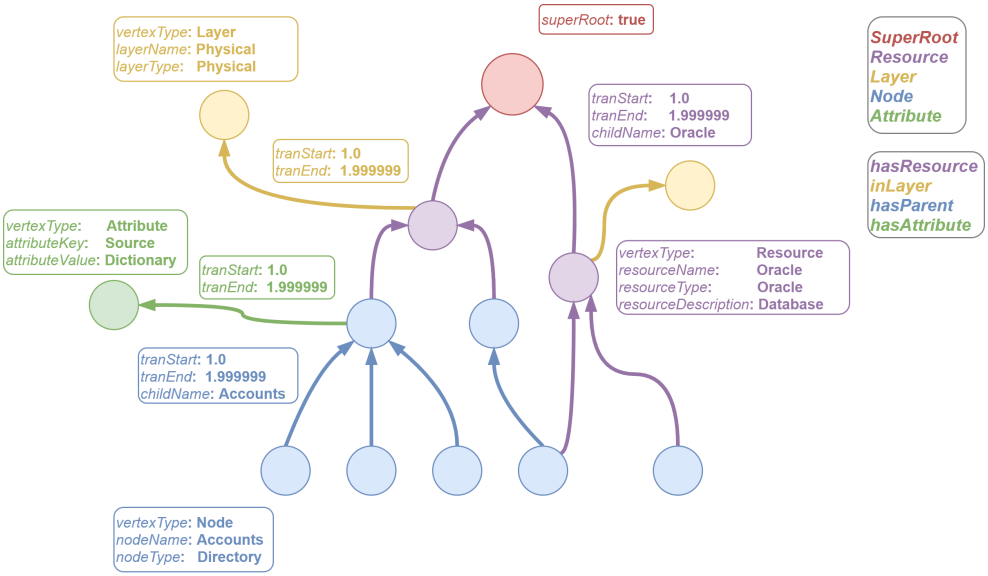


Figure 1.3: Super root subtree vertex hierarchy

- **Major revisions** Used for the initial extraction and analysis where all the metadata is inserted into the database or for the full updates where the user expects a lot of changes.
- **Minor revisions** If there are only minor changes, the user can use the minor revision, which updates changed vertices by updating the decimal part of the *tranEnd* property.

1.3 Titan

The main database to store user's metadata in a format described in (see section 1.2) used in MANTA is the graph database Titan. Titan is a scalable graph database optimized for storing and querying graphs containing hundreds of billions of vertices and edges distributed across a multi-machine cluster. It is a transactional database that can support thousands of concurrent users executing complex graph traversals in real-time. [7].

Titan is a graph database engine. Titan itself is focused on compact graph serialization, rich graph data modeling, and efficient query execution. [8]. This section focuses on the general properties of the Titan graph database, especially on the configuration that is used within MANTA.

1.3.1 Structure

Titan allows various data storage backend implementations, where data is physically stored (storage backend is required) and several index backends

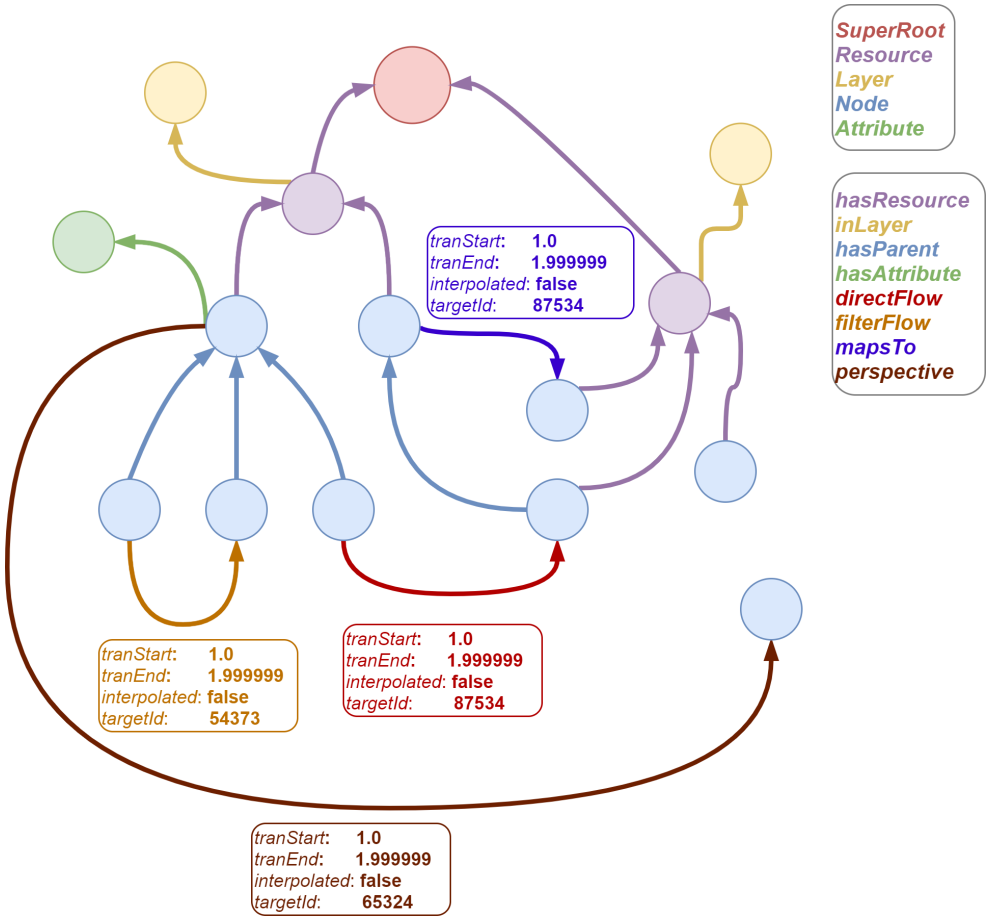


Figure 1.4: Super root subtree edge hierarchy

(index backend is optional) to provide better performance of database queries. The figure 1.5 shows the high-level architecture of Titan. This subsection describes the data model used internally for Titan as well as how edges and vertices

1.3.1.1 Internal data model [1]

Titan stores graphs in adjacency list format which means that a graph is stored as a collection of vertices with their adjacency list. The adjacency list of a vertex contains all of the vertex's incident edges (and properties).

This storage format speeds up the traversal as neighbors of each vertex are stored compactly in one place, however, the down side of this is that each edge has to be stored twice, once for source and once for target vertex. The adjacency list is sorted by the order defined by the sort key and sort order of the edge labels.

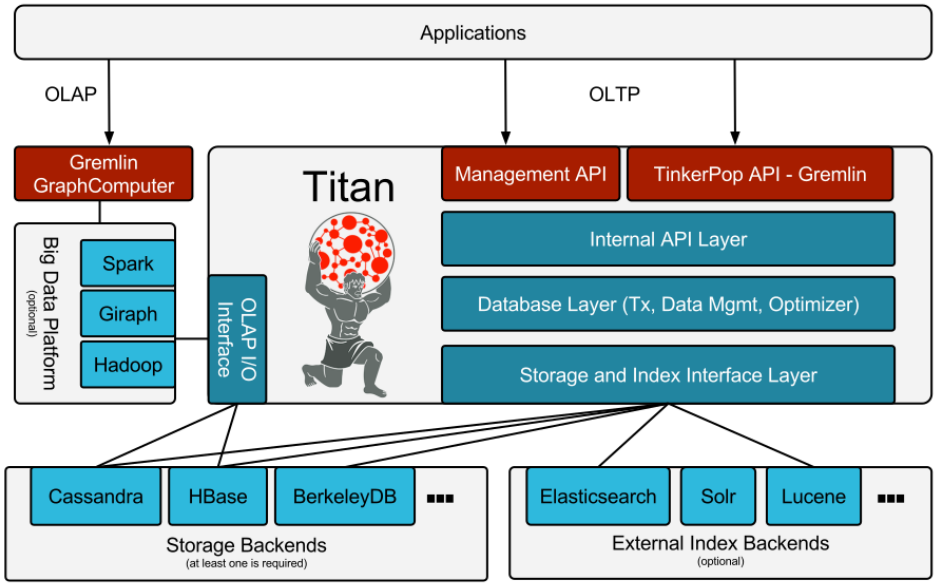


Figure 1.5: Titan architecture [2]

Titan stores the adjacency list representation of a graph in any storage backend that supports the BigTable data model depicted in the figure 1.6. Additionally, the cells must be sorted by their columns and a subset of the cells specified by a column range must be efficiently retrievable (e.g. by using index structures, skip lists, or binary search)

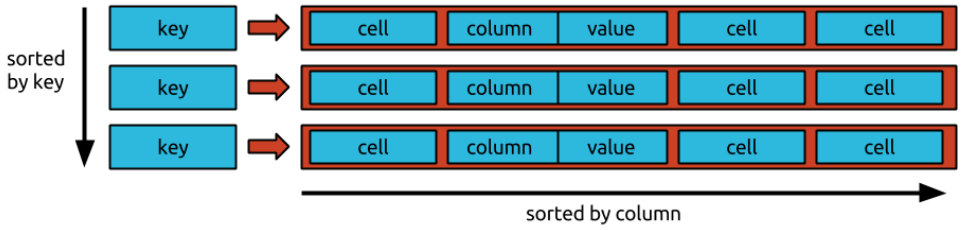


Figure 1.6: BigTable data model [1]

1.3.1.2 Edge

Titan uses edges to connect vertices within a graph. Each edge contains a label and has a certain direction. This label defines the semantics of the relationship between the vertices. Edges can contain numerous properties which can be used for indexing.

In Titan, it is possible to define edge label multiplicity, which defines a constraint of what edges can be created between given pair of vertices. Based on the documentation [9] Titan recognizes the following multiplicity settings:

- **MULTI** Allows multiple edges of the same label between any pair of vertices. This is the default multiplicity configuration that MANTA uses for all edge labels.
- **SIMPLE** Allows at most one edge of such label between any pair of vertices.
- **MANY2ONE** Allows at most one outgoing edge of such label on any vertex in the graph but places no constraint on incoming edges.
- **ONE2MANY** Allows at most one incoming edge of such label on any vertex in the graph but places no constraint on outgoing edges.
- **ONE2ONE** Allows at most one incoming and one outgoing edge of such label on any vertex in the graph.

1.3.1.3 Vertices

Each Titan vertex can have a label, but unlike edge labels, vertex labels are optional. Although optional, Titan assigns all vertices a label as an internal implementation detail.

1.3.1.4 Properties

Each vertex and edge can have numerous properties. Keys of the properties have to be of unified data type to ensure valid graph data. Each value associated with a property key has to be of one of the native Titan data types. There are three types of cardinality allowed for each property [9].

- **SINGLE** Allows at most one value per element for such key. In other words, the key-value mapping is unique for all elements in the graph. This is a default cardinality configuration as well as the one used within MANTA.
- **LIST** Allows an arbitrary number of values per element for such key. In other words, the key is associated with a list of values allowing duplicate values.
- **SET** Allows multiple values but no duplicate values per element for such key. In other words, the key is associated with a set of values.

1.3.2 Tinkerpop Blueprints

Blueprints is a property graph model interface. It provides implementations, test suites, and supporting extensions. Graph databases and frameworks that implement the Blueprints interfaces automatically support Blueprints-enabled applications. Likewise, Blueprints-enabled applications can plug-and-play different Blueprints-enabled graph backends. [10] Titan natively implements the Blueprints API which means that Blueprints is the core interface for Titan. [11]

1.3.3 Storage

There are various storage backends available to use in Titan. As mentioned previously, the backend has to support the BigTable data model. Titan documentation [12] describes three main storage backends – Cassandra, HBase and BerkeleyDB. MANTA, however, uses Persistit backend to allow simple configuration and installation for the end-users.

The Persistit storage backend runs in the same JVM as Titan and provides local persistence on a single machine. Hence, the Persistit storage backend requires that all of the graph data fits on the local disk and all of the frequently accessed graph elements fit into the main memory. This imposes a practical limitation of graphs with 10-100s million vertices on commodity hardware. However, for graphs of that size, the Persistit storage backend exhibits high performance because all data can be accessed locally within the same JVM. [13]

1.3.4 Deployment

Titan can be deployed on a remote standalone server. The user is then able to interact with it by submitting Gremlin [14] queries to the server. Titan natively supports the Gremlin Server component of the Tinkerpop stack.

Another way to use Titan is to have it embedded within the Java application. This means that all database processing happens inside the same JVM as the application. Communication with the storage backend, however, can be both local or remote.

MANTA uses the embedded version of Titan to increase query performance as the database is running on the same JVM as the server application and also to simplify the installation.

1.3.5 Index

There are also various index backends that can be used within Titan. Few examples listed in the documentation [15] are Elasticsearch, Solr, and Lucene. The index backend used within MANTA is Apache Lucene.

Apache Lucene is a high-performance, full-featured text search engine library written entirely in Java. It is a technology suitable for nearly any application that requires full-text search, especially cross-platform. [16] Titan supports Lucene as a single-machine, embedded index backend, which works well with Persistit storage backend.

1.3.6 Transactions

This subsection explains how transactions are handled within Titan. It is important to understand as all of the database algorithms are using the transactions.

Almost all interaction with Titan is associated with a transaction. Titan transactions are safe for concurrent use by multiple threads. Methods on a `TitanGraph` instance like `graph.v(...)` and `graph.commit()` perform a `ThreadLocal` lookup to retrieve or create a transaction associated with the calling thread. Callers can alternatively forego `ThreadLocal` transaction management in favor of calling `graph.newTransaction()`, which returns a reference to a transaction object with methods to read/write graph data and commit or rollback. [17]

The isolation level and ACID support are configured through the storage backend, meaning the graph database isolation level is inherited from the isolation level of the underlying storage backend. The isolation level used within MANTA is `repeatable read`, meaning that the data read repeatedly is always the same, even if other transaction concurrently modified it.

1.3.6.1 Transaction handling

As in any transactional storage, every graph operation in Titan occurs within the context of a transaction. According to the Blueprints' specification, each thread opens its transaction against the graph database with the first operation (i.e. retrieval or mutation) on the graph.

All subsequent operations occur in the context of that same transaction until the transaction is explicitly stopped or the graph database is `shutdown()`. If transactions are still open when `shutdown()` is called, then the behavior of the outstanding transactions is technically undefined. [17]

1.3.6.2 Transactional scope

All graph elements (vertices, edges, and types) are associated with the transactional scope in which they were retrieved or created. Under Blueprint's default transactional semantics, transactions are automatically created with the first operation on the graph and closed explicitly by committing or rollbacking.

Once the transaction is closed, all graph elements associated with that transaction become stale and unavailable. However, Titan will automatically transition vertices and types into the new transactional scope. Edges, on the

other hand, are not automatically transitioned and cannot be accessed outside their original transaction. They must be explicitly transitioned. [17]

1.3.6.3 Transactional failures

A transaction can sometimes fail due to several reasons. When committing a transaction, Titan will attempt to persist all changes to the storage backend. This might not always be successful due to IO exceptions, network errors, machine crashes or resource unavailability [17]. To handle failures one has to make sure that the transaction is retried upon failure so the system is eventually in a desirable state.

Based on the documentation [17] transaction failures that can occur are split into two categories:

- **potentially temporary** Potentially temporary failures are those related to resource unavailability and IO hiccups (e.g. network timeouts). Titan automatically tries to recover from temporary failures by retrying to persist in the transactional state after some delay. The number of retry attempts and the retry delay is configurable.
- **permanent** Permanent failures can be caused by complete connection loss, hardware failure, or lock contention.

1.3.6.4 Multi-threaded transactions

To utilize multi-core systems one can make use of multi-threaded transactions, where multiple threads share one transactional context. Titan supports multi-threaded transactions through Blueprint's `ThreadedTransactionalGraph` interface. [17] To open a thread-independent transaction, one has to use the `newTransaction()` method.

1.4 Neo4j

As the Titan graph database is no longer supported, MANTA is in the process of choosing the candidate for replacement. Neo4j is looking very promising as it is quite mature (documentation-wise and production readiness) and the performance of the required algorithms and queries is better than within other candidate databases.

Neo4j's primary product and focus is its graph database that stores data in the form of nodes and relationships. It handles both transactional and/or analytics workloads and is optimized for traversing paths through the data using the relationships in the graph to find connections between entities. [18] Neo4j supports both embedded and remote versions of the graph database.

This section describes Neo4j from its structural perspective, query language Cypher, and transactional properties along with its possible pitfalls as

that are closely related to the main goals of this work - optimizing graph algorithms that run within a transactional context.

1.4.1 Structure

As with all graph databases, Neo4j is comprised of vertices (in Neo4j terminology nodes) and edges (called relationships), which are, in conjunction with their properties, forming a property graph.

Neo4j documentation [18] describes nodes as entities in the graph, which can hold any number of attributes (key-value pairs) called properties. Nodes can be tagged with labels, representing their different roles in modeled domain. Node labels may also serve to attach metadata (such as index or constraint information) to certain nodes and are optional.

Relationships connecting two nodes provide directed, named, semantically relevant connections between them. A relationship always has a direction, a type, a start node, and an end node. Like nodes, relationships can also have properties. Due to the efficient way relationships are stored, two nodes can share any number or type of relationships without sacrificing performance. Although they are stored in a specific direction, relationships can always be navigated efficiently in either direction. [18]

1.4.2 Deployment

Neo4j supports both remote server deployment as well as embedded deployment. Running Neo4j in server mode involves having all the classes and logic to access and process interactions with the Neo4j database contained within its dedicated process, completely separate from any clients wishing to use it. [19] HTTP-based RESP API is provided to communicate with the server.

When using the embedded mode, the Neo4j engine is running on the same JVM as the application allowing direct access, which provides the user with greater control but also. Using Neo4j in embedded mode provides better performance than using the server mode as the native Java API is not slowed down by the network round-trips of REST communication.

1.4.3 Querying

In Neo4j there are three approaches to access and traverse the graph. Users can use Java Core API, traversal framework, or Cypher query language providing various degrees of control.

1.4.3.1 Java Core API

Java Core API is the most basic one and is the most similar to Titan's way of graph manipulation. Core API provides the biggest degree of freedom as the user is solely responsible for the graph traversing. Each node or relationship

can be queried for related objects, hence allowing the user to create complex traversals. The listing 1.1 demonstrates querying for node's relationships of certain direction and type and retrieval of relationship's start node.

```
node.getRelationships(
  Direction.INCOMING,
  RelationshipType.withName("hasParent")
)
relationship.getStartNode()
```

Listing 1.1: Example of Neo4j Java Core API

1.4.3.2 Traversal framework

The Neo4j Traversal framework Java API is a callback-based, lazily executed way of specifying desired movements through a graph in Java. [20] Traversal framework provides a compact interface to specify how to traverse the graph. User has to specify description to create an instance of traverser — define what to traverse, typically in terms of relationship direction and type, order of evaluation (depth-first or breadth-first), uniqueness determining whether nodes or relationships can be traversed multiple times, evaluation criterion to determine when to stop and starting nodes of traversal. However, this framework is now deprecated.

The listing 1.2 demonstrates breadth-first traversal starting from arbitrary *startingNode*, which traverses using incoming *hasParent* relationships. Traversal continues until whole accessible graph is visited and each relationship is visited at most once.

```
tx.traversalDescription()
    .breadthFirst()
    .relationships(RelationshipType.withName("hasParent",
      Direction.INCOMING)
    .evaluator(path -> Evaluation.INCLUDE_AND_CONTINUE;
    .uniqueness(Uniqueness.RELATIONSHIP_GLOBAL)
    .traverse(startingNode)
```

Listing 1.2: Processing of sub-phases

1.4.3.3 Cypher

The main asset of Neo4j is a Cypher query language. Cypher provides a powerful declarative way to query the graph. Its syntax provides a visual and logical way to match patterns of nodes and relationships in the graph. It is a declarative, SQL-inspired language for describing visual patterns in graphs using ASCII-Art syntax. [21] User can use Cypher to query the database both remotely and as embedded.

The listing 1.3 demonstrates basic Cypher syntax. Assume there are nodes of label *Node*, which are connected using a relationships of type *directFlow*, and the task is to retrieve all target nodes (nodes which are connected to source node with an incoming relationship) of a source node having a property *name* with value *TableA*.

```
MATCH (a:Node)-[:directFlow]->(b:Node)
  WHERE a.name='TableA'
RETURN b
```

Listing 1.3: Example Cypher query with properties inside `WHERE` clause

Alternatively, the syntax is shown in the listing 1.4, where property matching is used instead of `WHERE` clause and is equivalent to the previous query.

```
MATCH (a:Node {name:'TableA'})-[:directFlow]->(b:Node)
RETURN b
```

Listing 1.4: Example Cypher query with matching properties

1.4.4 Storage

In subsection 1.3.3, the storage backend used in Titan was discussed listing multiple variants that can be used as a storage backend. Neo4j, on the other hand, is a native graph database meaning the underlying storage is optimized for graphs.

What makes graph storage distinctively native is the architecture of the graph database from the ground up. Graph databases with native graph storage have underlying storage designed specifically for the storage and management of graphs. They are designed to maximize the speed of traversals during arbitrary graph algorithms by ensuring that data is stored efficiently by writing nodes and relationships close to each other. [22]

1.4.5 Index

Neo4j documentation [23] states there are two different index types: b-tree and full-text.

- **B-tree** B-tree indexes can be created and dropped using Cypher. Users typically do not have to know about the index in order to use it, since Cypher's query planner decides which index to use in which situation. B-tree indexes are good at exact look-ups on all types of values, and range scans, full scans, and prefix searches.
- **Full-text** Full-text indexes differ from B-tree indexes, in that they are optimized for indexing and searching text. They are used for queries that demand an understanding of language, and they only index string

data. They must also be queried explicitly via procedures, as Cypher will not make plans that rely on them.

1.4.6 Transactions

As stated earlier, Neo4j supports both transactional and analytical processing. Database algorithms used within MANTA often update the stored model, hence requiring transactional processing to ensure data's validity.

1.4.6.1 Transactional scope

Transactions in Neo4j use a read-committed isolation level, which means they will see data as soon as it has been committed and will not see data in other transactions that have not yet been committed. This type of isolation is weaker than serialization but offers significant performance advantages whilst being sufficient for the overwhelming majority of cases. [24]

1.4.6.2 Transaction failures

As Neo4j supports concurrent transactions and due to that there exists a locking mechanism to ensure no 2 transactions try to modify the same nodes or relationships. Because of this deadlock can occur. More information on these deadlocks can be found in subsection 2.4.1 as it closely relates to the optimization work.

1.5 Merger

This section describes the merging algorithm. After data is initially extracted and analyzed, all the entities have to be merged into the database. This includes new objects, but also already existing objects, which require updating of their revisions. The merging process is necessary for keeping track of different versions of user's data throughout different runs of flow analysis. This part of the flow is one of the biggest bottlenecks, as the algorithm is run only in serial and can take hours to finish in substantially large analyzed systems.

The first part of this section describes the merger from a higher perspective – the arrival of the merge request, types of data it contains, how it gets preprocessed, and how are the transactions managed. The second part describes various graph traversing and modifying operations. The last part focuses on the core of the merger algorithm, in other words, details of how the specific objects are merged.

