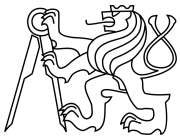
CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY



ASSIGNMENT OF MASTER'S THESIS

Title:	Incremental update of data lineage storage in a graph database
Student:	Bc. Jan Sýkora
Supervisor:	Ing. Michal Valenta, Ph.D.
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of summer semester 2018/19

Instructions

1. Get familiar with the Manta project, esp. with the persistent data flow storage module (data lineage) in the graph database and learn how these data flow changes are stored over time.
2. Study methods and procedures of a subgraph update, i.e., an incremental update. For the inspiration use version control systems, methods of incremental backup in databases, and the thesis [1].
3. Analyse the possibilities of incremental update of a graph representing data flows and discuss their suitability for the Manta project. Important criteria are the number of necessary changes in a graph, the ability to find data flows related to a certain time interval, and an amount of time necessary for the update as well as for the querying. Use the results of the analysis to design your own method. The input of the method is a subgraph to be updated.
4. Implement a prototype, carry out data performance testing on data provided by the supervisor.
5. Discuss the results and possibly suggest improvements.

References

[1] Holeček Petr: Temporální data v grafové databázi v projektu Manta. Magisterská práce na FIT ČVUT. 2015

Ing. Michal Valenta, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague October 3, 2017

Czech Technical University in Prague
Faculty of Information Technology
Department of Software Engineering

Master's thesis

Incremental update of data lineage storage in a graph database

Bc. Jan Sýkora

Supervisor: Ing. Michal Valenta, Ph.D.

26th April 2018

Acknowledgements

I would like to express my sincere gratitude to my advisor RNDr. Lukáš Hermann and to my supervisor Ing. Michal Valenta, Ph.D. who provided insight and expertise that greatly assisted the research. I would like to also thank all others who supported me and motivated me in overcoming numerous obstacles I have been facing through my research.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 2373 of the Act No. 89/2012 Coll., the Civil Code, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the Work), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization 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 Work, combine it with another work, and/or include the Work in a collective work.

In Prague on 26th April 2018 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Sýkora, Jan. *Incremental update of data lineage storage in a graph database.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2018.

Abstrakt

Cílem této práce je analýza a implementace inkrementálních aktualizací úložiště data lineage v softwarovém nástroji Manta Flow. Obsahem této práce je studium současného úložiště data lineage v Manta Flow, zkoumání existujících řešení inkrementálních atualizací v systémech řízení verzí, zkoumání inkrementálního zálohování v databázích, analýza a návrh nového řešení inkrementálních aktualizací v Manta Flow a následná imlementace prototypu a provedení výkonnostního testování. Výsledný prototyp je možné nasadit do existujícího produktu Manta Flow, a tím řádově snížit časovou složitost aktualizací v úložišti data lineage.

Klíčová slova Inkrementální aktualizace, data lineage, datové toky, grafová databáze, Manta Flow

Abstract

The purpose of this thesis is to analyze and implement incremental updates of data lineage storage in the software tool Manta Flow. The basis of this work is the study of current data lineage storage in Manta Flow, research of existing solutions of incremental updates in version control systems, research of incremental backups in databases, analysis and design of a new solution of incremental updates in Manta Flow and a subsequent prototype implementation

and performance testing execution. The resulting prototype can be deployed into the existing Manta Flow product, reducing time complexity of updates in data lineage storage in orders of magnitude.

Keywords Incremental updates, data lineage, data flows, graph database, Manta Flow

Contents

List of Figures

List of Tables

Introduction

The ever-increasing number of applications and the complexity of software systems is leading to significant growth in the total volume of data. Since systems do not usually work with their own data, but with other application data, interconnections between these systems arise and these data are affected by each other. In the case of complex systems, bringing a single error to the system may cause multiple problems.

This issue of data volume is addressed by Manta Flow. Manta Flow is a software tool that allows automatic analysis of programming code and the subsequent description of transformation logic contained therein. Manta Flow is able to automatically process large amount of data and construct from these data a visualized map of data flows across the BI environment, in other words, data lineage.

Manta Flow stores results of the analyzed data, metadata, in a graph database. Thanks to the thesis of Petr Holeček [1], temporality was implemented in Manta Flow. Thus, it is possible to track history of stored metadata in time (including time history of data flows).