1.5.1 Higher level

This subsection focuses on how the transactions are handled within MANTA, what is the format of the merge request and the specific way transactions are created and committed throughout each merge request processing.

1.5.1.1 Transaction management

To avoid problems with concurrent writes locks for accessing the database are used on the application level. When creating a new transaction for accessing the database, `TransactionLevel` value is used for determining the type of isolation. Based on this value, a specific application-level lock is acquired, followed by a creation of a new transaction, which is then used to process the specific request using a callback. Java `ReentrantReadWriteLock` lock implementation is used to guard the database access for different threads. There are 5 different values of `TransactionLevel`:

- **READ** Used for reading access of the database, which can be performed by multiple threads at the same time, as the reading lock can be held by more threads simultaneously. The read-only transaction is created for this level.
- **READ_NOT_BLOCKING** Used for non-blocking reading access. No lock is applied, meaning the other transactions can access and modify the database concurrently. The read-only transaction is created for this level.
- **WRITE_EXCLUSIVE** Used for writing to the database exclusively as the writing lock can be owned by one single thread and only if there is no other thread that holds the reading lock. For this isolation level, a read-write transaction is used.
- **WRITE_SHARE** Used for concurrent modification, the same lock as in `READ` level is applied, ensuring that processing multiple transactions within this isolation level is possible. The transaction type is the same as in the exclusive writing level.
- **WRITE_SOURCE_CODE** Used for storing the source code files. The lock used is distinct from the main database lock as it is used to modify different subgraphs. Only one thread can hold this lock. The same type of transaction as in other writing levels is used.

1.5.1.2 Merge request format

The input of the merging process is a file containing a linearized graph represented by a list of several object types – source codes, layers, resources, nodes,

node attributes, edges, and edge attributes. This order is always respected as the merging of certain types must precede other types.

One of the reasons for the specific order is the layers preceding the resources as each resource always belongs to a specific layer. Another reason for this order is the merging of the nodes as it must happen before the merging of the node attributes and edges because a node attribute is always bound to a certain node and an edge can only exist between already existing nodes. The last case where the order is important is the precedence of the source code objects as the node attribute objects may reference it. A detailed description of the specific object types and their attributes:

- **Source code** Represents stored source node. Consists of local identifier, name, hash, technology, and connection string.
- **Layer** Represents resource's layer. Consists of local identifier, layer name, and layer type.
- **Resource** Represents resource. Consists of local identifier, name, type, description, and identifier reference to the layer it belongs to.
- **Node** Represents node. Consists of local identifier, reference to its parent and resource, name, type, and optionally a flag (`REMOVE_MYSELF`, `REMOVE`) used in minor revisions for more efficient merging. The reference to its parent node can be blank if the node has no parent.
- **Node attribute** Represents attribute of the node. Consists of the local identifier, key, and value of the attribute.
- **Edge** Represents edge between two nodes. Consists of local identifier, name, type, the source node, and the target node.
- **Edge attribute** Represents attribute of the edge. Consists of the local identifier, key, and value of the attribute.

1.5.1.3 Merge orchestration

As a certain technology is being analyzed, the merge requests are sporadically sent to the MANTA Flow Server endpoint. Each technology can be analyzed in parallel, meaning multiple requests can arrive in a small time frame. After the request arrives, the head of the stream gets processed, as it contains source code objects which need to be validated against the MANTA license.

After that, the request proceeds to the merging phase, where it tries to retrieve `WRITE_EXCLUSIVE` lock (see part 1.5.1.1) to merge its content. Each merging transaction is committed after processing 500 objects and the lock is released. The count of objects was determined to be an optimal threshold to minimize commit time. Each request keeps track of the objects processed

in their most recent transaction in case of transaction failure to retry the processing.

By limiting the number of merged objects within single transaction, various merging requests contend with each other for the database lock, and only after owning process finishes processing its batch of objects the lock is released, which is then available for other processes. An important thing to note is that it is not only available for the merging processes but also for other MANTA modules which use the database mostly for reading (can be done concurrently), however, if at least one thread holds the reading lock the writing lock can not be acquired.

1.5.2 Graph functions

This subsection describes various graph methods used to modify and retrieve entities within the Titan graph used for merging purposes. Within embedded Titan, the imperative approach is used, meaning graph traversals are done by querying vertices recursively. There are 3 structures containing these methods for more specific reasons.

1.5.2.1 Graph creation

Graph creation focuses on creating vertices and edges connecting various vertices. Methods used for merging purposes:

- **createLayer** Used for creating layer vertex.
- **createResource** Used for creating resource vertex along with an edge connecting it to a specific layer to which it belongs.
- **createNode** Used to create node vertex. There are 2 versions of this method, one where only an edge connecting the node to its parent is created and the second, where an edge connecting it to its resource is created as well (see part 1.2.3).
- **createEdge** Creates an edge of a specific type between two given vertices. Contains validation of whether the specific type of edge can be created between given vertices.
- **createNodeAttribute** Creates a node attribute vertex with an edge connecting it to the owning node vertex.
- **createRevisionNode** Creates a new revision node vertex in the revision subtree to represent new revision.
- **createSourceCode** Creates a new source node vertex in the source node subtree.

1.5.2.2 Graph operation

Graph operation focuses on traversing and retrieval of specific entities contained within a graph.

- **getVertexByQualifiedName** Retrieves the vertex by specified path. The path contains a list of vertices represented by three values – name, type, and resource's name. The graph is then traversed within a given revision starting from *super root* vertex going through resources and vertices by the list values.
- **setSubtreeTransactionEnd** This method is used for incremental updates. The revisions of the whole subtree of certain node vertex are updated and in some cases, invalid vertices are deleted. This operation is done by traversing the edges connecting the given node to its subtree (by MANTA hierarchy) of a node recursively.
- **getResource** Retrieves the resource of the vertex by traversing the graph from the given vertex to its resource. If the vertex is of *node* type, the traversal is done through its parent recursively until resource vertex is reached. If the vertex is of *attribute* type its owning node is first retrieved and then the same approach is used.
- **getLayer** Retrieves the layer of the vertex by retrieving the resource of the vertex by using the previous method and then retrieving the directly connected layer.
- **getVertexPathString** Retrieves the full path of the vertex. The full path of the certain vertex is retrieved by traversal starting from the given vertex to its resource. The same approach is used as in *getResource* method, but instead of retrieving the resource, a string concatenation of all vertices on the path is returned.
- **getEdge** Retrieves the edge of a specific type connecting two given vertices within a specific revision.

1.5.2.3 Revision utilities

Revision utilities contain methods for updating revisions of the control edges connecting certain vertices. Methods used in the merger process are as follows:

- **setVertexTransactionEnd** Sets the new revision of the given vertex. Used when merging an already existing vertex to update its revision to a newer one by updating the revision property of its control edge.
- **setEdgeTransactionEnd** Sets the new revision of the given edge. Used when merging an already existing edge to update its revision property.

1.5.3 Current merging process

As the specific object is read from the request's input stream, the merging of that object begins. Merging is done differently for every object type. Through merging, the contextual structure is maintained to keep track of mappings of local identifiers of input objects to their merged counterparts in the graph. This subsection describes how specific types of input objects are merged. Merging of each type is of similar structure – at first, the previous existence of the object is determined and then based on the gained information, the object is either created or its revision is updated. The common merging processes of the vertices and edges are depicted in the figures 1.7 and 1.8 respectively.

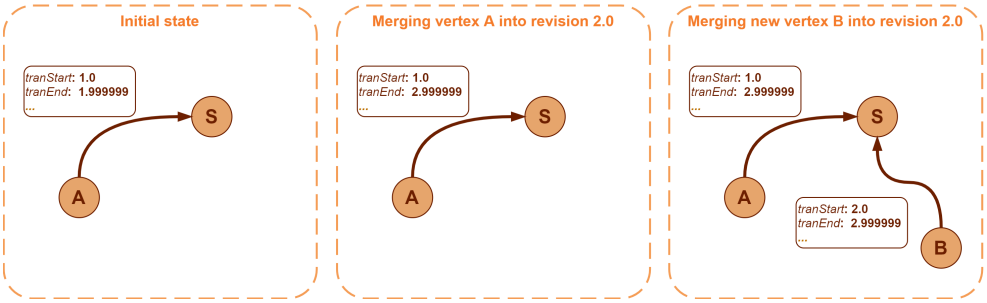


Figure 1.7: Common vertex merging

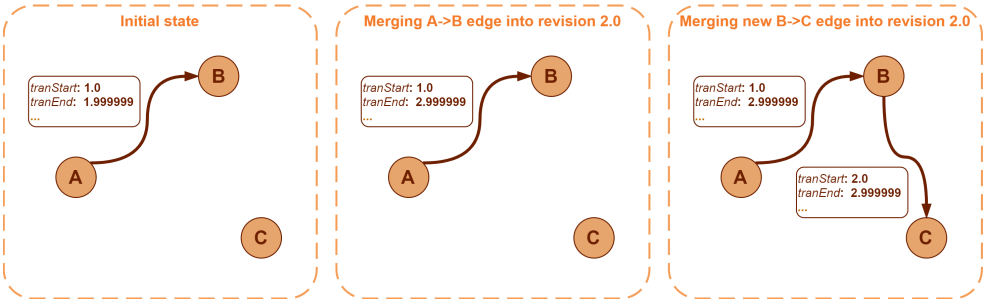


Figure 1.8: Common edge merging

1.5.3.1 Source code

Vertices representing source nodes are directly connected to the source root vertex. Due to this, the source root vertex is first queried to determine the previous existence of the source node vertex. This is done by querying for vertices connected to the source root vertex by an edge of type hasResouce having the value of *tranEnd* property higher or equal to the *latestCommittedRevision*. To differentiate it from other source node vertices, its name and technology are compared to the input object's properties.

If the given source node existed in the latest revision, ending revision (*tranEnd*) of its control edge is updated to merged revision number and in the case, it did not exist, it is created with an edge connecting it to the source root vertex.

1.5.3.2 Layer

Layer type is processed together with the resource as the control edge has to be added between them to mark validity of the layer and the resource's affiliation to the layer, so the merging of this type only consists of adding the mapping of local layer identifier to its attributes, hence no database access is done there.

1.5.3.3 Resource

Resource vertices are connected to the super root vertex by hasResource edge. So to determine the previous existence of the resource, the super root vertex is queried for all its children connected by hasResource edge belonging to the latest revision and having the value of *childName* property equal to the name from the input. Then the correct resource vertex is determined by comparing its type (*resourceType*) to the type retrieved from the input properties.

Previous non-existence of the vertex representing a given resource implies that its layer does not exist either so both the resource and the layer (layer information retrieved using the mapping table from the contextual structure) are created. Firstly the layer vertex is created and then the resource vertex is created with two edges, one serving as a control edge for the resource (having hasResource type), connecting it to the super root, and the second one as a control edge of the layer (having inLayer type).

If the vertex representing resource exists, revision of its control edge is updated and the database is queried to determine whether the layer to which the resource is supposed to belong exists. If the vertex representing layer exists, the ending revision of its control edge is updated. It can also happen that the resource already has a layer, whose type does not match with the type of the input properties, but this difference is ignored and the same approach is used. If the vertex representing layer does not exist, both the vertex and the control edge connecting it to the resource vertex are created.

1.5.3.4 Node

Each node vertex is either connected directly to the resource by hasResource edge or its parent node vertex by hasParent edge. To decide which vertex is to be queried to determine the previous existence of the node, input properties are used. At first, both vertices representing the node's resource and parent nodes are fetched from the database using their database identifiers retrieved from the mapping contained in the contextual structure. Both resource's and

parent's vertex must exist (if specified in the input properties) as it had to be processed before the given node following the structure of the input file. Next, the type of control edge is determined by whether the node has a parent or a resource. If the parent's value in input properties was blank, hasResource is used, otherwise hasParent is used.

After determining the type of control edge, the correct vertex (parent or resource) is queried for all vertices connected to its by the determined edge type belonging to the latest revision having *childName* property value set to the *name* value retrieved from the input properties. The correct vertex is determined by its *nodeName* and *nodeType* properties.

If the correct vertex exists, mapping of a local identifier to database vertex identifier is added to the contextual structure for further use, and the following step is determined by its flag (input property). If the flag's value is `REMOVE_MYSELF`, the node with its whole subtree is removed. This means all the ending revisions (*tranEnd*) of control edges of the node's subtree are set to the latest committed revision. Another possible value of the flag is `REMOVE`, which behaves in the same way as `REMOVE_MYSELF`, but preserves the node itself. If the already existing node does not contain any flag, only the *tranEnd* property of its control edge is updated.

In the other case, the new vertex representing the node along with its control edge. In case that node belongs to a different resource than its parent (see part 1.2.2) 2 edges have to be created. One of these edges is the node's control edge connected to its parent and the other is connecting it to its resource. In the case that the node shares the same resource as its parent, only the edge connecting it to its parent is created. The decision of whether the resource of the node is equal to the resource of its parent is done by retrieving the parent's resource by traversing the graph upwards and then comparing it to the resource's node.

1.5.3.5 Node attribute

Each node attribute belongs to a specific node. To determine whether the node attribute exists, the owning node vertex has to be fetched first. It is fetched from the database using its internal identifier retrieved from the contextual structure. There are two special types of node attributes *mapsTo* and *sourceLocation*.

If the attribute key is *mapsTo*, it represents the mapping of the source node to a different node, represented by a path to the node from the *super root* vertex. Target node vertex is then retrieved. After retrieving the vertex the database is queried for the *mapsTo* edge between source and target vertices belonging to the latest revision. If it does exist, its *tranEnd* property is updated, otherwise, it is created.

If the value of the key is *sourceLocation*, the stored database identifier of the source code is used as the attribute value and the new attribute vertex

with an edge connecting it to owning node vertex is created.

Then the owning node vertex is queried for adjacent vertices connected by hasAttribute edges belonging to the latest revision. The correct node attribute vertex is chosen by its *attributeKey* and *attributeValue* properties. If the vertex exists, revision of its control edge is updated, otherwise, the vertex with its control edge is created.

1.5.3.6 Edge

Both source and target vertices are retrieved based on their identifiers from the mapping retrieved from the contextual structure. After the two vertices are retrieved, the database is queried for all the edges of the requested type belonging to the latest revision, which are connecting the source and target vertices. If the edge already exists between the two nodes its *tranEnd* property is updated. Otherwise, a new edge with specified properties is created between the source and target vertices.

The *perspective* edges (see subsection 1.2.2) are processed a bit differently. If the merged edge is of this type, all outgoing edges of the source vertex having *perspective* type are retrieved. Then the layers of all target vertices of the retrieved edges are compared with the layer of target vertex, potentially resulting in multiple retrievals of the layers, each being an expensive operation. If the layer of the target vertex is not equal to any other compared layer, the *node attribute* vertex belonging to the source node is created containing the layer's name as *attributeKey* and path to target node as *attributeValue*. The edge connecting source and target vertices are also created, but with an additional *layer* property with the layer's name as a value. The described process is depicted in the figure 1.9.

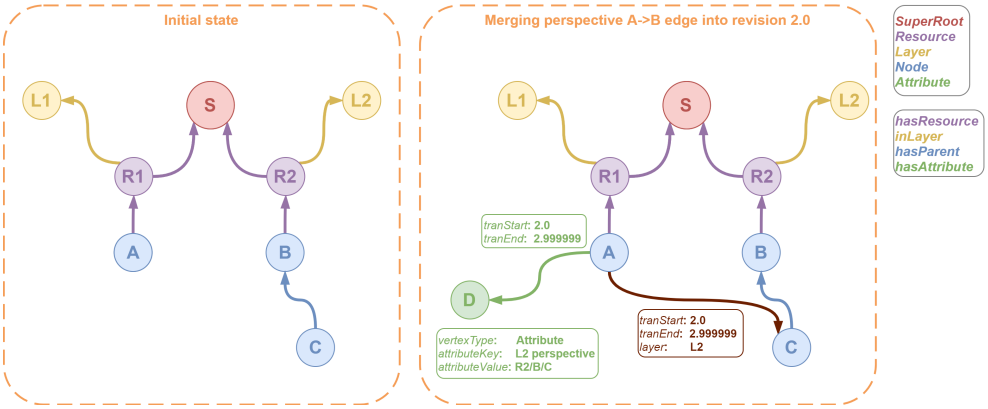


Figure 1.9: Perspective edge merging

1.5.3.7 Edge attribute

Each edge can have its attributes, which are stored as the properties of the edge in the database. The edge can be easily retrieved as the database identifier was already collected when processing the given edge. After getting the edge, its attributes are checked to determine whether the edge already contains the input attribute with its value.

If the edge does not contain the attribute, it is added. If it does contain the attribute but with a different value, the value is overwritten if the edge was created in the merged revision. If the edge was created in the previous revision, it is copied with a new attribute value, and the old edge's *transEnd* property is set to *latestCommittedRevision* number. Various situation of merging edge attributes are depicted in the figure 1.10.

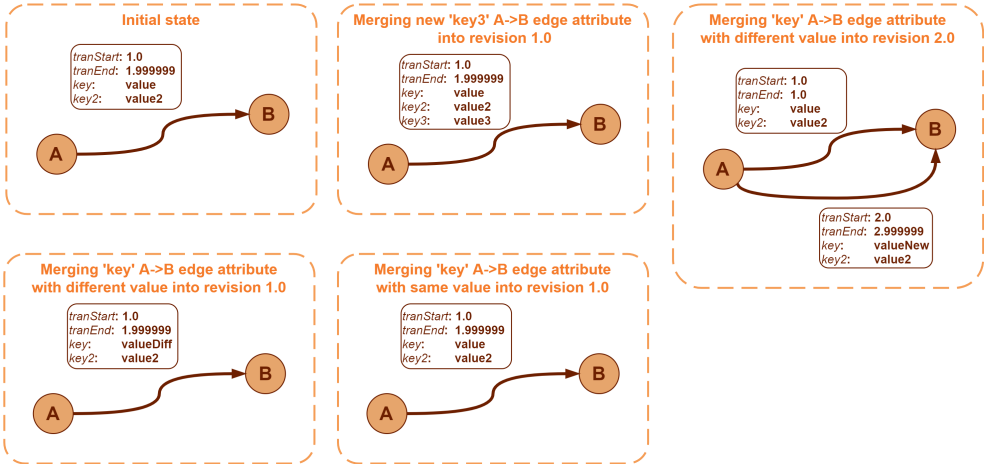


Figure 1.10: Edge attribute merging

The goal of this chapter was to acquaint the reader with MANTA in general and the data model it uses. As the Titan is currently used database, its main properties were described as well as general properties of the Neo4j database, to which the project will be migrated. The last focus of the chapter was the current merging algorithm, whose migration and optimization will be the main focus of the following chapter.

Chapter 2

Analysis

This chapter describes the possible parallelization of the current merging algorithm within the Titan graph database. The chapter later focuses on the migration of the merging process to the Neo4j graph database as well as possible improvements to increase its overall performance. These improvements include minor optimization steps as well as parallelization of the whole process.

2.1 Titan parallelization

Titan graph database supports multithreaded transactions (as mentioned in part 1.3.6.4), so it might be possible to use this feature to improve the merging process within the current database. If successful, this would only be a temporary solution to boost performance as Neo4j, to which the MANTA project will soon be migrated, does not support threaded transactions. This section describes previous attempts to parallelize the merging process as well as possible parallelization using multithreaded transactions.

2.1.1 Optimistic parallel merging

There was an attempt to enable merge requests to run in parallel. During merging `WRITE_SHARE` lock is used allowing other threads to acquire the lock as well and create transactions for database access.

However, in some cases two or more merging processes may want to edit a single vertex or edge which causes `RepositoryRollbackException` exception to be thrown in all secondary threads, causing them to rollback. When this occurs each transaction is re-run using the kept journal and all objects are processed again.

Another problem is that if two or more merging processes want to add the same new vertex representing a certain type and the requested vertex did not exist at a time when merging processes attempted to determine its previous existence. Both evaluate the vertex as non-existent (which was correct at

that given moment), so both create it and then proceed to work with created database vertex while processing following vertices and edges causing more distinct subtrees to be created. This is later fixed by traversing the database graph and removing the duplicates, however, this approach proved to be less effective than the sequential approach.

2.1.2 Parallelization posibilities

There are two possible solutions for parallel merging within the Titan graph database. One solution is to use multithreaded transactions throughout for multiple merging requests where the database would be synchronized through the shared singleton transaction. Synchronizing multiple merge requests as in previously described optimistic merging is not possible as various transactions do not see local changes of other transactions. To synchronize multiple transactions one would need to ensure that no two transactions would attempt editing the same subtree to avoid locking exception or duplicate subtree creation. To make this possible, locks for all vertices would need to be maintained – which could be done using full path of the vertex from resource, but this could bring another problem such as deadlocks and synchronization overhead.

On the other hand, if a multithreaded transaction for all merging processes is used, writing synchronization is not required as it is all done within one transaction, reading locks, on the other hand, are still required to prevent merging processes from creating duplicate objects in the database. To have only one running merging transaction, various operations need synchronization, be it creating and committing the transaction, controlling the count of objects processed as well as troubleshooting in case of failed or rollbacked transaction, as all participating merging processes would lose their progress.

2.1.3 Multithreaded transaction

This subsection discusses various locking scenarios required for merging specific object types as well as transaction synchronization mechanisms.

2.1.3.1 Transaction synchronization

To maintain a single transaction used by various threads running in parallel synchronization of a transaction is required in a number of places:

- **Creation or transaction retrieval** Before merging a certain object, the current transaction has to be retrieved by merging the thread. If there is no transaction existing, a new one must be created. This process must be guarded by a lock to ensure exclusive access.
- **Commiting** After a certain number of processed objects is reached, only one thread can be allowed to commit the transaction, while all other threads have to wait before continuing their work.

- **Rollbacking** If the committing of the transaction fails or a certain thread fails transactional processing, the transaction is rollbacked. It must be ensured that all participating transaction rerun their journal, as all processing within that transaction is deemed invalid.
- **Counting** There has to be an atomic counter to count the number of objects processed within a current transaction and a number of objects submitted to merging to make sure that the commit happens at right time – after all submitted objects are processed.

2.1.3.2 Locking

Each object type requires reading locks in different places to ensure that no 2 transactions attempt to create distinct vertices or edges representing the same object. Description of locks for each object is as follows:

- **Source code** To process this type, the reading lock has to be applied for the source root vertex to determine whether the vertex representing the given source code already exists.
- **Layer** This type does not require any kind of locking as the processing is not accessing the database and only storing the layer's attributes.
- **Resource** Reading lock must be applied to the super root as it is queried for the resource vertices. If the resource does not exist, the reading lock is kept until the resource, its layer, and all the control edges are created, otherwise, it is released.
- **Node** To avoid creating more distinct subtrees the control vertex has to be locked and as it was already processed previously, its database identifier is already stored, so the lock can be bound to the identifier.

  After determining control vertex and edge, the database is queried for the control node's children and this is where the reading lock must be applied so that no other process queries for children of this node. Even tighter lock could be applied to lock pair of parent and child (parent's database identifier is known, however for input vertex only name and type is known). This tighter locking would mean that database could be queried for different children in two different merging processes. If the vertex did not exist it will be created and the lock will be released or if it did exist, the lock can immediately be released.
- **Node attribute** Node attribute can be of different types and different types require different locking.

  If the type is *mapsTo*, the target vertex, whose existence is ensured, as it had to be merged in the previous part of the analysis, gets fetched

based on its qualified name and then the database is queried for the edge of type *mapsTo* between the source vertex and the target vertex. This part needs to be locked by reading lock for edges, which is described in the following edge locking.

If the node attribute is of normal type or processed attribute of type *sourceLocation*, the reading lock needs to be applied for querying of node attribute vertices belonging to a given node to prevent duplicates.

- **Edge** To prevent duplicate edges from being created some locking must be applied here as well. Locking of the edges was described in the *mapsTo* attributes, as the edges are created there.

  As the edges are unidirectional, using the reading locks for both source and target vertices is not required, locking one of those is sufficient. Tighter locking which would consist of source and target vertices and edge's type could be used here as well, and would allow more permissive querying for the edges. The lock can be released after the edge was created or after determining it was already existing.

- **Edge attribute** Database retrieval of the edge has to be locked as the following processing depends on the retrieved edge. Only after the processing, the reading lock can be released.

This approach, however, proved to be quite inefficient and problematic as various problems arose after it was implemented. A detailed description will be provided in the last chapter 4.

2.2 Neo4j migration

> **Graph entities naming**
>
> In the following sections and chapters graph vertices will be referred as nodes and graph edges will be referred as relationships when referring to Neo4j structure.

This section discusses Neo4j from a querying perspective to explore ways to migrate the database accessing parts of the merging algorithm. The first subsection explains various Cypher structures needed to implement the algorithm. The `MERGE` command is discussed in a separate subsection, as it might be directly applicable to the algorithm.

In the case of Neo4j, the plan is not to use the embedded version as it currently is with Titan, but to use the server mode to provide better scalability and the possibility of distribution.

2.2.1 Graph functions

A basic example of Cypher query language was already shown in part 1.4.3.3 of the Neo4j description, but this subsection addresses structures more specific to the merging algorithm. Mainly used queries in the merging algorithm are the retrieval queries used to fetch the children of a certain object by specific properties and relationships or to fetch the objects by their database identifiers. Another category of simple queries is represented by the creational queries used to create nodes and relationships.

The more complex queries are required for the merging operations which need graph traversals such as retrieval of the object by its path from *super root* vertex or traversal and manipulation of the subtree of a certain node.

2.2.1.1 Simple queries

The most commonly used operations are simple selections as well as creations. Only Cypher statements required are `MATCH`, `WHERE` or alternatively using property matching and `CREATE`. Here are a few examples of how certain operations can be performed:

- **Retrieval by identifier** Both nodes and relationships have their internal identifier by which they can be queried. The listing 2.1 shows query can be used to retrieve node with an identifier value `547380`. Retrieval of the relationship by its identifier is shown in the listing 2.2.

```
MATCH (node)
  WHERE ID(node)=547380
RETURN node
```

Listing 2.1: Retrieval of a node by its identifier

```
MATCH ()-[relationship]-()
  WHERE ID(relationship)=547380
RETURN relationship
```

Listing 2.2: Retrieval of a relationship by its identifier

- **Retrieval of nodes to ensure existence** Merging of every object type contains the part, where the previous existence of the object is determined. This is either the existence of the node connected by a relationship to another existing node, or the existence of a relationship between two existing nodes.

  Consider the following example where the goal is to retrieve *nodeB* which is connected to *nodeA* (having database identifier `547380`)by a relationship of type `RelationshipType` incoming to *nodeA*. Following query can be used to retrieve *nodeB* by the given specification.

```
MATCH (nodeA)<-[relationship:RelationshipType]-(nodeB)
  WHERE ID(nodeA)=547380
RETURN nodeB
```

Listing 2.3: Retrieval of a node by its relationship

- **Simple creational operations** Another category of queries contains the simple creational queries which either create a new node connected to a given node or create a relationship between 2 existing nodes. The listing 2.4 demonstrates the creation of a new node of label `Node` with an outgoing relationship of type `RelType` to a given node with identifier `547380`.

```
MATCH (givenNode)
  WHERE ID(givenNode)=547380
  CREATE (givenNode)<-[relationship:RelType]-(node:Node)
RETURN node
```

Listing 2.4: Creation of a new node connected to certain node

- **Property modification operations** Last very common operation is to update revision properties on relationships. This can be done using `SET` statement (see listing 2.5).

```
MATCH ()-[relationship]-()
  WHERE ID(relationship)=547380
SET relationship.tranEnd=2.5
```

Listing 2.5: Updating *tranEnd* property of certain relationship

2.2.1.2 Complex traversal queries

This part focuses on more complex queries for methods mentioned in the part 1.5.2.1 of the current merging process. In Titan, these queries were done imperatively by recursively retrieving neighboring nodes. The same thing could be done while using embedded Neo4j, but using Cypher to match patterns is more efficient.

The goal is to create general Cypher queries which would work for all possible types of input nodes.

- **Retrieval of node's resource** In the merging process, the only retrieval of *node* label occurs, however in other modules even querying for the resource of other underlying types (such as node attribute) might be required. In Cypher it is possible to quantify the number and types

of relationships in the matched pattern. This property allows for simple traversal.

Consider the example depicted in the figure 2.1 and the Cypher query in the listing 2.6, where the task is to retrieve the resource of nodes having various labels.

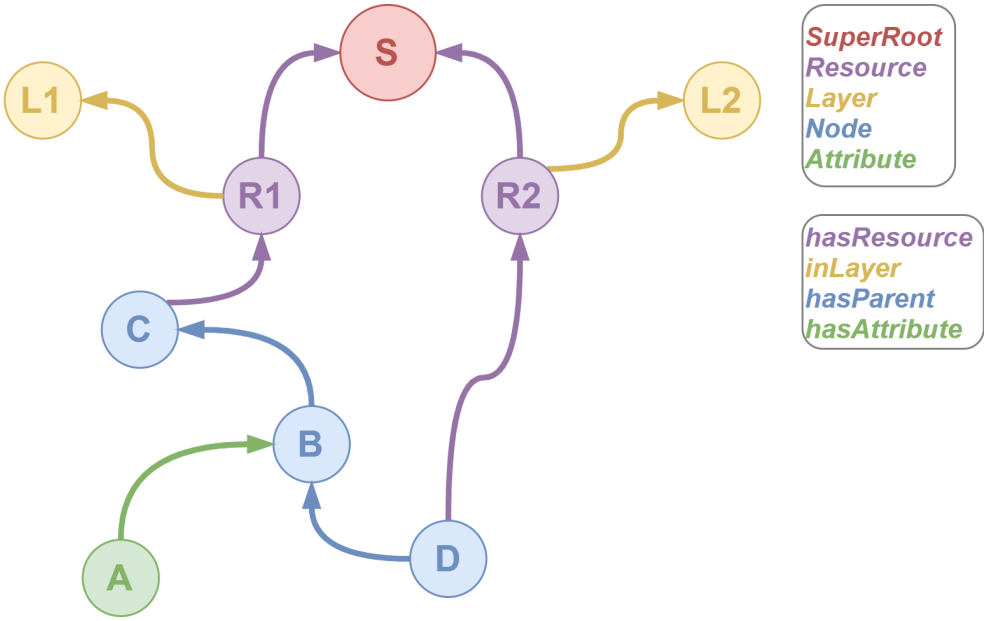


Figure 2.1: Example for resource retrieval

```
MATCH path=(node)<-[:hasAttribute*0..1]
      -()-[:hasParent*0..]
      ->()-[:hasResource]->(resource)
  WHERE ID(node)=···
RETURN resource
```

Listing 2.6: Retrieval of a resource of certain node

There are 3 cases that need consideration:

- *Attribute* Consider resource's retrieval of a node A, which has *Attribute* label, the first relationship pattern will match it - as it expects up to 1 *hasAttribute* relationship. Then it expects arbitrary sequence of outgoing *hasParent* relationships. Based on the figure 2.1, 2 *hasParent* relationships will be matched – as node B and node C are on the path to resource.
- *Node* Retrieval of the node having node label works exactly same as in the attribute's case, with the difference that no hasAttribute relationship will be matched by the pattern.

- *Node with different resource* In the scenario where a node has a different resource than its parent, multiple resource nodes would be matched. To select the correct one, it is important to limit the size of the result to one, but also sorting of matched paths is required. This sorting ensures that the resource from the shortest path is returned (see listing 2.7).
- *Resource* If the node's label is a resource, it is a bit tricky as it also has an outgoing hasResource relationship connecting it to *super root* node, meaning the *super root* node would be returned if used query as-is. To deal with this conditional return has to be added – either checking for the label of retrieved resource variable or checking for the label of matched input node (see listing 2.8).

```
...
RETURN resource
ORDER BY LENGTH(path) ASC
LIMIT 1
```

Listing 2.7: Sorting result and limiting its size

```
...
RETURN CASE
  WHEN 'Resource' in labels(node)
    THEN node
    ELSE resource
  END
as resource
```

Listing 2.8: Conditional return

Another very similar use case is retrieving the layer of the node, which is very similar to discussed cases with the addition of *inLayer* relationship and layer node variable to the pattern.

- **Getting node by its path from the super root** The input of this task is a list of elements containing the resource's name, name, and type. The original method in Titan for this retrieval was iteratively retrieving children of the node vertices starting from the resource vertex.

  The original implementation is as follows. After retrieving children of a given vertex (node or resource), for every child, its resource is fetched for further comparison with the initial resource. Then for each child, its name and type are compared with the input properties and the fetched resource's name with the initial resource. By this comparison, the children are split into 2 groups – exact matches and case insensitive matches. After every child is evaluated exactly one child is selected for further evaluation, where the group of exactly matched children takes precedence.

However, if the group, from which the child is selected, contains more than 1 node, an exception is thrown as that is an invalid state.

In the Cypher pattern can be created to match the path based on the list from the input. There are two approaches to creating the required query pattern, each starting from the super root node.

The first approach is to create the pattern concatenation of the matched parts, as shown in the following example. The pattern must be supplemented by the correct revision properties of all control edges in the `WHERE` clause. To make sure the paths contain both exact and case insensitive matches the properties of the nodes need to be included in the `WHERE` clause as well (example shown in the listing 2.9).

This solution however has a few drawbacks. Its readability is not very good as the final query would only be essentially glued from multiple parts. Another problem is that due to the query not being parameterized, the potential for query caching is not there. The last problem is also related to parametrization, as there is a risk of database injection attacks due to creating queries this way.

```
MATCH paths=(superRoot)<-[r1:hasResource]
    -(:Resource {resName: 'resA', resType: 'resTypeA'})
    <-[r2:hasResource]
    -(:Node {nodeName: 'nodeA', nodeType: 'nodeTypeA' })
    <-[r3:hasParent]-···
    WHERE r1.tranEnd >= 1.0 AND r2.tranStart <= 2.0 AND ···
```

Listing 2.9: Matching full path by string concatenation

The second approach is to directly pass the input list into the query as a parameter. This makes query caching possible and is much more readable. Every matched path can be iterated and each value compared insensitively (with the input list) in the `WHERE` clause as a part of `ALL` predicate in the query using the `TOLOWER`[2] and `COALSCE`[3] statements.

In the initial solution, for every node in the retrieved path, the resource is retrieved for comparison with the resource in the path to ensure that no node with a different resource is selected. This can be a very expensive operation if performed separately, but in Cypher the addition of retrieving each path node's directly connected resource is much more effective. The listing 2.10 illustrates the described query.

```
WITH $inputList AS path
  MATCH p = (superRoot)<-[:hasResource]
        -(res)<-[*SIZE(inputList) - 1]-()
```

[2] `TOLOWER` converts strings to lower case
[3] `COALESCE` converts null values to predefined values

```
  ...
  WHERE TOLOWER(COALESCE(NODES(p)[idx][$nodeName], ''))
    = TOLOWER(path[idx-1][0])
  AND TOLOWER(COALESCE(NODES(p)[idx][$nodeType], ''))
    = TOLOWER(path[idx-1][1])
  ...
  UNWIND NODES(p) AS n
    MATCH resPaths = (n)-[:hasResource]->(r)
RETURN p, COLLECT(resPaths) AS nodeToResourcePaths.
```

Listing 2.10: Matching full path by input array

- **Updating node's subtree** Method `setSubtreeTransactionEnd` updates the ending revision property for each control relationship in the node's subtree, as well as for all adjacent relationships of the subtree. To achieve this functionality with Cypher, consider listing 2.6, where the path to resource from the node is matched. If the pattern was reversed in a way that paths from the node to the attributes were matched, matched paths would form a subtree of a node. After matching the subtree, all that's left to do is to update every relationship of every node belonging to the subtree. This can be done using `UNWIND` statement (see listing 2.11 for the full example).

```
MATCH path=()<-[:hasAttribute*0..1]-()-[:hasParent*0..]->(node)
  WHERE ID(node)=...
UNWIND NODES(path) as nodes
MATCH (nodes)-[relationships]-()
SET relationships += $property
```

Listing 2.11: Updating subtree of a node

2.2.2 Merge

Cypher query language provides `MERGE` statement which could, in theory, be very useful for the merging algorithm as it checks for a given pattern in the stored graph and creates it only if it did not exist.

In a concurrent environment, however, a race condition can occur as `MERGE` only combines `MATCH` and `CREATE` statements and has no way of knowing what changes are done in the database by a different transaction. To prevent this unique constraint can be used which ensures that no two same objects are created concurrently. Constraint, however, induces performance penalty. For performance reasons, creating a schema index on the label or property is highly recommended when using `MERGE` [25].

The versioning of the graph using the relationship produces another big problem, as the versioning is done using relationships, there can be multiple nodes of the same properties in the same position in the graph (under different revisions), which means that to use `MERGE` properly in a concurrent

environment, each of nodes would need to contain unique constraint on the combination of revision and its hierarchy position (e.g. full path from super root), which is highly impractical memory-wise.

2.2.3 Graph equality testing

To ensure that the migrated merging algorithm works correctly, some sort of evaluation technique is required. In the current MANTA, it is possible to export the whole Titan graph into the dump file, which can then be imported to the database. To ensure that the algorithm works correctly, this imported graph has to be compared to the graph created using the migrated algorithm.

The problem of determining equality between 2 graphs is called the exact graph matching problem or graph isomorphism problem. There are various algorithms that can be used for solving this problem, most of which are based on depth-first traversal of the graphs and finding appropriate matching. For example, the VF2 algorithm [26] uses a state-space representation and is based on a depth-first strategy while employing the feasibility rules to ensure consistency of the explored states.

However, in MANTA, there are some inconveniences that make it impossible to directly compare the graphs as some elements are interconnected not only by the graph edges but also by the properties. The first problem is that the node attributes with a key *sourceLocation* contain a source code identifier which is a UUID, which would result in the structurally same graphs having different values. The other cases of this interconnection are the flow edges, which contain *targetId* property representing the database identifier of that target node. The original database identifiers from the Titan database are also exported, so the mapping of these properties can be easily created, however, to solve the first problem, mapping of the source codes has to be created.

By taking advantage of the MANTA graph structure, simple algorithms to determine equality can be implemented. This subsection describes the use of the traversal framework and Cypher query language to compare 2 MANTA graphs.

2.2.3.1 Traversal framework and Java Core API

The deprecated traversal framework still works and can be used to compare the graph in conjunction with Java Core API. As the MANTA graph contains 3 subgraphs, the source subgraph, the revision subgraph, and the main metadata subgraph, each has to be compared.

To compare the source and the revision subgraphs Java Core API can be used to retrieve children of source root and *revision root* nodes respectively. After the retrieval, the mapping can be created by comparing the properties of each node.

The idea of the metadata subgraph comparison is to use a traversal framework to collect all paths from the super root node to the leaf nodes using the hasResource and hasParent relationships. The problem however is that the criteria determining the order of the evaluation of the traversed nodes can not be specified, resulting in a nondeterministic order of the paths, hence they can not be compared directly. Therefore, a mapping of each node has to be created, which can be very computationally expensive as each of the paths has to be compared with other paths. To improve performance the paths can be grouped by their length and starting and ending nodes. After the mapping of all nodes is finished, all the attributes and relationships of each node have to be compared. But this can easily be done by using Java Core API and retrieving the adjacent nodes and relationships and comparing all their properties. To solve the problem of node attributes of key *sourceLocation*, the mapping for each source code has to be created and then used when comparing node attributes of the mapped nodes.

2.2.3.2 Cypher

The main advantage of using Cypher directly instead of the previous solution is the ability to chose the order in which traversed nodes are returned as specifying the sorting order inside Cypher query is possible. This ensures that no path-to-path comparisons are required and each node is only traversed once. To compare 2 traversed graphs they have to be traversed and compared concurrently or the textual representation of each graph can be created while traversing and then the representations can easily be compared.

Once again the 3 subgraphs are compared separately with the exception of the source code nodes due to them being referenced by attribute nodes. As each source code node contains the hash code of the stored file, the *attributeValue* inside the attribute nodes can be replaced by the hash of the given node before saving the textual representation resulting in the equality of the compared attributes referencing the same source code nodes in different graphs.

The traversal of the metadata subgraph has to be done starting from *super root* node continuing by traversing hasResource and hasParent relationships to the leaf nodes to prevent ambiguity problems. Other relationship types would be traversed as well so the correct representation of the graph can be created, however, the relationship traversal can cause ambiguity problems as depicted in the figure 2.2, where the string representations of the relationships `A->D` and `B->C` are ambiguous due to `C` having same properties. These relationships which are candidates of causing ambiguity have to be resolved to ensure graph equality.

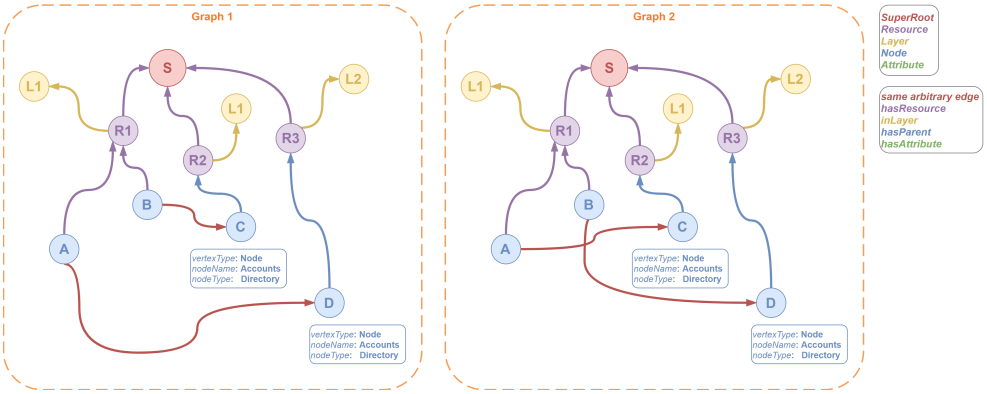


Figure 2.2: Edge ambiguity problem

2.3 Improvements

This section focuses on performance improvements of the migrated merging algorithm. The goal is to improve the performance of the re-implemented algorithm in Neo4j. The first subsection discusses minor optimization steps which will guarantee a performance boost in certain situations. The second and last subsection contains information about user-defined procedures within Neo4j and the possible use of them to improve overall performance. The results of various approaches will be discussed in the last chapter (see chapter 4).

2.3.1 Minor optimizations

This subsection describes minor changes of the merging algorithm, which could increase its performance. Most of the proposed changes are caching-based to trade-off a little of memory for better processing time.

- **Retrieving latest revision number** Retrieving a number of the last committed revision happens during the merging of every object type. Originally, this number is retrieved by querying revision root in the database and getting its *latestCommittedRevision* property. This, however, creates redundant database calls even if it is cached. Fetching this number only once during the initialization of the merging process and saving it in the contextual object, from which the processing is able to retrieve it as needed, would reduce thousands of database calls to just one.

- **Node identifiers instead of nodes** Originally, when processing each type that requires previously processed nodes (e.g. source and target nodes when creating a relationship), the required nodes are retrieved from the database. The properties of the retrieved nodes are rarely

needed for processing. Therefore, retrieving these nodes is often redundant and only the way to access them in the database is needed, so the relationships and other nodes can be added.

Database access insurance can be achieved in two ways – caching the entire node objects or only their database identifiers. The advantage of storing entire nodes is that in case of needing to process their properties they are directly available. However, it also comes with a memory disadvantage and could cause problems when large merge input is processed. Caching of database identifiers seems like a more suitable approach as the memory usage won't be as high and the need for properties to process is minimal.

- **Newly added nodes** When processing every object type, the database is always queried for the previous existence of the merged object to determine whether to create a new object or to update its revision. Consider merging a node, the parent of the node is queried for its children, however, if the parent was freshly created within current merge processing, this operation is unnecessary as there can not exist any children.

  Knowledge of which nodes are newly added can be used during merging processing in multiple cases – when merging a node, if its parent is newly created, it is safe to assume that the currently merged node could not have existed, therefore there is no need to query its parent for its previous existence. When merging relationships, if either source or target nodes are missing, querying for previous existence is unnecessary. This would work similarly with processing node attributes, as they could not have existed if the owning node was newly added. Caching newly added nodes only helps if there are new merged nodes in the given merge request.

- **Node to resource mapping** Given node can share the same resource as its parent or have a different resource. When the new node is being created, a comparison of its resource and its parent's resource is done to determine the fact. Originally this is done by retrieving the resource of the parent by traversing the graph upwards which can be very costly in deep graphs. As the node hierarchy is merged gradually, all the nodes are processed with their resource, therefore storing these relationships and later using them for comparison is much more efficient than traversing the graph.

2.3.2 User-defined procedures

Neo4j enables implementing and using user-defined procedures which might help performance-wise, mainly with the communication overhead, as each merge request can contain thousands of objects, where each translates to a number of database statements.

User-defined procedures have their context injected from the Neo4j server. Context can be either a single transaction or database service, which can be used to create multiple transactions allowing further optimization by making use of multiple processor cores.

These procedures have to be written in Java and built into a `.jar` library. Then, this library can be inserted into the Neo4j plugins folder. Each procedure inserted this way can be run using Cypher's `CALL` statement (see listing 2.12).

```
CALL custom.procedures.userDefinedProcedure(1000)
```

Listing 2.12: Calling user-defined procedure

Each procedure can have an arbitrary number of parameters of supported types, which are some of the basic Java types as well as some of the Neo4j types such as `Node`, `Relationship` or `Path`. Lists and maps containing supported types are also allowed. Therefore, each merge request has to be passed to the user-defined procedure to be processed. This can be done by storing the request into the list and passing the list as a parameter.

The Neo4j internal interface provided for implementing the user-defined procedures is a bit different than the interface provided by the Neo4j Java driver for client-side applications. This means that to implement the algorithm as a procedure, each component containing Neo4j driver classes has to be rewritten. Internal interfaces are defined in the package `org.neo4j.graphdb`, while the driver's interface is defined in the `org.neo4j.driver` package.