Unfortunately, the implemented temporality in Manta Flow supports only full update. When a change is made in the unprocessed data, all unprocessed data (changed and unchanged) have to be processed and stored in database to save the change in time. In other words, time spent on updating a change is directly proportional to the overall size of the all data, regardless the change size.

This size change insensitivity causes undesired consequences. When for instance only 1 % of all data is changed, the update still requires processing all 100 % of data. Therefore, in this case, the update takes about one hundred times longer than necessary.

This thesis addresses the issue of change size insensitivity by introducing incremental update. Incremental update is based on the principle of updating only the changed parts, avoiding unnecessary update of the unchanged parts. Hence, when using incremental update, the update time is directly

proportional to the change size, regardless the size of all data.

Since Manta Flow works with a large amount of data and changes in single updates represent typically only a fraction all data, incremental update can significantly reduce the time necessary for the update, thereby supporting scalability with the growing data size.

The main goal of this thesis is to implement incremental update of data lineage storage in Manta Flow. In the first phase, a study of the current data lineage storage in Manta Flow is performed. On top of that, research of existing solutions of incremental updates is analyzed. Namely, incremental updates in version control systems and incremental backups in databases have been researched and can be used as the inspiration for incremental updates in Manta Flow. In the second phase, based on the study and research from the first phase, a new incremental update method in Manta Flow is analyzed and designed. In the third phase, an implementation of a prototype supporting the designed incremental update process is performed and finally a performance testing of the implemented prototype is carried out.

If the implemented prototype proves the expected update time reduction and does not indicate any negative side effects disallowing it for real use, then the new incremental update method can be deployed and used in the product version of Manta Flow.

Chapter 1

Background

In the first chapter, we will study software tool Manta Flow. We will focus on the **module of persistent data flow storage**. This module is responsible for managing data and storing them in a **graph database Titan** in such a specific way allowing it to track the **data lineage**.

We will learn about the architecture of graph database Titan, we will study types of vertices, edges and their relationships and we will also cover types of indices used in the database.

Finally, we will study the way the **version control** is implemented in Manta Flow and we will analyze it for further improvements. In other words, we will define what functionality we want to improve within this thesis and why we want to improve it.

1.1 Manta Flow

Company Manta Tools introduced in 2013 one of its main products – Manta Flow [2]. Manta Flow analyzes customers' system metadata (for instance Manta parses complex SQL scripts or ETL configuration), extracts information of lineage and data flows and visualizes the entire path of data on a table or column level [3].

Simply put, as Lukáš Hermann explains [4], Manta Flow can reduce very complex SQL statements to a few simple rectangles connected by arrows. It helps to quickly understand what source tables the SQL queries read, what target tables they fill, what columns are involved in computing a particular column and how these columns are involved.

Therefore, Manta Flow is intended for clients with huge data warehouses who need to get end-to-end data lineage. It allows them to improve their data governance, fulfill compliance regulations and increase effectivity of their current solutions [5].

1.1.1 Manta Flow architecture

Manta Flow is a client-server based application. Communication between client and server is performed via a specified communication interface using the TCP/IP protocol.

Manta Flow client is a Java command line application. Manta Flow client *extracts* database dictionaries, DDL scripts, work flows and configuration files from user's databases and repositories, *analyzes* the extracted data and *uploads* all the gathered metadata to the Manta Flow server.

Manta Flow server is a java server application. Manta Flow server *processes* metadata received from the client, *stores* them in a metadata repository, in the graph database Titan, and *provides visualization* of the stored metadata.

Both client and server contain multiple modules. In this chapter, we will focus solely on the module of persistent data flow storage. Module of the persistent data flow storage is located on Manta Flow server. This module encapsulates persistent metadata storage and all logic related to the management of this persistent storage.

1.2 Graph database Titan

Manta Flow uses for the persistent data flow storage a graph database Titan. Titan is a distributed graph database optimized for storing and querying graphs represented over a cluster of machines [6]. However, in the Manta Flow project the distributivity is not used. Every instance of Manta Flow is running exactly on one machine.

Titan is also a **transactional database**. Every graph operation in Titan occurs within the context of a transaction. Each thread opens its own transaction against the graph database with the first operation on the graph [7]. The support of transactions allows execution of queries of concurrent users against the database at the same time, which is used in Manta Flow.