The differences are usually pretty minimal, for example, the retrieving of the identifiers of certain entity can be done directly by using `entity.id()` in driver's interface but syntax `entity.getId()` is required while using internal interface. Another difference is working with the result of the transaction, where the driver's interface provides easier access to the retrieved objects (see listing 2.13 and 2.14, where the entity has to be explicitly converted)

```
Result result = tx.run(···)
if (result.hasNext()) {
    return result.single().get("resource").asNode();
}
```

Listing 2.13: Returning a node from the transation's result using driver interface

```
Result result = tx.run(···)
if (result.hasNext()) {
    return (Node)result.next().get("resource");
}
```

Listing 2.14: Returning a node from the transation's result using internal interface

2.4 Parallelization

This section discusses the introduction of parallelism to the migrated merging algorithm to enhance its performance. Only the shared memory parallelization is taken into account as the processing is performed on a singular machine where MANTA runs (alternatively on the machine where the Neo4j server runs).

As mentioned in the description of the merge request format (see part 1.5.1.2), there are various merge types. Processing of different types can be split into phases which can be processed in parallel. The first subsection describes the locking properties of the Neo4j entities, followed by a detailed description of the multiple approaches to parallel processing of specific object types.

2.4.1 Locking properties

Each transaction acquires locks on certain entities upon accessing them. In Neo4j, there are 2 types of locks – shared lock, acquired when reading an object, and exclusive lock, acquired when modifying the object. Each of these locks is bound to a certain Neo4j entity, either node or relationship.

Shared locks are taken when we want to read something and at the same time prevent other transactions from writing to, or otherwise modifying that object.[27] These locks are released upon finishing the query execution. Each of the shared locks can be held by an arbitrary number of transactions at a given time.

Exclusive locks, on the other hand, can be only held by a singular transaction to avoid problems of concurrent writes and are released only upon committing or rollbacking the owning transaction. When creating or removing a relationship between 2 nodes, an exclusive lock is acquired on both of those nodes. On the other hand, if a relationship is only edited, the lock is acquired only on the given relationship.

2.4.2 Initial approach

The initial approach proposes the parallelization solutions for each of the object types based on the locking properties of Neo4j. The implementation of these solutions is later described in the chapter 3.

2.4.2.1 Source codes, resources, layers

Each merge request contains these 3 types, but only a few instances of each are merged, hence the parallelization of this part is not required.

2.4.2.2 Nodes and node attributes

This part describes the method of splitting the nodes to allow parallel processing using multiple threads as well as a method of parallel graph traversing.

The nodes are processed in a way that a parent of a node (or resource) has to be processed before its children. By finishing the processing (merging and committing the change) of the children belonging to a given parent node, all the underlying subtrees of the processed children nodes can be further processed. Therefore, the parallelization of the node processing is equivalent to the parallel breadth-first traversing, where the processing of each element comes with a database overhead as the database is queried and new vertices can also be created.

In the specific case of breadth-first traversal of the graph database with the occasional creational queries, it is important to distribute the traversed graph in a way that the locking of related nodes is minimal to none. Due to the structure of the MANTA metadata graph, this state is achievable by ensuring that the processing of a certain node means processing all its children (hence no lock contentions would apply to the given node as no 2 threads would create nodes connected to the same node within their distinct transactions). Due to the structure of the merging request (see part 1.5.1.2), the preprocessing of nodes is very straightforward as only the mapping of nodes to its children has to be created and later used to traverse the graph.

The problem with using the suggested solution is that a node can sometimes have both parent and resource nodes (see part 1.2.2 of the model description), which could create lock contentions as different threads might try to create relationships connected to the same resource using their transactions. To prevent database locks or possible deadlocks, the definitions of these relationships have to be collected and processed sequentially only after all nodes from the merge request have been processed.

To make the described processing possible shared synchronized queue needs to be available for all the participating threads. This queue would contain all the nodes which are ready to be processed. To lessen the synchronization overhead, each thread would also need an additional internal queue from which it would primarily consume the work, and only if this queue would be empty it would try to get the next work from the shared queue. Upon all working threads depleting their internal queues and the shared queue, the processing would proceed to create stored node-resource relationships.

The following list describes possible scenarios of merging the nodes in parallel as described. The example of the processing is depicted in the figure 2.3.

- Shared queue is initialized with the already processed resources.
- Each thread tries to retrieve nodes primarily from their internal queues or secondarily from the shared queue.

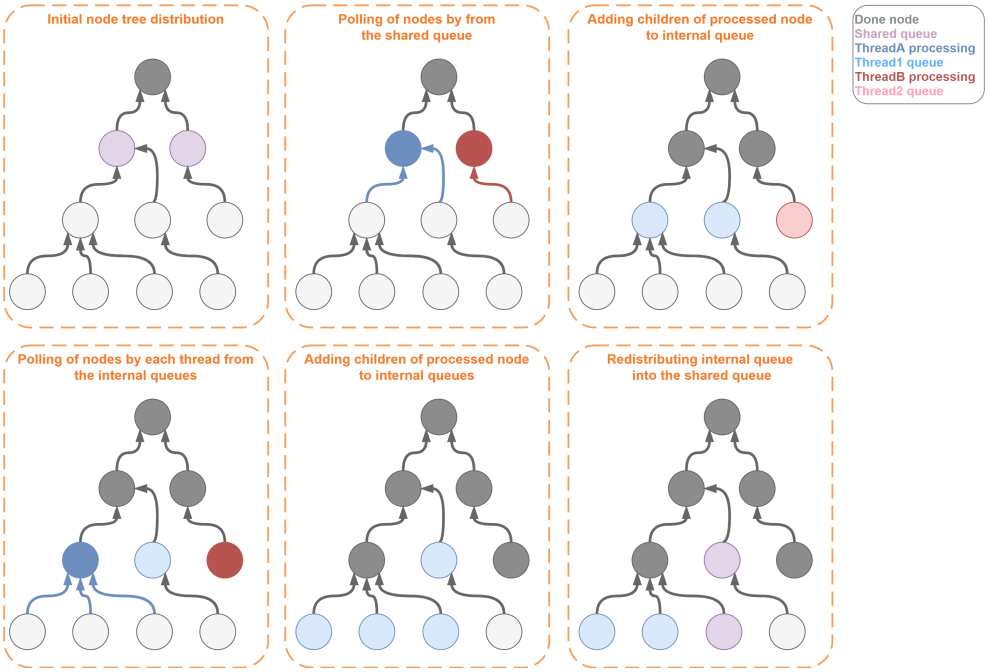


Figure 2.3: Parallel node processing

- If a thread has an empty internal queue and the shared queue is also empty, the thread is put to sleep.
- Each thread processes all children of the node they retrieved from the queue and inserts all children into their respective queues.
- After processing a specific node and inserting its children into the internal queue, the thread commits its transaction and redistributes its internal queue into the shared queue in case of any sleeping merging thread. This is done to ensure that no thread is idle for a long time.
- When the sleeping thread is notified it tries to retrieve the next node from the shared queue and proceeds with the work.
- After all objects from the shared queue and all internal queues are depleted the node processing can proceed to the next phase

2.4.2.3 Node attributes

As each of the node attributes is connected to a certain node all the attributes can be split by the nodes they belong to and be processed in parallel using multiple threads. This split is done to avoid multiple threads modifying the same nodes which could potentially lead to lock contentions. As mentioned in

2.4.1, write lock acquired during by transaction is held until the transaction is committed (or rollbacked), hence the requirement of the same locks by multiple transactions would essentially serialize the processing or in the worst case cause a deadlock and require rerunning of the processed part.

Processing of the special attribute *mapsTo*, which creates a relationship to a different node, could cause a lock contention in case of different transactions processing *mapsTo* attribute and creation of a relationship between conflicting nodes. Due to this, *mapsTo* attributes have to be processed at the end of this step by a single transaction.

The figure 2.4 depicts the distribution of the node attributes for 2 threads. It can happen that the distribution will not be balanced in the case of very large differences of the attribute count for different nodes (see figure 2.5), but in the reality, this should not happen.

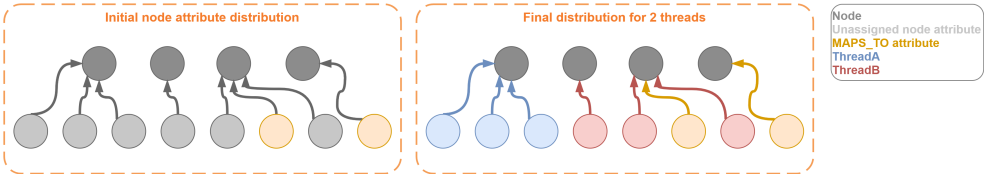


Figure 2.4: Parallel node attribute processing

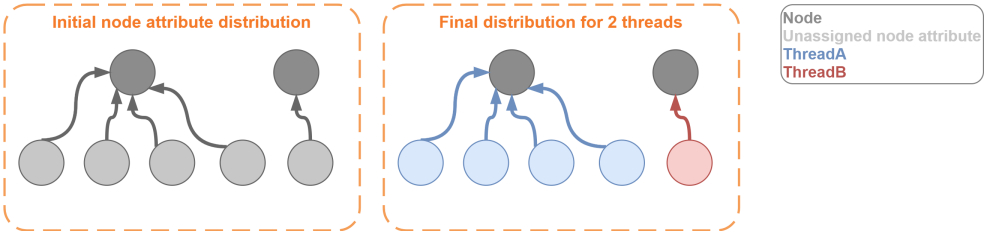


Figure 2.5: Parallel imbalanced node attribute processing

2.4.2.4 Edges and edge attributes

Using a shared queue to merge edges in parallel by multiple threads could possibly cause locking contention, especially if the number of edges is high. It is the same problem as with node attributes described in 2.4.2.3, but in the case of edges, 2 nodes are affected per each edge object.

Therefore to ensure lock-less distribution of the edges, the edge graph needs to be partitioned into subgraphs, where the numbers of nodes in each partition are similar in size. These subgraphs could then be separately processed by different threads.

There are various algorithms that can be used to split the graph. For example, Kernighan–Lin [28], or the Fiduccia–Mattheyses algorithm [29] use

heuristics to partition the graph into a bipartite graph, however, both require initial preprocessing of the nodes. This would require continuously applying the algorithm in case of more than 2 partitions needed. In this thesis, only the simple iterative approach of partitioning is used and is described followingly.

The proposed solution for the node attributes was to split them into containers by a number of threads. The same method can be used for processing edges, however, it is a bit trickier as the edges need to be split by the pair of source and target nodes. The way that could be achieved is to iterate the list of edges sequentially and split it into the containers by their source and target node identifiers. The splitting would be done in a way that a mapping of each container to the nodes of the edges assigned to it would be maintained. Edge's affiliation would be determined by the ownership of its source or target node.

If both source and target nodes of a certain edge are assigned, but to different containers, the edge can not be processed in a given batch, if no node is owned by any container it would be assigned by an assignment mechanism – round-robin, smallest-first or random fashion. After the splitting is done (no more edges to preprocess), the size of the largest container is reduced to match the size of the second-largest container to make the processing is more balanced. This splitting can be repeated recursively on all the remaining edges (which could not be assigned to any container due to conflicts) until a certain threshold condition is met (the count of remaining edges is too small or the sum of edges in split containers is too small). This further reduction should boost the performance as more threads will be allowed to run in parallel for a longer time.

Another problem is that certain edges are of *perspective* type, which can create both relationships and node attributes, potentially causing a lock contention. The solution to this problem is the same as with *mapsTo* node attributes – store them and process them sequentially after all the batches of edges are processed.

The following list describes possible scenarios of splitting the edges into 2 containers. The figure 2.6 depicts these scenarios and the figure 2.7 shows a small example.

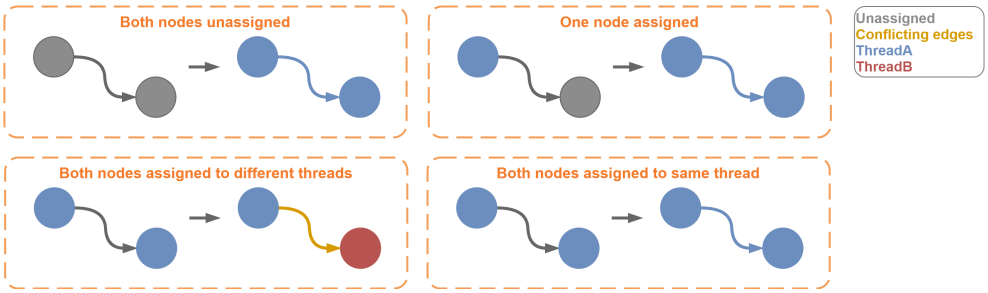


Figure 2.6: Edge distribution

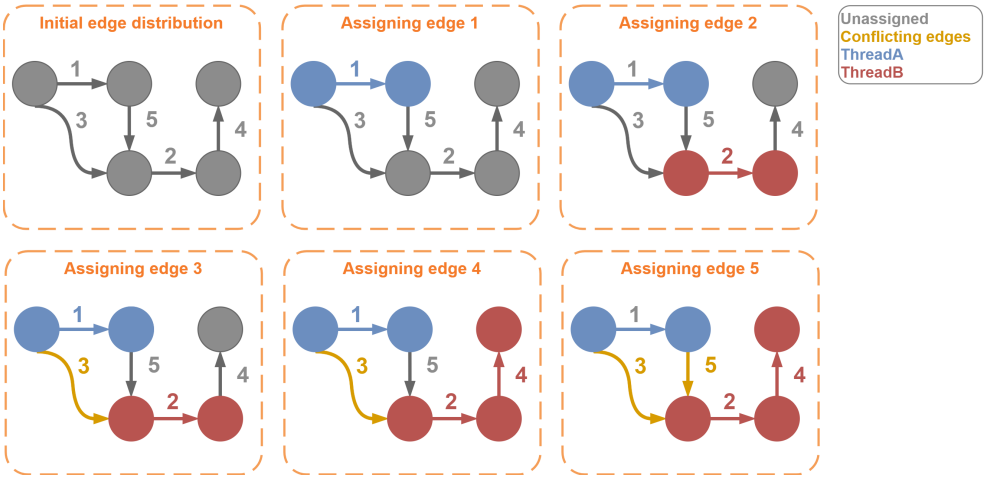


Figure 2.7: Edge distribution example

- In the case of both nodes of an edge being unassigned, the edge and the nodes are assigned by the assignment mechanism.
- In the case of one node being assigned and the other one not, the edge and the unassigned node are assigned to the same container as the assigned node.
- If both nodes are assigned to the same container, the edge is be assigned to it.
- If the nodes are assigned to the different containers, the edge is conflicting and can not be processed.

Edge attributes can be processed in the same way as edges, but as the number of these objects is usually small, they can be processed with the edge they belong to.

2.4.3 Locking minimizing solution

After the initial implementation of the proposed solutions, a problem has occurred – when splitting the node attributes and the edges only by nodes as described in parts 2.4.2.3 and 2.4.2.4, deadlock would sometimes occur.

After analyzing the cause of these deadlocks, it turned out that the reason was that the write locking a node to create a relationship also locks all its adjacent relationships and hence causing undesired blocking and sometimes even deadlocks. The figure 2.8 demonstrates the described situation, where the concurrent threads could be blocked on the relationship **1** or the relationship **2**. This issue does not happen when processing nodes, because the situation causing lock contention can not occur there, but it can happen in all

other phases of parallel processing. A simple solution is to retry processing upon deadlock while reducing the number of objects processed within a single transaction (so the retry does not take too much time), which would solve the deadlock problem, but as the deadlocks are caused by writing locks, blocking of processing would be still present. The following parts of this subsection describe the way to avoid locking problems caused by the previous solution.

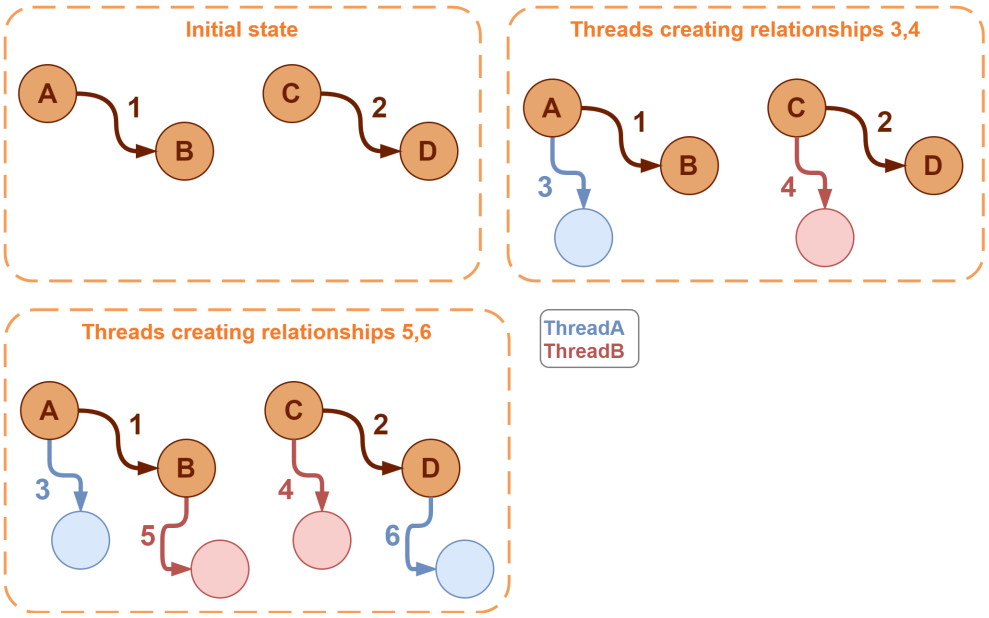


Figure 2.8: Deadlock inducing behavior

2.4.3.1 Nodes and node attributes

As mentioned previously, this problem can not occur while processing nodes, as it only occurs when creating new relationships and if the new child of a certain parent node is added there could not have been any relationships bound to the new node which could cause any lock contentions and as all children of the certain parent node are processed by a single thread no contention of parent's lock is.

The solution to reducing node attributes locking is to process node attributes together with the nodes they belong to during the node processing part. This would however require storing the *mapsTo* attributes along the relationships connecting nodes to resources and then processed sequentially.

2.4.3.2 Edges and edge attributes

To avoid locking issues when processing edges, another constraint needs to be added to the process of splitting edges into containers. As the issues originate

from the adjacent edges of the nodes processed, they need to be taken into account.

To make this work, all neighboring nodes of all processed edges need to be collected so they can be used during splitting. This operation can be very expensive if the sizes of the whole graph and merge requests are large. In the case of splitting the edges into several iterations, all previously added edges have to be counted as well.

The following list describes possible scenarios of splitting the edges into 2 containers while also taking into account their neighborhoods. The figure 2.9 depicts these scenarios.

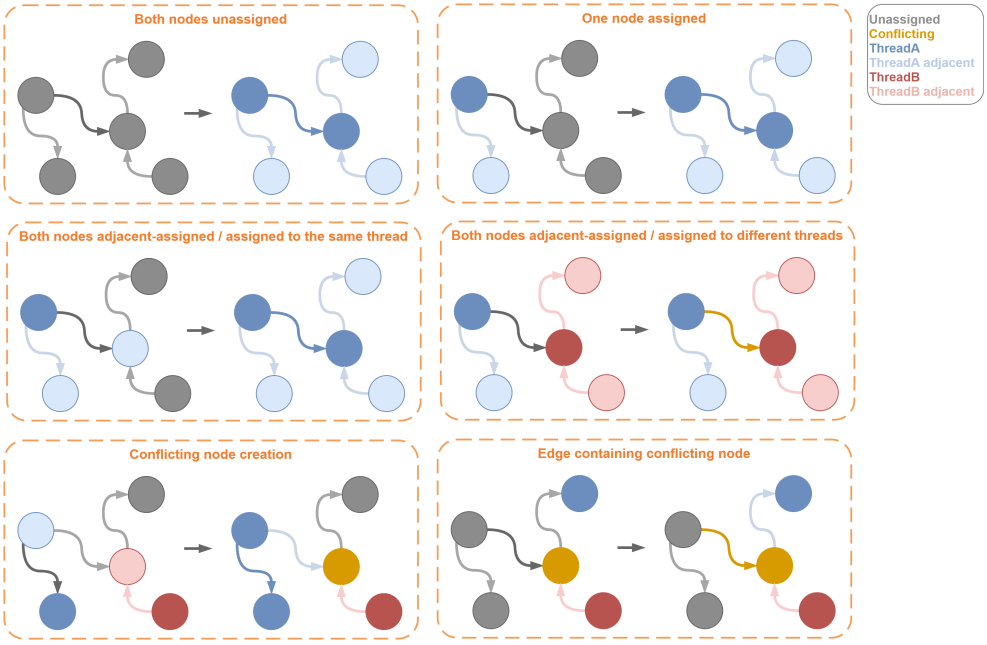


Figure 2.9: Edge adjacency-aware distribution

- If both nodes are unassigned, they are assigned to a container by the assignment mechanism. Every adjacent node of the source or the target node is adjacent-assigned to the same container.

- If one node is assigned and the other one is unassigned, the unassigned node with the edge is assigned to the same container as the assigned node.

- If both nodes are assigned to different containers, the edge is deemed conflicting and is evaluated in the next batch.

- If the adjacent node of a newly assigned edge already is adjacent-assigned to a different container than the edge's, the node becomes the conflicting node.
- If either of the nodes is conflicting, any edge connected to it is also deemed conflicting and can not be assigned within a given batch.

2.4.4 Transaction-retry improvements

This part discusses possible enhancements to the transaction-retry solution. As the locking and deadlocks happen while using this approach, it is possible to mitigate it by splitting node attribute processing and edge processing into non-blocking and potentially-blocking phases.

Non-blocking phase would contain reading operations, to determine what needs to be done and writing operations, which do not cause lock contentions. All operations potentially causing lock contention would be stored inside supplementary structure and evaluated in the following potentially-blocking phase.

2.4.4.1 Node attributes

Creating a new node attribute is a potentially-blocking operation as a new relationship is created between the owning node and the new attribute node. However, if only the revision of the control edge is updated, only lock on the given relationship is acquired which will not cause lock contention as no other transactions will access the relationship.

In the non-blocking phase, it is beneficial to split this phase into two parts. The first part would process only *mapsTo* attributes split evenly into containers based on thread count. The reason for this division of the non-blocking phase is simply the fact, that processing of the *mapsTo* takes much more time as the graph needs to be traversed, so by doing this more expensive operations are evenly distributed to all merging threads. If it is determined that a new relationship of type *mapsTo* needs to be created, it is stored and later evaluated within the potentially-blocking phase of edge processing. The second part of this phase is processing other types of node attributes. As this is non-blocking phase, any creational queries are stored for later evaluation, while the relationship updating operations are evaluated directly.

All previously stored node attribute creational queries are split in the same way as in part 2.4.2.3 to further reduce potential conflicts as well as reduce the processing time if the retry is needed.

2.4.4.2 Edges and edge attributes

This part can be done similarly to the node attributes processing as the edges are either updated or created as well. The special type *perspective* is a par-

allel to the *mapsTo* attribute, as it also requires longer processing due to the retrieving of the layer as well as node attribute creation.

Non-blocking phase in this case would be divided into two parts, where the first part deals with more expensive *perspective* edges, and the second part processes the rest of the edges. Once again, the potentially-blocking operations are stored for later evaluation.

In the following potentially-blocking phase, all stored potentially-blocking edge operations (collected in the previous phase, but also during *mapsTo* node attribute processing) would be split into node containers recursively and then processed in parallel.

2.4.5 Merge request unification

Currently, implemented merger only processes single input request at a time, while other requests need to wait for their turn. Unifying multiple merge requests together (possibly in parallel) and then merging the created subgraph to the database could further reduce processing time as multiple duplicities would be removed. This would also affect parallel merging as more separate subtrees might be contained within the merged file, making the splitting of the nodes and edges could be more exclusive, reducing locking contentions. This subsection discusses needed synchronizations and the process of merging the merge requests.

2.4.5.1 Multiple type synchronization

As there can be multiple types of merging algorithms available (parallel, sequential, client, server, and maybe even more in the future), the request needs to contain the type (or would be provided with a default). With the merge requests of various types, it makes sense to only unify requests of the same merging type, hence a certain synchronization mechanism is required.

To synchronize this part, a barrier containing type needs to be maintained. If no merge request is being processed, the first incoming request would set the type of the barrier to the value of its own merging type. All the following merge requests of the same merging type would be allowed further, therefore possibly be merged together. Requests of different types are put to sleep and wait for the release of the guard's type. The different scenarios are depicted in the figure 2.10.

2.4.5.2 Request unifying synchronization

After merge request passes the barrier, another synchronization to control submitting the requests to the database as well as unifying waiting requests is required. The idea is to ensure that the request proceeds to the database merging phase when the database is available. If the specific request is being merged into the database, therefore making the database unavailable, all

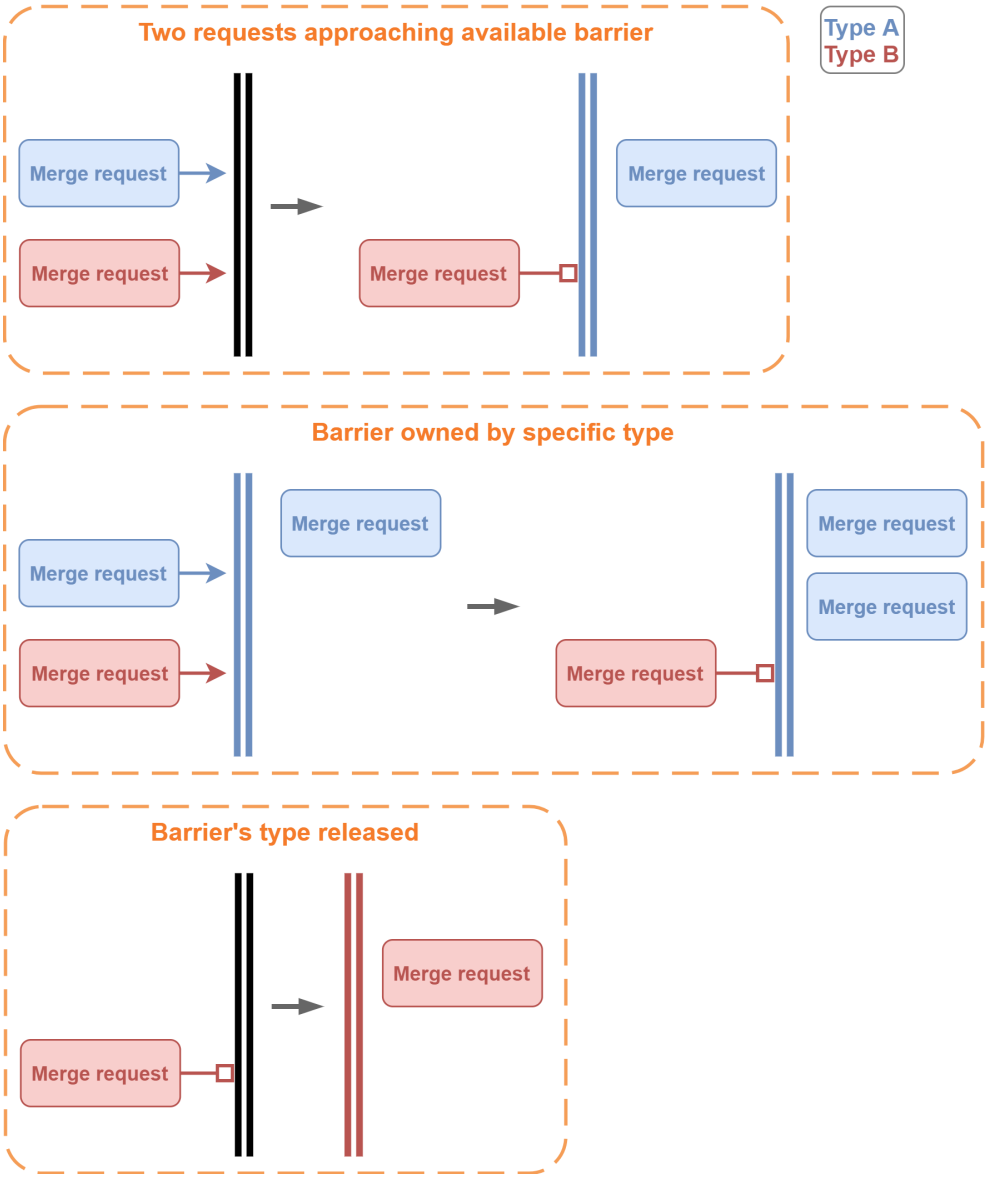


Figure 2.10: Request type barrier

other requests waiting for the release of the database are unified together and submitted to the database upon the completion of the merging. The figure 2.11 shows the synchronization process in detail.

2.4.5.3 Request unifying process

To unify multiple merge requests mimicking the database merge process is needed. This can be achieved by using a global structure containing all merged

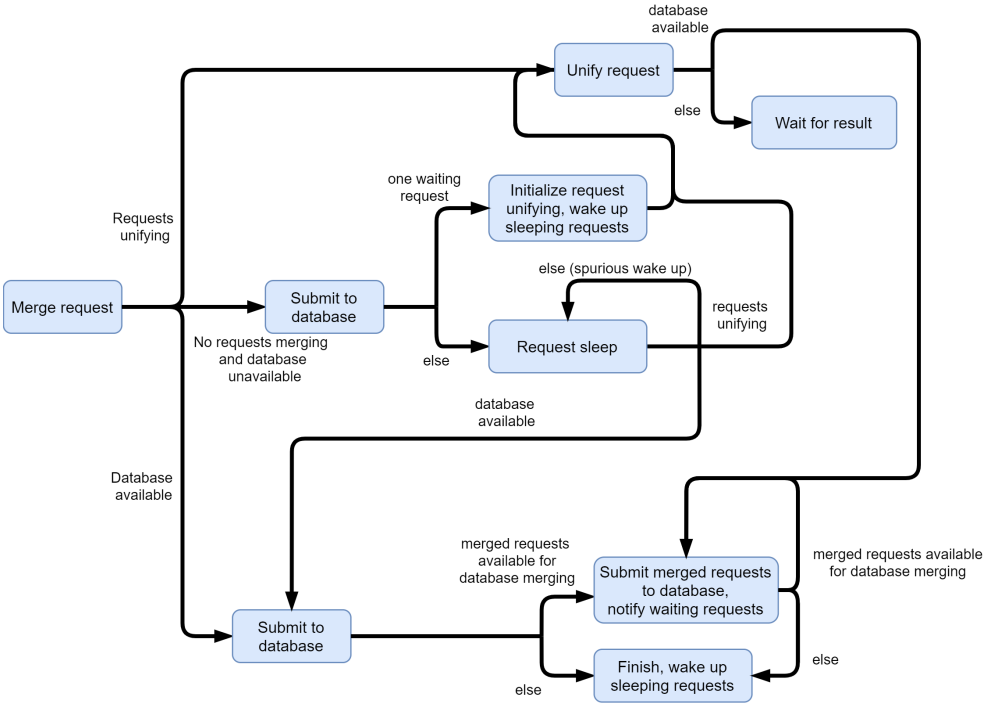


Figure 2.11: Request merge/unify process

objects serving as the database, while all merged requests would merge their objects into it. Each of the requests would manage their local contextual structure containing mappings of their local objects to the merged global objects. Synchronization needed for each type is as follows:

- **Source code** Source codes require mapping of their local identifiers to global identifiers as these identifiers are sometimes referenced by node attributes as their value.
- **Layer** To ensure only one layer representing the same layer exist in the merged structure, properties of the layer object have to be used – *layerName* and *layerType*.
- **Resource** As each resource belongs to a certain layer, the global identifier has to replace the local identifier of the layer. Resources are then merged using their *resourceName* and *resourceType* as well as affiliation to the certain layer to ensure that no two same resources are created.
- **Node** To merge node objects, the mapping of a node to its children needs to be maintained as each node is linked to a certain parent and resource. Properties *nodeName* and *nodeType* are used to differentiate different nodes of same parent.

- **Node attribute** Properties *attributeKey* and *attributeValue* are used to differentiate attributes of the same owning node. Owning node's global identifier is retrieve for correct mapping.
- **Edge** To differentiate edges of the same source and target nodes, its type has to be used. The direction is already given by the source and target division.
- **Edge attribute** This type is merged in a same way as node attribute, using the *attributeKey* and *attributeValue* properties to differentiate different properties of the same edge.

The goal of this chapter was to analyze possible enhancements, which could increase the overall performance of the merging algorithm along with the possibilities of migrating specific parts needed for merger to work correctly. The first section contained the analysis of parallelization within Titan graph, followed by the analysis of querying operations needed for migrating the algorithm. The discussion of database queries was followed by analysis of possible optimization steps.

Chapter 3

Realization

This chapter describes the implementation of the proposed changes. All of the changes were implemented in Java. The implementation sections reflect the sections of the chapter 2, namely the Titan parallelization, migration of the merging algorithm, improvements, and the parallelization of the migrated algorithm.

3.1 Titan parallelization

This section is divided into 2 subsections, one containing implementation details of the singleton transaction synchronization and the second one describing the locking used in the merging algorithm.

3.1.1 Transaction synchronization

This part describes the implementation of the synchronization for the singleton transaction as per section 2.1. After reading the object from the input stream it is sent to the singleton object of class `MergerTransactionHandler`. This object handles all submissions of objects to be merged.

The `MergerTransactionHandler` singleton contains `BATCH_COMMIT_COUNT` property to determine when the transaction should commit. Then it contains `batchesBeingProcessed` and `batchesDoneProcessed` to ensure that all submitted work is done. The singleton also needs to keep track of all the requests, which have submitted its input objects to the current transaction. The reason for this is that when the request submits all their input objects and they are merged into the database, they need to wait for the latest transaction to commit successfully. The singleton also keeps track of the requests that need to be rollbacked using the `rollbackedProcesses` set and the requests which submitted the objects to the transaction that has been successfully committed.

Each merge request gradually reads the input object and submits it to the singleton while keeping track of objects submitted to the latest transaction in case of transaction failure.

```
TitanTransaction transaction = null;
try {
    transactionLock.lock();
    while((batchesBeingProcessed >= BATCH_COMMIT_COUNT
    || !rollbackedProcesses.isEmpty())
    && !rollbackedProcesses.contains(caller)) {
          waitingForCommitCondition.await();
    }
    if (rollbackedProcesses.contains(caller)) {
        throw new RepositoryRollbackException(···);
    }
    batchesBeingProcessed++;
    if (committedProcesses.contains(caller)) {
        transactionLock.unlock();
        caller.prepareForNextTransaction(inputBatch);
        transactionLock.lock();
    }
    transaction = getTransaction(caller);
    registeredProcesses.add(caller);
    return transaction;
} finally {
    transactionLock.unlock();
}
```

Listing 3.1: Retrieve/creating transaction

As shown in the listing 3.1, firstly the transaction lock, an instance of Java `ReentrantLock`, is acquired. After that, the value of `batchesBeingProcessed` is checked to determine if the commit threshold has been reached. If it has been reached the given process is put to sleep, in other cases it is allowed to continue and the counter is increased. The `batchesBeingProcessed` is not the only value checked, but also the `rollbackedProcesses` set to determine whether the running request needs to rollback its submitted objects. The last part checks for the request's affiliation to the `committedProcesses` set to determine whether the journal of a given object is to be cleared.

Then the object is merged to the database, followed by ending synchronization of the object processing, as shown in the listing 3.2, where the threshold `BATCH_COMMIT_COUNT` is compared to the `batchesDoneProcessed` to see if the transaction needs to be committed (as shown in the listings 3.2 and 3.3)

```
transactionLock.lock();
batchesDoneProcessed++;
if (batchesDoneProcessed == BATCH_COMMIT_COUNT) {
    commitTransaction();
    waitingForCommitCondition.signalAll();
}
transactionLock.unlock();
```

Listing 3.2: Ending merge object processing

```
if (transaction != null) {
    transaction.commit();
    transaction = null;
    committedProcesses.addAll(registeredProcesses);
    registeredProcesses.clear();
    batchesBeingProcessed = 0;
    batchesDoneProcessed = 0;
    doneProcesses = 0;
    dbLock.unlock();
}
```

Listing 3.3: Committing Titan transaction

Another important method depicted in the listing 3.4 is the method used by the requests which have finished merging all their input objects but need to wait for the final commit of the transaction. If all requests have all their input merged the final request will commit the transaction and notify other processes. If the request arrives at this section but the other requests are still submitting, the request is put to sleep and waits until the latest transaction is committed or failed.

```
transactionLock.lock();
doneProcesses++;
if (doneProcesses == registeredProcesses.size()) {
    commitTransaction();
    waitingForCommitCondition.signalAll();
}
while (!committedProcesses.contains(caller) && !rollbackedProcesses.
    contains(caller)) {
  waitingForCommitCondition.await();
}
if (rollbackedProcesses.contains(caller)) {
    throw new RepositoryRollbackException(···);
}
committedProcesses.remove(caller);
transactionLock.unlock();
```

Listing 3.4: Waiting for transaction commit

If the transaction error occurs, all merge requests, which have submitted objects to the given transaction, have to rerun their journals. This however is done using an exclusive lock on the database and not using the singleton transaction, as only one of the requests might be faulty.

3.1.2 Merging algorithm

This subsection contains the implementation of the locks used to ensure exclusive access to certain elements of the graph.

For merging objects `ReadLockedMergerProcessor` class is used which is an extension of the original `StandardMergerProcessor` containing merging logic. This class contains an instance of `LockingManager` class serving as a provider of all required locks. As mentioned in the section 2.1, each operation used to determine the previous existence of the certain object needs to be guarded to prevent the creation of duplicates. These locks can be split into 2 categories – vertices and edges.

For locking vertices synchronized Java `BlockingConcurrentMap` is used, which is an extension of the Java `Map` providing concurrent access. The key used in the case of vertices is a `LockableVertexQuery` instance, which contains the identifier of the vertex, name of the target, and the direction of the edge - to differentiate queries used to retrieve adjacent objects of a certain name. The value of each map entry is the reentrant lock.

In the case of edges, `BlockingConcurrentMap` is used as well, but using a different type of keys – EdgeIdentification, to guard querying for the existence of certain edges. The values used in the map are also reentrant locks.

The listings 3.5 and 3.6 demonstrate the retrieval of the lock for the certain vertex query (children of certain node object) and edge query respectively.

```
LockableVertexQuery vertexQuery =
  new LockableVertexQuery(parentVertexId, childName, Direction.IN);
Lock vertexLock = mergerLockingManager.retrieveQueryLock(vertexQuery);
vertexLock.lock();
```

Listing 3.5: Retrieving edge lock

```
EdgeIdentification edgeId = new EdgeIdentification(edgeDatabaseId);
Lock edgeLock = mergerLockingManager.retrieveEdgeLock(edgeId);
edgeLock.lock();
```

Listing 3.6: Retrieving edge lock

3.2 Neo4j migration

This section describes the implementation of the migrated merging algorithm in the Neo4j database, namely the methods mentioned in the merging algorithm description (see part 2.2.1). There is a number of helper methods, which

are not necessarily accessing the graph but only retrieving the properties or used to determine certain facts. These functions can be often used as-is or only a small part needs to be changed to use the Neo4j interface.

The more challenging parts are the methods accessing the graph, as they need to be rewritten from imperative Titan's approach to Cypher's declarative approach. Originally, in Titan's `StandardMergerProcessor` a lot of querying (mostly to determine previous existence) was done in-place, not using the methods inside structures, essentially worsening the code readability. The goal of this migration is also to move this database logic to the more appropriate structures.

3.2.1 Graph creation

This subsection describes the implementation of the queries used for creating new objects. The Neo4j driver interface provides a way to parametrize Cypher queries, however, one can not parametrize labels of nodes and types of relationships, which forces the programmer to use string concatenation or similar approaches making the readability worse. It is also impossible to parametrize the direction of the relationship.

The listing 3.7 provides example of creating a new node attribute. In the example, for both node and relationship parameter maps are created and filled with the required properties, then they are put into the final `parameters` map which is directly inserted into the query. Notice how `String.format` is used to parametrize label and type in the query, as it is currently not possible to parametrize the type.

```
Map<String, Object> parameters = ···
tx.run(String.format(
    "MATCH (node)\n"
    " WHERE ID(node)=$idNode" +
    "CREATE (node)-[edge:%s $relProps]->(attr:%s $attrProps)",
      "hasAttribute", "Attribute"), parameters);
```

Listing 3.7: Creating a new node attribute

Another quite different example is a case of inserting properties to an already created note or a relationship. The listing 3.8 represent this case. In Cypher `+=` operator can be used to add a map of properties to an entity.

```
Map<String,Object> parameters = ···
tx.run(
  "MATCH (v) \n"
  " WHERE ID(v)=$id\n"
  "SET v += $newNodeProperties\n"
  "RETURN v", parameters);
```

Listing 3.8: Inserting properties to an existing node

Most of the creational queries are similar in structure to those mentioned, hence no more examples will be provided in this part.

3.2.2 Graph operation

> **Cypher examples**
>
> Examples in this section and the following parts only show parametrized Cypher queries to provide better readability, but in real case similar approach as in the previous subsections is used.

This subsection describes the implementation of the queries used mostly for retrieving certain objects from the graph as well as new methods for retrieval existing objects to determine previous existence in the merging algorithm.

The listing 3.9 shows the retrieval of the node's children to determine whether the node which is currently merged exists. The listing only shows the Cypher query for better readability.

```
MATCH (child)-[r:hasParent]->(parent)
    WHERE r[$transEnd]>= $transEndVal
        AND ID(parent) = $idParent
        AND child[$name]=$nameValue
        AND child[$type]=$typeValue
        AND r[$edgeChild]=$edgeChildValue
RETURN child
```

Listing 3.9: Retrieve node's children

The listing 3.10 shows the retrieval of a relationship between given nodes (`sourceNodeId` and `targetNodeId`) of type *directFlow*.

```
MATCH (source)-[relationship:directFlow]->(target)
    WHERE relationship[$transEnd] >= $transEndVal
        AND ID(target) = $idTarget
        AND ID(source) = $idSource
        AND relationship[$edgeTarget]=$edgeTargetValue
RETURN relationship
```

Listing 3.10: Retrieve node's children

Retrieval of the resource is shown in the listing 3.11, as mentioned in the part 2.2.1.2 of the analysis, the best way to retrieve the resource of a node is to traverse from the node to the node representing the resource.

```
MATCH (node)<-[:hasAttribute*0..1]-
()-[:hasParent*0..]->()-[:hasResource]->(resource)
  WHERE ID(node)=$idNode
RETURN
  CASE
    WHEN $resourceLabel IN LABELS(node)
    THEN node ELSE resource
  END
AS resource ORDER BY length(p) ASC LIMIT 1
```

Listing 3.11: Retrieve node's resource

The implementation of full method `getVertexByQualifiedName` from the analysis using the list of strings as an input parameter to the query is depicted in the listing 3.12. Each element from the list is compared to the node on the corresponding position in the matched path as well as revisions of the control edges.

```
WITH $inputList AS path
    MATCH path = (superRoot)<-[:hasResource]-(res)<-[:hasResource*]-()
    WHERE ID(superRoot) = $idSuperRoot
        AND TOLOWER(res[$resNameProp])=TOLOWER($resName)
        AND res[$transStartProp] <= $tranStart
        AND res[$transEndProp] >= $tranEnd
        AND ALL(idx IN RANGE(2, SIZE(nodes(p))-1)
WHERE TOLOWER(COALESCE(NODES(p)[idx]["nodeName"], '')) = TOLOWER(path[
    idx-1][0])
AND TOLOWER(COALESCE(NODES(p)[idx]["nodeTypes"], '')) = TOLOWER(path[
    idx-1][1])
AND RELATIONSHIPS(p)[idx-1][$transStartProp] <= $tranStart
AND RELATIONSHIPS(p)[idx-1][$transEndProp] >= $tranEnd)
RETURN res
```

Listing 3.12: Retrieve node by its path from super root

All paths returned by the query are then evaluated to select the singular correct path based on the evaluation definition. This approach proved to be quite inefficient as the performance was substantially worse than in original Titan implementation (see section 4.9). Therefore another approach was implemented, which reflects the original implementation (see part 2.2.1.2), where Cypher queries are used for gradual traversing. The solution is further improved by only retrieving the resources of the nodes when the node has a direct link to a resource.

The second complex query is used to change the revisions of the subgraph of a certain node. At first, the control edge of the root node is retrieved to ensure that the revision is correct. After that, the query 3.13 is used to collect all paths representing the subtree of the given node while also updating *tranEnd* property of all the relationships connected to every collected node.

```
MATCH path = (m)<-[:hasParent*0..]-(b)-[:hasAttribute*0..1]->(a)
  WHERE id(m)=$idNode
    AND (nodes(path)[size(nodes(path))-1][$type]=$attribute
    OR (nodes(path)[size(nodes(path))-1][$type]= $node
    AND NOT (a)<-[:hasParent]-() AND NOT (a)-[:hasAttribute]->()))
  UNWIND NODES(path) AS allNodes
  MATCH (allNodes)-[relationships]-()
  SET relationships += $edgeProps
RETURN path
```

Listing 3.13: Retrieve node's subgraph

Retrieved paths are then iterated and all control edges are checked to ensure their correctness. In the case of the control edge being created in the latest committed revision, the node with whole its subtree is removed from the graph (see listing 3.14). The query is constructed to only match full paths to the ending nodes (hence the `WHERE` clause).

```
MATCH path = (a)<-[:hasAttribute*0..1]-(b)-[:hasParent*0..]->(m)
  WHERE ID(m)=$idNode
    AND (nodes(path)[0][$type]=$attribute
    OR (nodes(path)[0][$type]=$node
    AND NOT (a)<-[:hasParent]-() AND NOT (a)-[:hasAttribute]->()))
  UNWIND NODES(path) AS allNodes
DETACH DELETE allNodes
```

Listing 3.14: Remove node's subgraph

3.2.3 Revision utilities

This subsection describes Cypher queries used for revision editing of the graph. There are only few distinct queries used here – `setVertexTransactionEnd` and `setEdgeTransactionEnd` depicted in the listing 3.15.

```
MATCH (node)<-[controlEdge:hasParent]-()
    WHERE ID(node)=$idNode
SET controlEdge += $edgeProperties

MATCH ()-[relationship]-()
  WHERE ID(relationship)=$idRelation
SET relationship += $edgeProperties
```

Listing 3.15: Setting ending revision of a node and a relationship

3.2.4 Graph equality testing

The following subsection describes the implementation of MANTA graph equality testing. Both approaches from the analysis (see part 2.2.3 of the analysis) have been implemented.

3.2.4.1 Traversal framework and Java Core API

The listing 3.16 shows the traversal definition, which collects all the paths from *super root* to leaf *nodes*. The special condition in the `evaluator` part ensures that the paths, where the node has both parent node and resource are skipped, as it is not required for later comparison, therefore only increases the complexity.

```
tx.traversalDescription()
    .breadthFirst()
    .relationships(RESOURCE_TYPE)
    .relationships(PARENT_TYPE, Direction.INCOMING)
    .evaluator(path -> {
        if (path.length() == 2 && path.endNode().hasRelationship(
   Direction.OUTGOING, PARENT_TYPE)) {
            return Evaluation.EXCLUDE_AND_PRUNE;
        } else return Evaluation.INCLUDE_AND_CONTINUE;
    })
    .uniqueness(Uniqueness.RELATIONSHIP_GLOBAL)
    .traverse(tx.findNodes(SUPER_ROOT_LABEL)
```

Listing 3.16: Getting paths from super root node to leaf nodes

After the paths are collected the source codes are compared using the Java Core API and the mapping of nodes is created by comparing paths. After the mapping is done all node properties with their attributes and relationships are compared. The final step is to compare revisions, which is done similarly to the source code comparison.

3.2.4.2 Cypher

Graph comparison using Cypher was also done as mentioned in the analysis (part 2.2.3). The listing 3.17 shows the query used for DFS traversing from the starting super root node. The `ORDER BY` clause defines the ordering of the matched objects. The direction of the query is also parametrized to ensure the right order.

After retrieving nodes and relationships that are connecting the queried node and retrieved nodes their textual representation is saved. If the relationship is possibly ambiguous (see figure 2.2), it is stored and later evaluated with the possibly ambiguous relationships of the other graph. After the graph traversals are done for both graphs, these representations are compared.

```
MATCH (n)%s-[relationship]-%s(other)
  WHERE ID(n)=$idNode
OPTIONAL MATCH p=(other)-[]-()
RETURN other, relationship, count(p) as cnt
ORDER BY type(relationship), relationship.tranStart ASC,
    other.nodeName ASC, other.nodeType ASC ···
```

Listing 3.17: Traversing graph for comparison using Cypher

3.2.5 Testing

As a part of the algorithm migration, the various unit tests were implemented to ensure the correctness of certain operations based on the original merging process. The main focus was the testing of the merging algorithm alone to cover most of the different scenarios, which can occur while merging different object types. Tests covering the other implemented optimization improvements were implemented as well.

3.3 Improvements

This section contains the implementation of the minor improvements of the merging algorithm and the implementation of the merging algorithm as a user-defined procedure.

3.3.1 Minor optimizations

This subsection focuses on the implementation of the minor optimizations of the merging algorithm. Most of these adjustments are based on using some sort of cache.

- **Retrieving latest revision number** In this case only the new property is added to the contextual structure `ProcessorContext`, which is set before the merging of the given merge request begins and then is retrieved every time it is required.
- **Node identifiers instead of nodes** All methods have to be reworked to use the identifiers of the nodes as parameters instead of the nodes. Most of the methods use the full objects only to directly access the object within a database using its identifier, therefore it did not cause many complications to change them. However in some cases, the properties of these objects were used, therefore the queries had to be rewritten to use the properties if possible. The listings 3.18 and 3.19 show the different approaches, where when working with full node object, `childName` string is acquired by processing source node.

```
String childName
  = GraphOperation.processChildName(source.name()));

MATCH (source), (target)
  WHERE ID(source)=$idSource
    AND ID(target)=$idTarget
    AND $sourceLabel in LABELS(source)
    AND $targetLabel in LABELS(target)
  CREATE (source)-[edge:$edgeType $edgeProps]->(target)
  SET edge.childName= $childName
RETURN edge
```

Listing 3.18: Create relationship by using node

```
MATCH (source), (target)
  WHERE ID(source)=$idSource
    AND ID(target)=$idTarget
    AND $sourceLabel in LABELS(source)
    AND $targetLabel in LABELS(target)
  CREATE (source)-[edge:$edgeType $edgeProps]->(target)
  SET edge.childName= CASE source[$vertexType]
    WHEN $typeNode THEN substring(source["nodeName"], 0, 200)
    ELSE substring(source["resourceName"], 0, 200) END
RETURN edge
```

Listing 3.19: Create relationship only by using node identifiers

- **Existing nodes** This optimization requires adding a new set containing node identifiers to the contextual structure. It also requires adding the logic to the merging algorithm – adding a node's identifier when the merged node existed in the database prior to the merging of the current revision, and querying the set when processing objects, which require the previous existence of a given node. (see listing 3.20)

```
if (context.nodesExistedBefore().contains(parentVertexId)) {
    List<Node> children = graphOperation.getChildren(···)
    ···
}
```

Listing 3.20: Retrieving the children of a node

- **Node to resource mapping** This optimization requires adding a new map containing node identifiers as keys and database identifiers as values to the contextual structure. Then adding the logic of insertion and retrieval of this map when processing resource and node objects (see listing 3.21 where the mapping is retrieved and inserted)

```
Long resourceParentId = context.mapNodeResourceMapping().get(
    parentVertexId);
if (resourceParentId != null && resourceParentId.equals(
    resourceVertexId)) {
    ··· create new node only connected to parent or resource
} else {
    ··· create new node connected to both parent or resource
}
context.mapNodeResourceMapping().put(newNode.id(),
    resourceVertexId);
```

Listing 3.21: Creating a new node

3.3.2 User-defined procedures

This section describes migrating the implemented merging algorithm in Neo4j to a user-defined procedure to improve the performance of the algorithm.

3.3.2.1 Differences

The whole merging algorithm, as well as graph accessing structures, had to be rewritten to the internal interface. (see part 2.3.2 of the analysis)

3.3.2.2 Transaction handling

As mentioned in the analysis, it is possible to either use the injected transaction object or to use the injected database service. The approach of using the latter has to be used as it is required to create multiple transactions under the same procedure allowing further parallelism. The listing 3.22 shows how the transaction is created and committed.

```
Transaction transaction = graphDatabaseService.beginTx();
transaction.execute(···);
transaction.commit();
```

Listing 3.22: Transaction cycle inside user-defined procedure

3.3.2.3 Handling the procedure

This part describes how these procedures are handled from the MANTA server-side. The whole merge request needs to be sent to the Neo4j server so that it can be processed. This is done by reading the whole input and creating a list of the merge objects, which can be used as a procedure parameter. The list is created in a way that groups of each objects types are stored separately which eliminates preprocessing needed in the case of parallel processing as each object type is processed separately.

Listing 3.23 displays how the procedure is called with the parameters.

```
CALL manta.merge($mergeObjects, $revision)
    YIELD time, newObjects, errorObjects, existedObjects,
        unknownTypes, processedObjects, requestedSourceCodes
RETURN time, newObjects, errorObjects, existedObjects, unknownTypes,
    processedObjects, requestedSourceCodes
```

Listing 3.23: Calling a user-defined procedure

The returning object of the user-defined procedure has to be a stream of objects. These objects can only contain supported types of Neo4j procedures [30]. Special class `MergeOutput` was created for this, containing the result of the merging (count of new objects, merged objects, ...).

3.4 Parallelization

The following section describes the implementation details of the parallelization approaches proposed in the analysis chapter (see part 2.4). The first subsection describes the managing process of thread creation and synchronizing. The next subsection describes each working thread. The following subsections describe various approaches to parallelization.

3.4.1 Contextual structure

For using multiple threads all structures inside the contextual object need to be synchronized to ensure correct concurrent access. Only updated values are various sets and maps, for which Java's `ConcurrentHashMap` implementation can be used.

For the advanced parallelization with splitting the processing of node attributes and edges into 2 phases (non-blocking and potentially-blocking), the contextual object is further supplemented by additional fields to store the operations, which are evaluated at the potentially-blocking phase. The way the storing of these operations is implemented is that instead of evaluating them, they are added to the lists used for this storing. An example of this is creating a new node attribute, in the listing 3.24, the node attribute is directly created, but in the listing 3.25 it is stored inside the contextual structure.

```
graphCreation.createNodeAttribute(
  context.getServerDbTransaction(),
  nodeIdentifier,
  attributeKey,
  attributeValue,
  revisionInterval
);
```

Listing 3.24: Creating a new node attribute

```
DelayedAttributeQuery query = new DelayedNodeAttributeQuery(
  nodeIdentifier,
  attributeKey,
  attributeValue,
  revisionInterval
);
context.getNodeAttributeQueryList().add(query);
```

Listing 3.25: Storing a node attribute creation

Each of these postponed operations has its evaluation reflecting the original evaluation.

- **DelayedNodeResourceEdgeQuery** This query consists of creating a new edge connecting the node and its resources and is used when creating nodes that have both parent and resource to ensure no lock contention occurs.
- **DelayedEdgeQuery** Edge is defined by its source and target nodes' identifiers, by its type, and by the arbitrary number of properties.
- **DelayedNodeAttributeQuery** Node attribute is defined by the owning node identifier and by the *attributeKey* and *attributeValue* properties.
- **DelayedPerspectiveEdgeQuery** This query consists of creating a new edge and a new node attribute and is evaluated in the potentially-blocking phase of edge processing.

3.4.2 Managing threads

This subsection explains how threads are managed and how the whole process looks from a higher perspective.

3.4.2.1 Creating and completion of the threads

For managing each created thread the synchronization is needed. The thread life cycle is managed by the instance of Java `ExecutorService`. The listing 3.26 shows how the instance of `ExecutorService` is created and how new threads are submitted to it. The listing 3.27 shows how the executor waits for the completion of the execution of all the submitted threads.

```
ExecutorService executorService =
    Executors.newFixedThreadPool(threads);
IntStream.range(0, threads).forEach(
        e -> executorService.submit(
                new MergerWorker(···)
        )
);
```

Listing 3.26: Submitting a working threads to executor service

```
executorService.shutdown();
try {
    if (!executorService.awaitTermination(
        Integer.MAX_VALUE, TimeUnit.MILLISECONDS)) {
        executorService.shutdownNow();
    }
} catch (InterruptedException e) {
    executorService.shutdownNow();
}
```

Listing 3.27: Executor service waiting for submitted threads to finish

3.4.2.2 Controlling the process flow

To ensure the correct processing barrier is required. For this case, Java `Phaser` was used, which allows controlling the flow of the processing. An instance of `Phaser` serves as a barrier that forces all registered threads to wait until all arrive.

The execution of parallel processing alone is split into various stages controlled by `Phaser` instance. The first stage is merging the starting parts (source codes, layers, and resources) sequentially.

3.4.2.3 Transaction-retry processing

This part explains the process flow when using the solution, where locking problems occur and transactions have to be retried in case of deadlock. Initially, the source codes, layers, and resources are processed sequentially.

The following phase is a preprocessing phase, in which the nodes, the node attributes, and the edges are preprocessed. The implementation of the preprocessing is described in more detail in the following part.

Node attributes are split evenly by the owning nodes as described in part 2.4.2.3 of the analysis. Edges are split recursively by the source and target nodes.

The next phase is the node processing which is divided into two subphases. The first merge the nodes in parallel, followed by the creating of the node-resource edges, which is delayed to avoid locking problems.

After the node phases are finished, node attributes can be processed. This phase is as well divided into 2 sub-phases, where the first sub-phase only contains node attributes, which are not of type *mapsTo*. It is processed in the second phase sequentially to avoid locking problems as the relationships are created when processing this type.

The last phase is the processing of the edges, which are split into multiple sub-phases in a node-conflict reducing way. Edge attributes are processed together with the edge they are assigned to. Each phase has to be guarded by a barrier so that no thread, having finished its work, can proceed further, as the lock conflicts may arise. The listing 3.28 shows the workflow described above.

```
··· merge initial objects sequentially

··· node preprocessing
··· node attribute preprocessing
··· initial edge preprocessing

··· create merging threads

··· barrier node processing

··· barrier node attribute processing

··· barrier edge and edge attributes processing

··· wait for the completion of the processing
```

Listing 3.28: Processing flow of parallel merging

3.4.2.4 Deadlock-free processing

This processing is very similar to the previously mentioned processing, with a few differences. The node attributes are not processed separately but are processed with the nodes. During the node preprocessing, along with the children of each node, node attributes belonging to it are also stored. This ensures that all nodes and node attributes are processed in a non-conflicting way.

The other difference is the edge processing. The edges are preprocessed into phases in such a way that the neighborhood of the source and target nodes is also respected. For this solution to work properly, all the adjacent nodes of every node participating in the edge creation need to be collected. In this implementation, only the adjacent nodes from the merge request are used, to see whether this solution is better than the others (see the last chapter 4). The listing 3.29 shows the deadlock-free merging workflow described above.

```
··· merge initial objects sequentially

··· node and node attributes preprocessing
··· edge and edge attributes preprocessing

··· create merging threads

··· barrier node and node attribute processing

··· barrier edge processing

··· wait for the completion of the processing
```

Listing 3.29: Processing flow of deadlock-free parallel merging

3.4.2.5 Delayed processing

This processing is an improvement of the initial parallelized solution, where during the node attribute and edge preprocessing two sub-phases for each type are created.

The first sub-phase of node attribute preprocessing contains all but *mapsTo* attributes split evenly for the number of threads. In this sub-phase, only reading or relationship updating is done. In the second sub-phase evenly distributed *mapsTo* attributes are processed in the same way as in the first sub-phase. Creational operations are stored in the contextual object, which is followed by the preprocessing of the stored node attribute creational queries, where they are evenly split by the owning nodes and then processed by multiple threads. Note that result of new *mapsTo* node attributes are edges, which are processed in the next phase.

The edges are processed similarly, non-blocking phase of edge processing is divided into a sub-phase processing normal edges and a sub-phase processing *perspective* edges. Creational edge queries are stored and then split together with the previously stored *mapsTo* edges by the source and target nodes and processed in the next phase. The listing 3.30 shows a workflow of the merging processing with the non-blocking and potentially-blocking phases.

```
··· merge initial objects sequentially

··· initial node preprocessing
··· initial node attribute preprocessing
··· initial edge preprocessing

··· create merging threads

··· barrier node processing

··· barrier first sub-phase of the node attribute processing
```

```
··· split creational node attributes queries
··· barrier preprocessing completion
··· barrier second sub-phase of the node attribute processing

··· barrier first sub-phase of the edge attribute processing
··· split creational edge queries with their attributes
··· barrier preprocessing completion
··· barrier second sub-phase of the node attribute processing

··· wait for the completion of the processing
```

Listing 3.30: Processing flow of delayed parallel merging

3.4.3 Preprocessing

This subsection explains how the node attributes and the edges are preprocessed before they are merged into the database. After preprocessing, each part of the node attributes or the edges, which can be processed by a single merging thread is stored within a wrapping structure `MergeContainer`. A list of the instances of this structure is stored within a blocking queue to allow concurrent polling of multiple threads.

3.4.3.1 Nodes

Nodes are preprocessed in a way that each node or resource mapping to its children is stored, which is required for merging the nodes in parallel. The shared queue is initialized by inserting all resources.

3.4.3.2 Node attributes

Node attributes are split by their owning nodes. At first, the approximate size of the part assigned to each thread must be calculated, which is done by dividing the number of node attributes by the number of threads. After that, iteration through all node attributes assigns them to the list gradually. If the given list reaches the calculated size, all the remaining node attributes owned by a node, whose node attributes were already assigned to the given list, must be assigned to the same list to prevent locking problems. As the node attribute's input list is sorted by the owning nodes, no sorting is required, only the rest of the attributes must be assigned and then the filling of the following list begins.

3.4.3.3 Edges and edge attributes

For each thread, one instance of `MergeContainer` is created, to which the edges are assigned. The first variant is the splitting of the edges only by their source and target nodes, which is done in a way that all edges are iterated and

are assigned (along with its source and target nodes) based on the previous assignments. In the case of the source and the target node not belonging to any containers, the edge with its nodes is assigned to a certain container given by one of the splitting algorithms described in part 2.4.2.4. In the case of using the second variant, the same approach is used, but the algorithm, which takes also the node's neighborhood into account, is used for determining the edge's alignment.

3.4.4 Working thread

This subsection describes implementation details of the parallel processing of the nodes using the internal and shared queue and the processing of the node attributes, the edges, and the edge attributes after their preprocessing are done.

3.4.4.1 Nodes

This part describes the implementation of parallel node processing. The synchronization is done using Java `ReentrantLock` and `Condition` is used for thread's sleeping and notifying.

The main merging loop (see listing 3.31) describes the first sub-phase of node processing.

```
NodeMergingStep step = getNextStep(internalQueue);
if (step.getState() == PROCESSING_CONTINUE) {
    continue;
} else if (step.getState() == PROCESSING_DONE) {
    break;
} else {
    nodeId = step.getNodeId();
}
List<List<String>> ownedNodes = nodeQueue.getOwned(nodeId);
if (!ownedNodes.isEmpty()) {
    mergeNodesAndAttributes(ownedNodes, journal, internalQueue);
}

if (nodeLock.getWaitingCounter() > 0 && internalQueue.size() >
    nodeLock.getWaitingCounter()){
    splitInternalQueue(internalQueue, journal);
}
```

Listing 3.31: Loop of merging nodes

The method `getNextStep` (see listing 3.32) is used to retrieve the next node identifier for further processing and to terminate the loop. In the method, retrieval from the internal queue is attempted. In case that this queue is empty, retrieval from the shared queue is attempted. If the retrieval is not successful, the thread is put to sleep and the thread waiting counter is increased. If all

merging threads are put to sleep, the phase is deemed finished and the threads are woken up.

```
if (!internalQueue.isEmpty()) {
    nodeId = internalQueue.poll();
} else {
    nodeId = nodeQueue.poll();
    if (nodeId == null) {
        nodeLock.getWaitLock().lock();
        nodeLock.increaseWaitingCounter();
        if (nodeLock.getThreads() == nodeLock.getWaitingCounter()) {
            nodeLock.getWaitingForWorkCondition().signalAll();
            return NodeMergingStep.of(PROCESSING_DONE);
        } else {
            nodeLock.getWaitingForWorkCondition().await();
        }
        if (nodeLock.getThreads() == nodeLock.getWaitingCounter()) {
            return NodeMergingStep.of(PROCESSING_DONE);
        }
        nodeLock.decreaseWaitingCounter();
        nodeLock.getWaitLock().unlock();
        return NodeMergingStep.of(PROCESSING_CONTINUE);
    }
}
return NodeMergingStep.of(PROCESSING_NODE, nodeId);
```

Listing 3.32: Retrieving next node element

The method `splitInternalQueue` (see listing 3.33) shows how the internal queue is split when the merging thread detects that there are other merging threads with no more work. The part of elements from the internal queue is moved to the shared queue from which the other merging threads can retrieve them.

```
ensureTransactionCommit(journal);
Transaction tx = graphDatabaseService.beginTx();
context.setServerDbTransaction(tx);
int keep = internalQueue.size() / (nodeLock.getWaitingCounter() + 1);
for (int i = 0; i < internalQueue.size() - keep; ++i) {
    nodeQueue.addToQueue(internalQueue.poll());
}
nodeLock.getWaitLock().lock();
nodeLock.getWaitingForWorkCondition().signalAll();
nodeLock.getWaitLock().unlock();
```

Listing 3.33: Splitting an internal queue

The second sub-phase of the node processing is the creation of the node-resource relationships as depicted in the listing 3.34.

```
nodeLock.getWaitLock().lock();
NodeResourceRelationshipQuery query = nodeQueue.
    getNodeResourceRelationship();
if (query == null) return;
Transaction tx = graphDatabaseService.beginTx();
context.setServerDbTransaction(tx);
while (query != null) {
    processor.processNodeResourceRelationship(query, context);
    query = nodeQueue.getNodeResourceRelationship();
}
tx.commit();
nodeLock.getWaitLock().unlock();
```

Listing 3.34: Second sub-phase of the node processing

3.4.4.2 Node attributes and edges

For both the node attributes and the edges. The listing 3.35 shows how the sub-phases are processed. For example in the case of delayed node attribute processing, the first line of the listing refers to non-blocking processing, followed by the phaser barrier and by a potentially blocking processing. The listing 3.36 shows how each part of the previous sub-phases is processed. An instance of `MergeContainer`, containing the list of all objects to merge, is retrieved from the synchronized queue by each merging thread and then further merged into the database.

```
queuePhaser.getCurrentPhase().getQueueList()
  .forEach(this::processMergeQueue);
phaser.arriveAndAwaitAdvance();
queuePhaser.getCurrentPhase().getQueueList()
```

```
  .forEach(this::processMergeQueue);
```

Listing 3.35: Processing of sub-phases

```
MergeContainer container = mergeQueue.poll();
if (container != null && !container.getContent().isEmpty()) {
    mergeObjectsToDatabase(container.getContent());
}
phaser.arriveAndAwaitAdvance();
```

Listing 3.36: Processing within a sub-phase

3.4.5 Unifying merge requests

This section describes the implementation of the unification of the different merge requests. As mentioned in the part 3.4.5 of the analysis, two levels of synchronizations are needed.

3.4.5.1 Synchronizing merge types

To ensure only one type of merging request can be unified, synchronizing structure `MergerManager` was created, which handles the registration and unregistration of the requests. The synchronization is done by Java `ReentrantLock`. The listings 3.37 and 3.38 reflect these operations respectively.

```
lock.lock();
if (lockedGroup == null) {
    init(mergingGroup);
} else if (mergingGroup.equals(lockedGroup)) {
    registered++;
} else {
    while (true) {
        lockCondition.await();
        if (lockedGroup == null) {
            init(mergingGroup);
            return;
        } else if (lockedGroup.equals(mergingGroup)) {
            registered++;
            return;
        }
    }
}
lock.unlock();
```

Listing 3.37: Registering a new merge request

```
lock.lock();
registered--;
if (registered == 0) {
    lockedGroup = null;
    lockCondition.signalAll();
}
lock.unlock();
```

Listing 3.38: Deregistering a merge request

3.4.5.2 Synchronizing unification and database submissions

To synchronize unification of the merge requests, 3 different flags are used – *request-merging*, *database-merging* and *waiting* flags, along with a locks to determine how requests should proceed.

If a request arrives and *request-merging* flag is set, it subscribes to the request merging and merges itself with other requests. Upon finishing merging itself it checks whether all subscribed requests have finished merging. If that is the case and database-merging flag is unset and it proceeds to merge the unified request into the database. If not all subscribed requests have finished merging or the database-merging flag is set the thread is put to sleep and waits for the shared result.

If the last request is allowed into the *database-merging* phase and finishes the database work, it checks whether there is any unified request available for the processing – this is a situation, where the second batch of requests finished merging themselves (while another request was being merged to the database), but the last merged request could not proceed to the database merging phase due to the first batch request holding the lock. If there was a unified merge request it is sent to the database for processing. This is done recursively until no other request is ready for database merging. Upon ending, the *database-merging* flag is unset and all sleeping threads are awoken.

If there is no request processed currently - the process proceeds to the database merging phase and sets the *database-merging* flag. After this is done - the request checks whether there is any merged request available for processing and proceeds in the same way as in the previous situation.

If the other request arrives and the *request-merging* flag is unset and *database-merging* flag is set, it checks whether any other process is waiting. If that is true, the request merging phase is initialized and the other waiting thread awakens and is merged, following the first situation. If the *waiting* flag is unset, it is set and the request is put to sleep. Once the request is awakened it checks whether *request-merging* flag is set and it joins the merging if it is. The other case is that *request-merging* flag is unset and *database-merging* flag is unset as well, in that case, the request proceeds to merge itself into the database as in the second situation.

3.4.5.3 Request unification

This part describes the implementation of the actual merging of the different requests. The implementation lies in the use of Java `ConcurrentHashMap` and its method `computeIfAbsent`, ensuring concurrent synchronization. The synchronization structures were implemented as per analysis (see part 2.4.5.3), where various merged types require some sort of identification to ensure no duplicates are created. These identification classes only contain vital data, which is used to compute hash codes that are used as the keys within the `ConcurrentHashMap`'s instances.

The listing 3.39 shows how node attribute object is handled, where the `unifiedNodeId` represents the new mapped node identifier in the unified request, `nodeToAttributes` represents the mapping of a node to its attributes and `nodeAttributeIdentifier` is an instance of node attribute identification class containing the key and value to ensure no duplicates are created.

```
nodeAttribute.set(NODE_ID, unifiedNodeId);
nodeToAttributes.computeIfAbsent(nodeId,
  key -> new ConcurrentHashMap<>())
        .computeIfAbsent(nodeAttributeIdentifier, key -> {
            nodeAttributes.add(nodeAttribute);
            return true;
        }
);
```

Listing 3.39: Unifying requests – merging a node attribute

The goal of this chapter was to acquaint the reader with how the certain parts of this work were implemented. Each of the parts discussed in the chapter 2 were described from the implementation perspective.

Chapter 4

Evaluation

This chapter focuses on testing how various implemented algorithms performed in comparison with the original Titan solution. The first subsection discusses the initial implementation of the parallelization within the Titan algorithm. The next subsection introduces the data used to test the performance on the migrated Neo4j, and the testing machine's specifications, followed by subsections showing the testing results of the performance using the introduced data.

4.1 Titan parallelization

After implementing Titan parallelization, various problems occurred when testing this implementation. The problems are as follows:

- **Transaction closed** Transaction closes and an exception is thrown when one of the merging threads attempted to access it. This is probably related to synchronization, where concurrent modification of a single object caused it to close abruptly.

- **SimpleVertexQuery** Exception thrown when a query is being created as there are 2 different types of queries, which can be used. The specific type is deduced in Titan by the state of vertex, which is queried. There are various states such as *new*, *loaded*, *modified* etc. This problem is caused because the type of query is firstly deduced when evaluating a query, but the queried vertex is subsequently modified in some way by other concurrently running threads and the state changes, which causes the first thread to fail when the state is later rechecked. This is a synchronization problem within the threaded transaction implementation and can be solved by reattempting to query but could cause significant overhead.

Dataset	Nodes	Edges	Split files	Unified files
Small	1017	3170	5	1
Medium	55361	205883	9	1
Large	1484721	3212213	169	22

Figure 4.1: Main datasets description

- **Missing vertices** It seems that vertices are not always correctly created as the merging process fails due to required vertices not being in the database (they should be added previously as MANTA hierarchy is followed).

These problems could probably be fixed by adding a writing synchronization on the specific queries, however, the performance while ignoring these errors was not better than running sequential merging using Titan, therefore if another layer of synchronization was added, the process would be slowed down even more.

4.2 Testing environment and data introduction

Properties of the testing environment are following:

- **RAM** 16384 MB DDR4 @ 3200 MHz
- **CPU** Intel Core i7 (10th Gen) 10610U @ 1.8 GHz
- **SSD** 500 GB NVM Express (NVMe)

Various testing datasets were created for performance-testing of the migrated merging algorithm, implemented optimizations as well as the parallelization of the whole algorithm. There are 3 main datasets of the various size described in the figure 4.1. For each dataset, there are 2 different versions – split, where the files retrieved from the analysis are used and unified, where the unification of requests is used (see part 3.4.5).

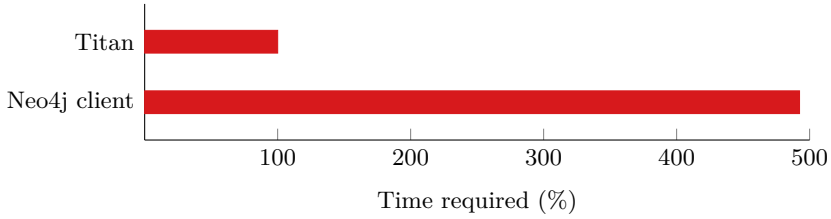
There are also 2 additional datasets used for performance testing of expensive operations – merge file containing **2500** *mapsTo* edges of nodes in depth 12, and merge file containing **1000** *perspective* edges.

4.3 Migrated merging process

The tables and graphs in the figures 4.2, 4.3, and 4.4 show how client-sided merging process performed in comparison with the original Titan implementation.

Create graph

	Time (ms)	%
Titan	737	100
Neo4j client	3629	492.4



Update graph

	Time (ms)	%
Titan	716	100
Neo4j client	2886	403.1

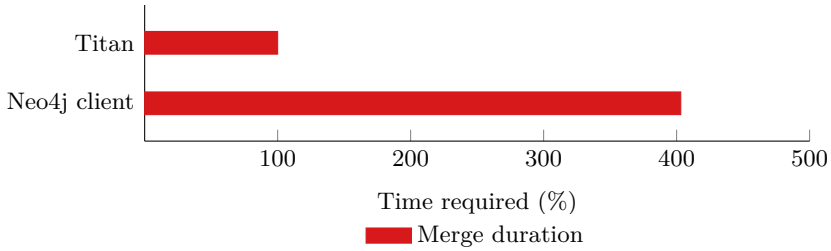


Figure 4.2: Small dataset – Neo4j client merger performance

As seen in the graphs, the performance of the initial migrated version is much worse than the original Titan implementation. This is caused by a substantially large amount of database calls, which generate heavy overhead.

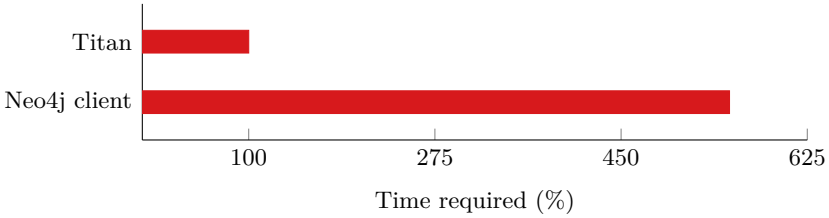
4.4 User-defined procedure

The figures 4.5, 4.6, and 4.7 provide comparison of the original Titan implementation and the implementation of merging process as an user-defined procedure.

The implementation of the merging process as a user-defined procedure eliminated thousands of distinct database calls resulting in a significantly better performance (time required was reduced to 60-70% for initial graph creation and to 50% for updating revisions).

Create graph

	Time (ms)	%
Titan	45115	100
Neo4j client	248981	551.9



Update graph

	Time (ms)	%
Titan	52283	100
Neo4j client	207882	397.6

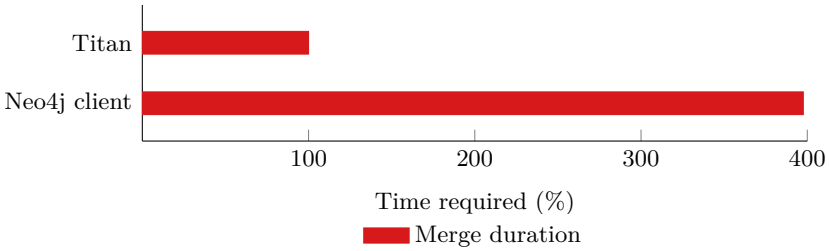


Figure 4.3: Medium dataset – Neo4j client merger performance

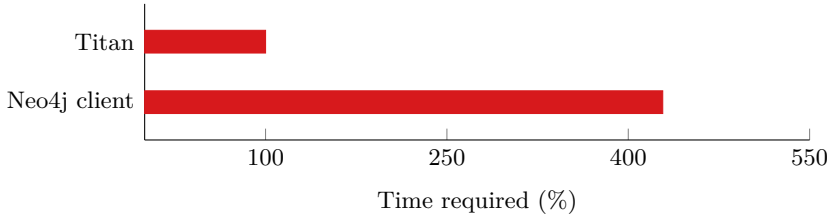
4.5 User-defined procedure with improvements

This section describes how each of the minor optimization steps improved the performance of the initial user-defined procedure. Performance was tested on each of the main datasets using the user-defined procedure version of the merging algorithm. The figures 4.8, 4.9, 4.10 show how each of the optimization steps improved the performance for both the initial creation of the graph and the update of the graph.

Using the medium and large dataset, the time required for initial graph merging was more than halved when compared to initial user-defined implementation, where the largest contribution was due to caching *latestCommittedRevision* and caching nodes, which were existing in the database before merging. The updating of the graph was only influenced by the *latestCommittedRevision* caching and by using node identifiers instead of retrieving the nodes when processing various object types. The other two optimization techniques did not affect the merging, as all the nodes already did exist, making

Create graph

	Time (ms)	%
Titan	879523	100
Neo4j client	3768942	428.5



Update graph

	Time (ms)	%
Titan	987580	100
Neo4j client	3334681	337.7

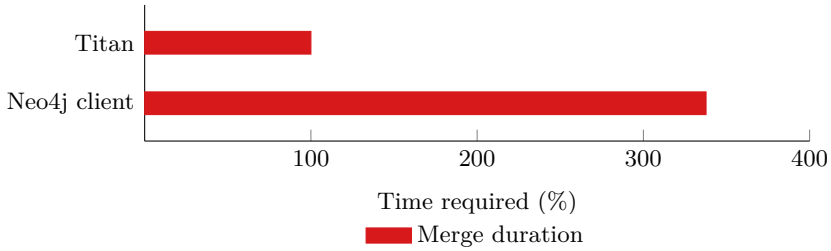


Figure 4.4: Large dataset – Neo4j client merger performance

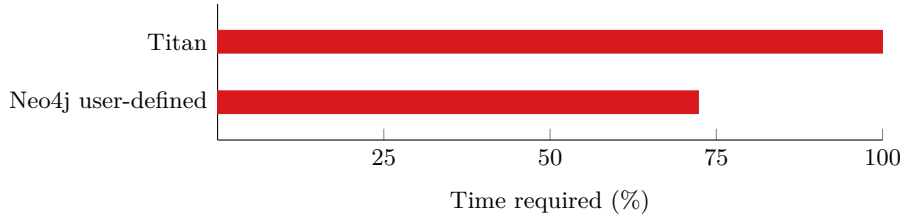
the cache pointless and resource retrieval not required.

4.6 Edge preprocessing splitting techniques comparison

This section describes the differences in splitting the edges by using smallest-first, round-robin, and random approaches. The figure 4.11 shows the reduction of the edges' count on various merging input files (taken from the medium dataset) by using each of the three approaches. When using only 2 containers, the smallest-first approach performed similarly to the round-robin approach, but when splitting into 4 containers, it performed significantly better. The random approach performed the worst overall.

Create graph

	Time (ms)	%
Titan	737	100
Neo4j user-defined	533	72.3



Update graph

	Time (ms)	%
Titan	716	100
Neo4j user-defined	409	57.1

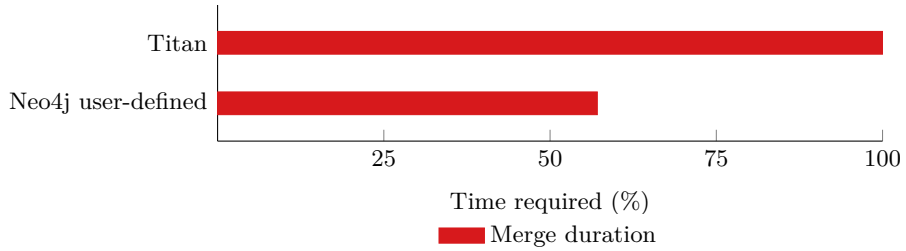


Figure 4.5: Small dataset – Neo4j user-defined merger performance

4.7 Comparison of transaction-retry and deadlock-free solution

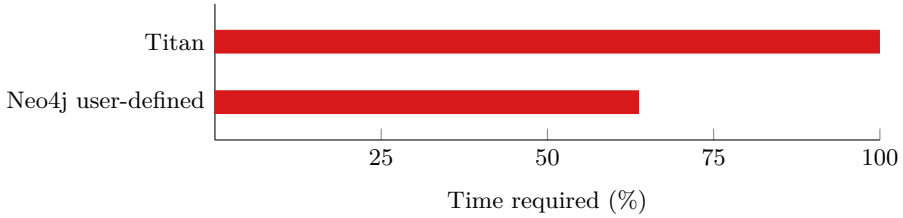
The figure 4.12 compares the performance of the transaction-retry solution with the deadlock-free solution. These two approaches were compared using the medium dataset.

From the graph, it seems that the deadlock-free solution performed slightly better, even despite the worse reduction of edge subset due to more complex splitting. In the test case, the neighborhood was determined only by using the merge input as this was not fully implemented, just to see whether it will be substantially better than the transaction-retry solution. When querying the database for all the adjacent nodes of nodes for which edges are created, it took around 2 seconds (as seen in the figure 4.12), making it much worse.

Moreover, after sending the problematic case of deadlocks in transaction-retry implementation to Neo4j, the cause of the locking/deadlocks was determined to be false positivity caused by the change of locking management, hence the transaction-retry implementation will be used for further testing.

Create graph

	Time (ms)	%
Titan	45115	100
Neo4j user-defined	28724	63.7



Update graph

	Time (ms)	%
Titan	52283	100
Neo4j user-defined	25655	49.1

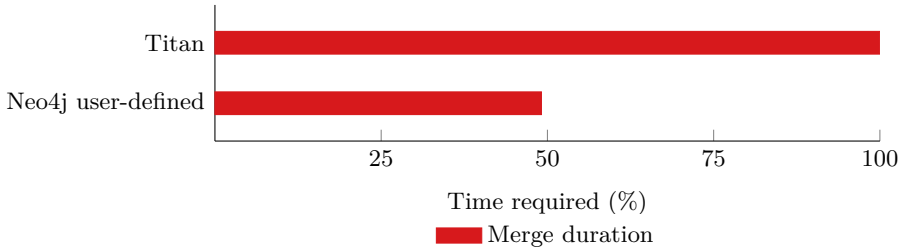


Figure 4.6: Medium dataset – Neo4j user-defined merger performance

Once the issue is resolved within Neo4j, the performance gain should be even better.

4.8 User-defined procedure with parallelization

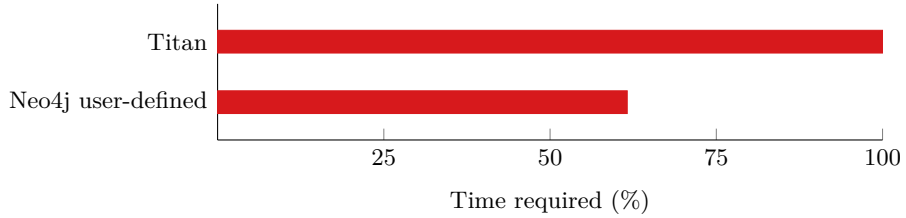
This section focuses on the performance of the transaction-retry parallel merging algorithm with all previously mentioned optimizations included.

The figures 4.13, 4.14, 4.15 show how the parallelized algorithm using 2 and 4 threads performed in comparison with sequential solution. Both creating and updating of the graph using three main datasets are depicted in the figures.

The performance of 2-threaded and 4-threaded solutions when creating a new graph was quite similar, which is likely caused by locking and deadlocking occurring more often when using more threads and also by synchronization, which Neo4j requires when creating new objects. When updating revisions, however, the performance difference was more significant.

Create graph

	Time (ms)	%
Titan	879523	100
Neo4j user-defined	541367	61.6



Update graph

	Time (ms)	%
Titan	987580	100
Neo4j user-defined	468887	47.5

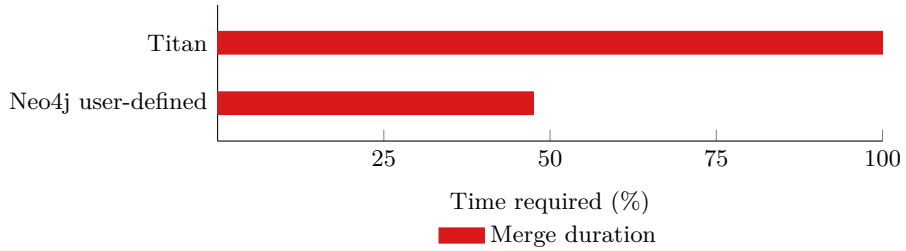


Figure 4.7: Large dataset – Neo4j user-defined merger performance

4.9 Special types performance

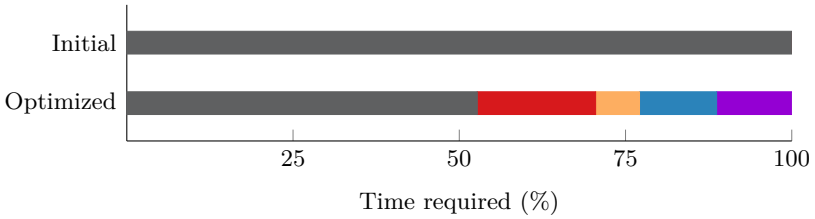
The figures 4.16 and fig:perspective-edges show how the migrated algorithm performed on datasets containing only *mapsTo* node attributes and *perspective* edges. These types require substantially more computing time, so it's important to know how well they perform.

The initial query matching full pattern, where the properties were evaluated based on input array using `ALL` statement performed significantly worse than the original Titan implementation, hence the imperative approach using Cypher queries mirroring the original implementation was implemented. This approach provided much better results, as seen in the figure 4.16.

The merging of the *perspective* dataset was slightly slower in sequential solution, but had quite good performance scalability when using the parallel approach, therefore providing better results.

Create graph

	Time (ms)	%
Initial	533	100
Latest revision	438	82.2
Existing nodes	498	93.4
Resource	471	88.4
Node fetching	473	88.7
Final	281	52.7



Update graph

	Time (ms)	%
Initial	409	100
Latest revision	332	81.2
Existing nodes	409	100
Resource	409	100
Node fetching	349	85.3
Final	272	66.5

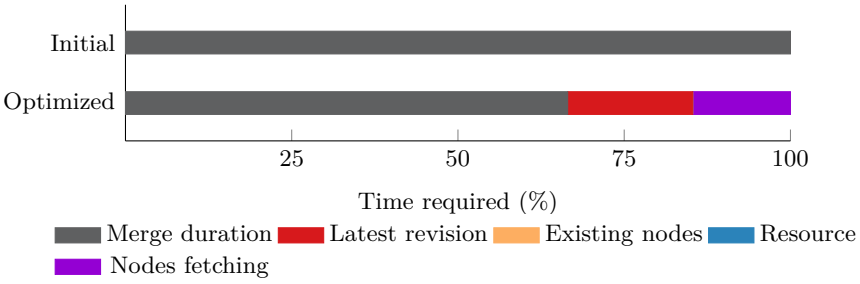
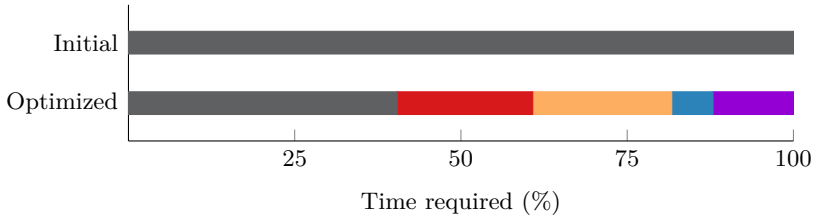


Figure 4.8: Small dataset – Reduction of merging duration by minor optimizations

Create graph

	Time (ms)	%
Initial	28724	100
Latest revision	22876	79.6
Existing nodes	22716	79.1
Resource	26948	93.8
Node fetching	25224	87.8
Final	11592	40.4



Update graph

	Time (ms)	%
Initial	25655	100
Latest revision	19222	74.9
Existing nodes	25655	100
Resource	25655	100
Node fetching	22155	86.4
Final	15334	59.8

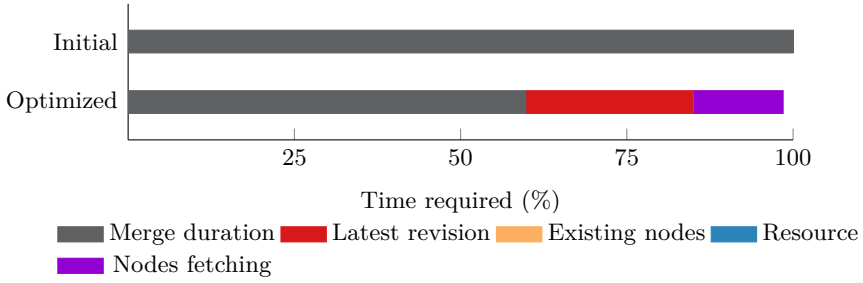
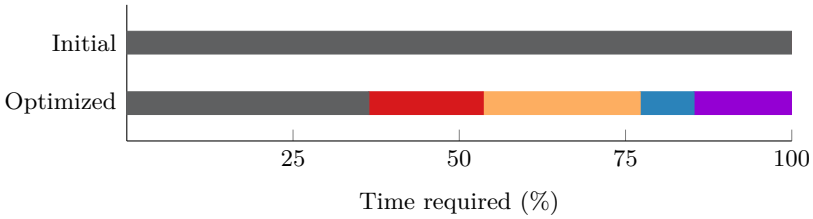


Figure 4.9: Medium dataset – Reduction of merging duration by minor optimizations

Create graph

	Time (ms)	%
Initial	541367	100
Latest revision	448508	82.8
Existing nodes	413752	76.4
Resource	497553	91.9
Node fetching	461367	85.2
Final	197079	36.4



Update graph

	Time (ms)	%
Initial	468887	100
Latest revision	369088	78.7
Existing nodes	468887	100
Resource	468887	100
Node fetching	388887	82.9
Final	289088	61.6

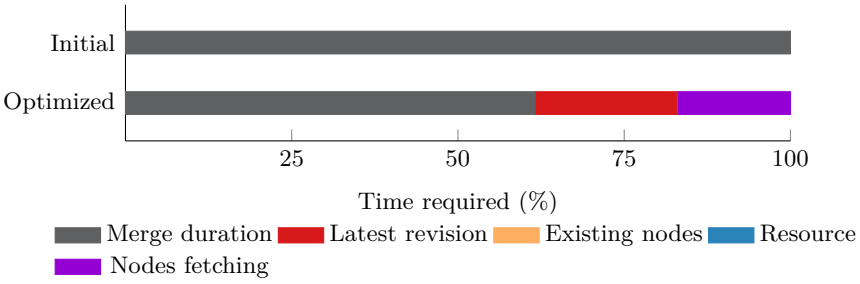


Figure 4.10: Large dataset – Reduction of merging duration by minor optimizations

2 containers

Edges	SF	RR	RN
23090	0.512	0.535	0.532
11661	0.507	0.508	0.533
26570	0.600	0.563	0.564
10951	0.580	0.514	0.550
11879	0.508	0.507	0.537
14370	0.510	0.521	0.537
14794	0.515	0.522	0.537
17660	0.502	0.549	0.535
19396	0.507	0.507	0.536

4 containers

Edges	SF	RR	RN
23090	0.273	0.345	0.365
11661	0.298	0.346	0.353
26570	0.405	0.475	0.490
10951	0.335	0.408	0.375
11879	0.279	0.284	0.338
14370	0.270	0.304	0.331
14794	0.264	0.327	0.322
17660	0.261	0.355	0.321
19396	0.276	0.334	0.338

Figure 4.11: Splitting edges comparison of smallest-first (SF), round-robin (RR) and random (RN) into 2 and 4 containers

Create graph

	Time (ms)	%
Sequential	11592	100
Transaction-retry	8705	75.1
Deadlock-free	8413 + 2031	72.6 + 17.5

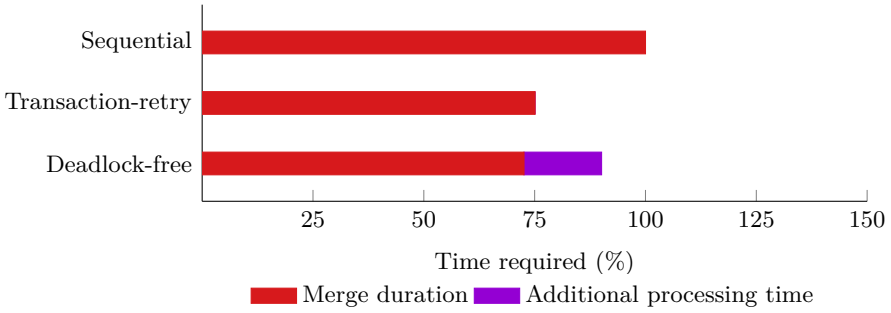
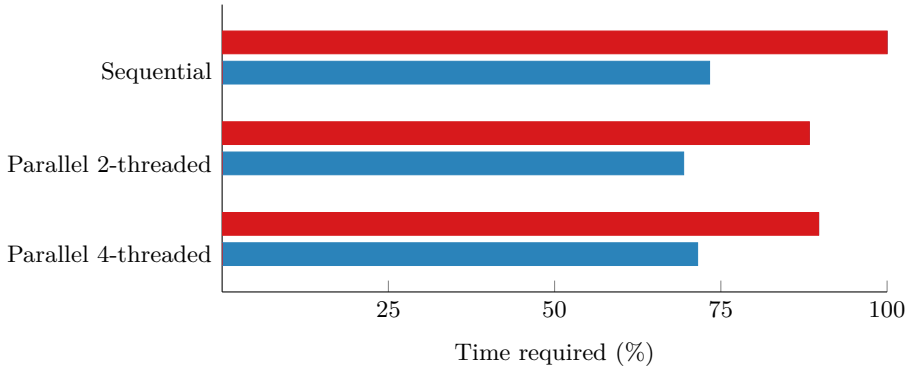


Figure 4.12: Comparison of deadlock-free and transaction-retry solutions using 4 threads

Create graph

Split	Time (ms)	%
Sequential	281	100
2 threads	248	88.3
4 threads	252	89.7

Unified	Time (ms)	%
Sequential	206	73.3
2 threads	195	69.4
4 threads	201	71.5



Update graph

Split	Time (ms)	%
Sequential	272	100
2 threads	221	81.3
4 threads	212	77.9

Unified	Time (ms)	%
Sequential	210	77.2
2 threads	158	58.1
4 threads	135	49.6

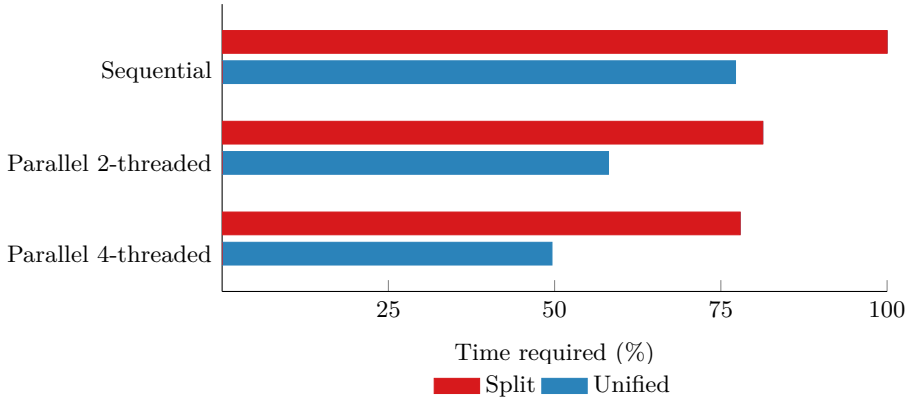
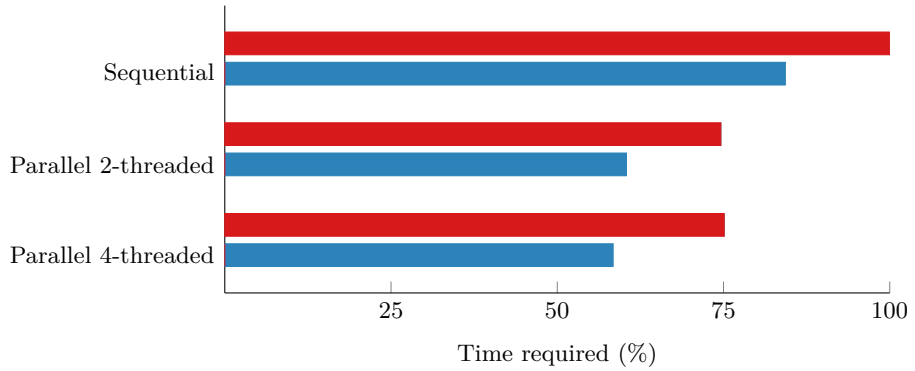


Figure 4.13: Small dataset – Parallel merging

Create graph

Split	Time (ms)	%
Sequential	11592	100
2 threads	8651	74.6
4 threads	8705	75.1

Unified	Time (ms)	%
Sequential	9777	84.3
2 threads	7002	60.4
4 threads	6775	58.4



Update graph

Split	Time (ms)	%
Sequential	15334	100
2 threads	11058	72.1
4 threads	9775	63.7

Unified	Time (ms)	%
Sequential	14568	95
2 threads	10452	68.2
4 threads	9098	59.3

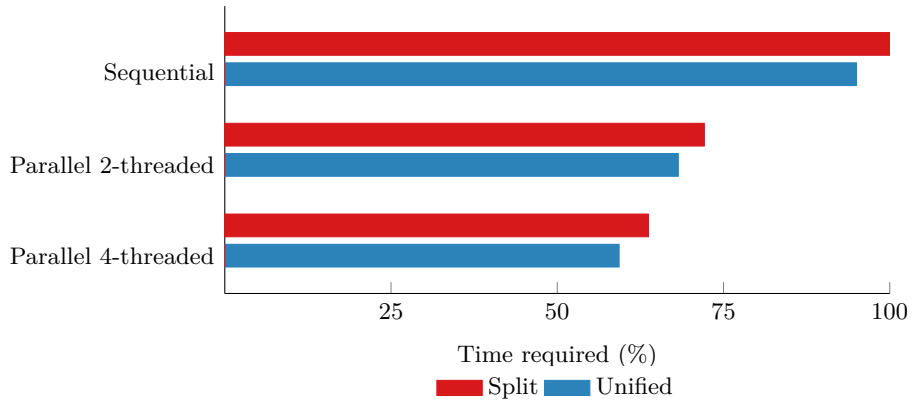
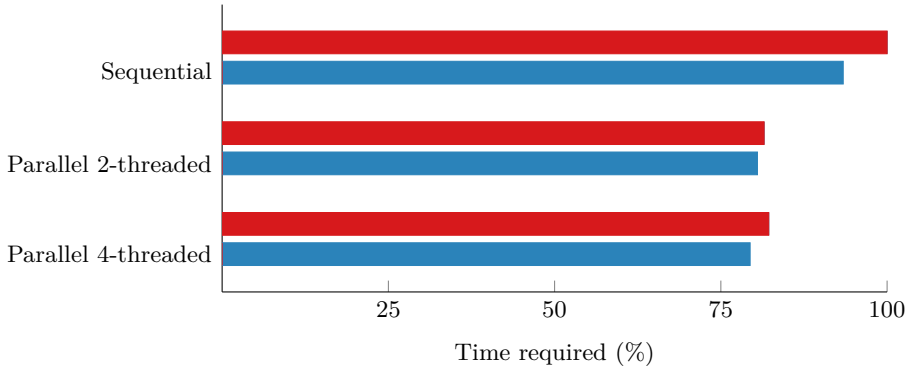


Figure 4.14: Medium dataset – Parallel merging

Create graph

Split	Time (ms)	%
Sequential	197079	100
2 threads	160584	81.5
4 threads	161924	82.2

Unified	Time (ms)	%
Sequential	183985	93.4
2 threads	158637	80.5
4 threads	156429	79.4



Update graph

Split	Time (ms)	%
Sequential	289088	100
2 threads	217728	75.3
4 threads	199866	69.1

Unified	Time (ms)	%
Sequential	276522	95.7
2 threads	197605	68.4
4 threads	185140	64

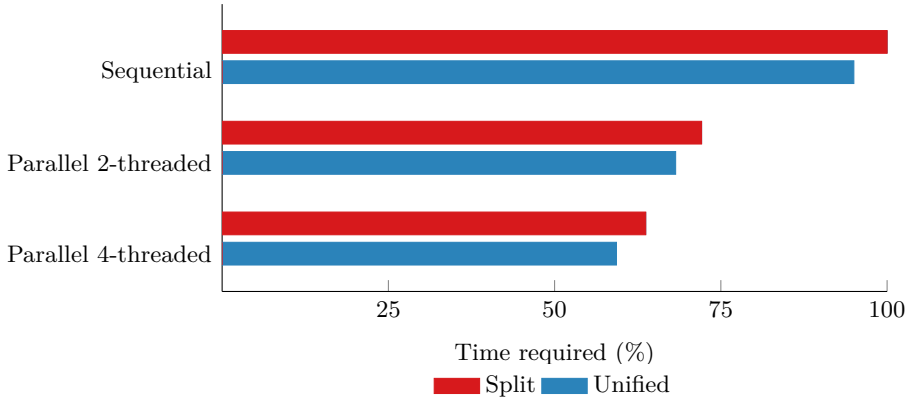


Figure 4.15: Large dataset – Parallel merging

Create graph with *mapsTo* edges

	Time (ms)	%
Titan	7395	100
Full pattern sequential	175819	2377.5
Full pattern 2-thread	112569	1522.23
Full pattern 4-thread	85425	1155.2
Imperative sequential	2045	27.65
Imperative 2-thread	1287	17.4
Imperative 4-thread	1113	15.1

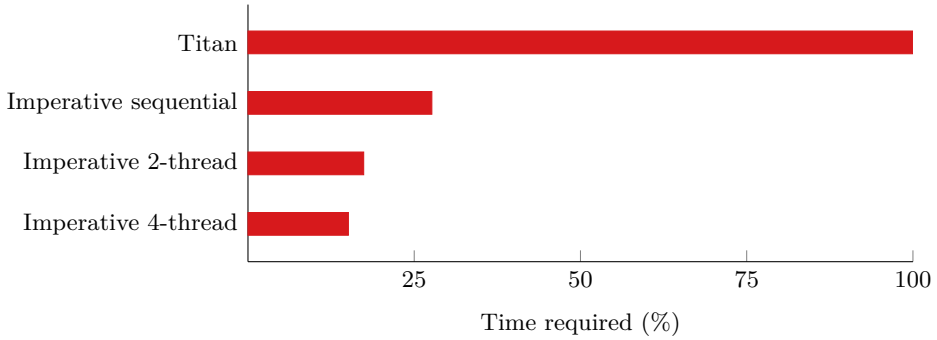


Figure 4.16: Performance of merging *mapsTo* edges (full pattern solution excluded from the graph due to readability reasons)

Create graph with *perspective* edges

	Time (ms)	%
Titan	1237	100
Sequential	1695	137.02
Parallel 2-thread	878	70.98
Parallel 4-thread	555	44.87

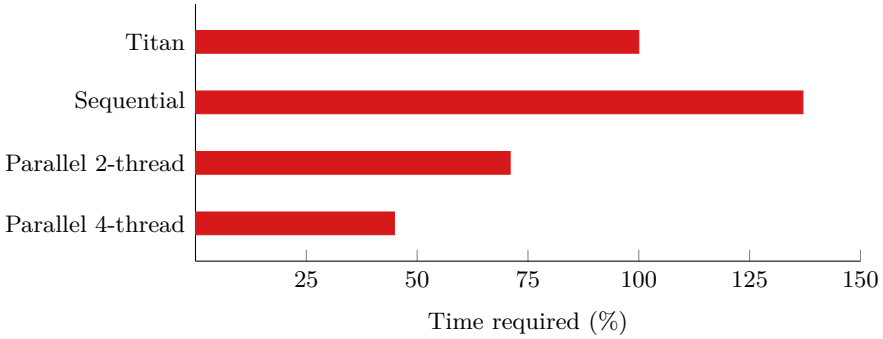


Figure 4.17: Performance of merging *perspective* edges

Create graph

Small dataset

	Time(ms)	%	Acc
Titan	737	100	1
Neo4j	201	27.3	3.67

Medium dataset

	Time(ms)	%	Acc
Titan	45115	100	1
Neo4j	6775	15	6.66

Large dataset

	Time(ms)	%	Acc
Titan	879523	100	1
Neo4j	156429	17.8	5.62

Update graph

Small dataset

	Time(ms)	%	Acc
Titan	716	100	1
Neo4j	135	18.9	5.3

Medium dataset

	Time(ms)	%	Acc
Titan	52283	100	1
Neo4j	9098	17.4	5.75

Large dataset

	Time(ms)	%	Acc
Titan	987580	100	1
Neo4j	185140	18.8	5.33

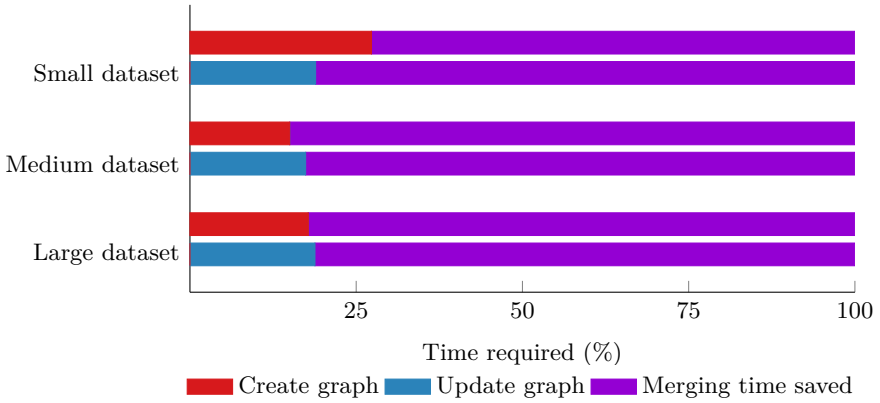


Figure 4.18: Final results (4-thread parallel Neo4j) comparison with original (Titan) implementation

4.10 Final results

The tables and the graph in the figure 4.18 show how final parallelized (4-threaded) implementation performed in comparison with original Titan implementation using all main datasets. The performance of the merging algorithm is around 5-6x better, which is quite a significant improvement.

Conclusion

The goal of this work was to analyze graph algorithms used in the MANTA Flow project and migrate it from the original Titan graph database to Neo4j graph database, and to optimize these algorithms. The only focus of this work was the merging algorithm required for the initial insertion and versioning of the analyzed data into the database, as it was quite complex.

In the first chapter, MANTA Flow project as a whole was described, the model of the data used and the fundamental properties of both Titan and Neo4j graph databases. The last part focused on the description of the original merging algorithm and all the related parts.

The second chapter described the analysis of the possible optimization steps and the migration of the algorithm to Neo4j database. The migrating part focused on the difference of the imperative approach used for querying in Titan and the declarative approach used by Cypher. The optimization analysis was split into 2 phases, where one was an initial proof of concept for Titan optimization created due to presumption of quick possibility of parallelization using Titan's threaded transaction interface. The second part focuses solely on the optimizations within migrated Neo4j version, and is divided into part focusing on minor optimizations by removing redundant processing, and the parallelization part focusing on the parallelization of the database processing within the algorithm. The analysis chapter was followed by the third chapter describing the implementation details of the previously analyzed parts.

The last chapter described the problems with the parallelization of the Titan implementation and the performance results of the migrated Neo4j implementation and its optimized versions. The performance achieved was significantly better than when using the original Titan implementation due to both Neo4j query times and the optimizations introduced.

The goal of this work was fulfilled as every part of it was carried out. The implemented version of the merging algorithm is not yet deployed in the production as the migration of the whole project to a new graph database is still ongoing, but this is one of the initial steps required.

Bibliography

[1] Titan Data Model. [cit. 2021-03-25]. Available from: http://s3.thinkaurelius.com/docs/titan/1.0.0/data-model.html

[2] documentation, T. Architectural Overview. [cit. 2021-03-31]. Available from: http://s3.thinkaurelius.com/docs/titan/1.0.0/arch-overview.html

[3] MANTA. March 2021, [cit. 2021-03-25]. Available from: https://getmanta.com/about-us/

[4] MANTA Supported technologies. March 2021, [cit. 2021-03-25]. Available from: https://getmanta.com/technologies/

[5] Holeček, P. *Temporal data in graph database of project Manta*. Master's thesis, Czech Technical University in Prague, 2015, [cit. 2021-03-25].

[6] Sýkora, J. *Incremental update of data lineage storage in a graph database*. Master's thesis, Czech Technical University in Prague, 2018, [cit. 2021-03-25].

[7] Titan. [cit. 2021-03-25]. Available from: http://titan.thinkaurelius.com/

[8] Laskowski, J. About Titan. [cit. 2021-03-25]. Available from: https://jaceklaskowski.gitbooks.io/titan-scala/content/titan-about_titan.html

[9] Titan Schema. [cit. 2021-03-25]. Available from: http://s3.thinkaurelius.com/docs/titan/1.0.0/schema.html

[10] Blueprints. [cit. 2021-03-25]. Available from: https://github.com/tinkerpop/blueprints

[11] LaRocque, D. Titan Blueprints. [cit. 2021-03-25]. Available from: http://titan.thinkaurelius.com/wikidoc/0.4.4/Blueprints-Interface.html

[12] Titan Storage Backends. [cit. 2021-03-25]. Available from: http://s3.thinkaurelius.com/docs/titan/1.0.0/storage-backends.html

[13] Using Persistit. [cit. 2021-03-25]. Available from: http://titan.thinkaurelius.com/wikidoc/0.4.4/Using-Persistit.html

[14] Getting Started. [cit. 2021-04-29]. Available from: https://tinkerpop.apache.org/docs/3.4.10/tutorials/getting-started/

[15] Index Backends. [cit. 2021-03-25]. Available from: http://s3.thinkaurelius.com/docs/titan/1.0.0/index-backends.html

[16] Apache Lucene. [cit. 2021-03-25]. Available from: https://lucene.apache.org/

[17] Transactions. [cit. 2021-03-25]. Available from: http://s3.thinkaurelius.com/docs/titan/1.0.0/tx.html

[18] Neo4j. [cit. 2021-03-25]. Available from: https://neo4j.com/developer/neo4j-database/

[19] Vukotic, A.; Watt, N.; et al. *Neo4j in Action*. [cit. 2021-03-25]. Available from: https://livebook.manning.com/book/neo4j-in-action/chapter-10/43

[20] The traversal framework. [cit. 2021-03-25]. Available from: https://neo4j.com/docs/java-reference/current/traversal/

[21] Cypher Query Language. [cit. 2021-03-25]. Available from: https://neo4j.com/developer/cypher/

[22] Chao, J. Graph Databases for Beginners: Native vs. Non-Native Graph Technology. [cit. 2021-03-25]. Available from: https://neo4j.com/blog/native-vs-non-native-graph-technology/

[23] Index configuration. [cit. 2021-03-25]. Available from: https://neo4j.com/docs/operations-manual/current/performance/index-configuration

[24] Rocha, J. Neo4j current transaction commit process order. [cit. 2021-03-25]. Available from: https://neo4j.com/developer/kb/neo4j-current-transaction-commit-process-order/

[25] MERGE. [cit. 2021-03-25]. Available from: https://neo4j.com/docs/cypher-manual/current/clauses/merge/

[26] Cordella, L. P.; Foggia, P.; et al. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, volume 26, no. 10, 2004: pp. 1367–1372, doi: 10.1109/TPAMI.2004.75.

[27] Muzammil, U. Shared vs Exclusive Transaction locks. [cit. 2021-03-25]. Available from: https://neo4j.com/developer/kb/shared-vs-exclusive-transaction-locks/

[28] Kernighan, B. W.; Lin, S. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, volume 49, no. 2, 1970: pp. 291–307, doi:10.1002/j.1538-7305.1970.tb01770.x.

[29] Fiduccia, C.; Mattheyses, R. A Linear-Time Heuristic for Improving Network Partitions. In *19th Design Automation Conference*, Los Alamitos, CA, USA: IEEE Computer Society, jun 1982, ISSN 0146-7123, pp. 175,176,177,178,179,180,181, doi:10.1109/DAC.1982.1585498. Available from: https://doi.ieeecomputersociety.org/10.1109/DAC.1982.1585498

[30] Procedures and functions. [cit. 2021-03-31]. Available from: https://neo4j.com/docs/java-reference/current/extending-neo4j/procedures-and-functions/

Appendix A

Acronyms

API Application programming interface

CPU Central processing unit

DFS Depth-first search

ETL Extract, transform and load

JVM Java virtual machine

RAM Random-access memory

SSD Solid-state drive

UUID Universally unique identifier

Appendix B

Contents of enclosed CD

- `readme.txt` ..... contents description
- `src` ..... the directory of source codes
  - `java` ..... implementation sources
  - `thesis` ..... the directory of LaTeX source codes of the thesis
    - `figures` ..... the figures used in the thesis
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format
  - `thesis.ps` ..... the thesis text in PS format