1.2.1 Titan architecture

Titan is a graph database engine. Titan is focused on compact graph serialization, graph data modeling and efficient query execution. Moreover, Titan utilizes Hadoop for graph analytics and batch graph processing. Titan implements robust, modular interfaces for data persistence, data indexing and client access [8]. Figure 1.1 taken from the official Titan documentation [8] describes Titan architecture.

Although Titan supports online analytical processing (OLAP), using Big Data platforms such as Hadoop, none of these tools are used in the Manta Flow.

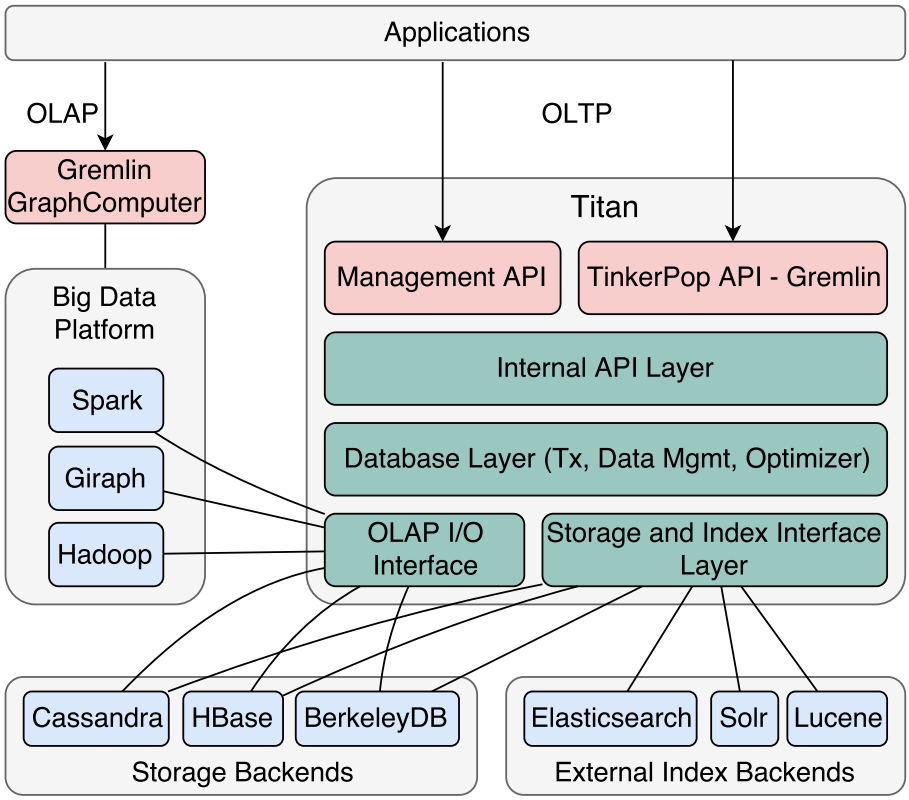


Figure 1.1: High-level Titan architecture and Context

Titan has a pluggable storage architecture which allows it to build on a proven database technology such as Apache Cassandra or Oracle BerkeleyDB [6]. Manta Flow is using **storage backend Persistit**. Storage backend Persistit runs in the same JVM as Titan and provides local persistence on a single machine. As Dan LaRocque, a former co-developer on Titan, mentions in the documentation [9], all data has to fit on the local disk and all frequently accessed graph elements have to fit into the main memory. Therefore, the graphs stored in the Persistit are limited to hundreds of millions of vertices. According to Lukáš Hermann, the director of engineering and the project leader of Manta Flow, the diagrams currently stored in the Persistit reach the maximum of millions of vertices. Hence, Persistit is fully capable of storing client's data on a single machine with a sufficient store for a new data.

Manta Flow is currently using Titan **version 0.4.4**. When designing Manta Flow it was decided to use the storage backend Persistit. The Titan version 0.4.4 is the last version supporting the storage backend Persistit. Therefore, the Titan version 0.4.4 was chosen despite the existence of newer versions of Titan.

As mentioned in the Titan documentation [10], Titan natively implements

the **Blueprints interface**[1]. Hence, Titan supports all of the open-source technologies in the TinkerPop graph stack. The hierarchy of a graph computing framework TinkerPop is described in the figure 1.2.

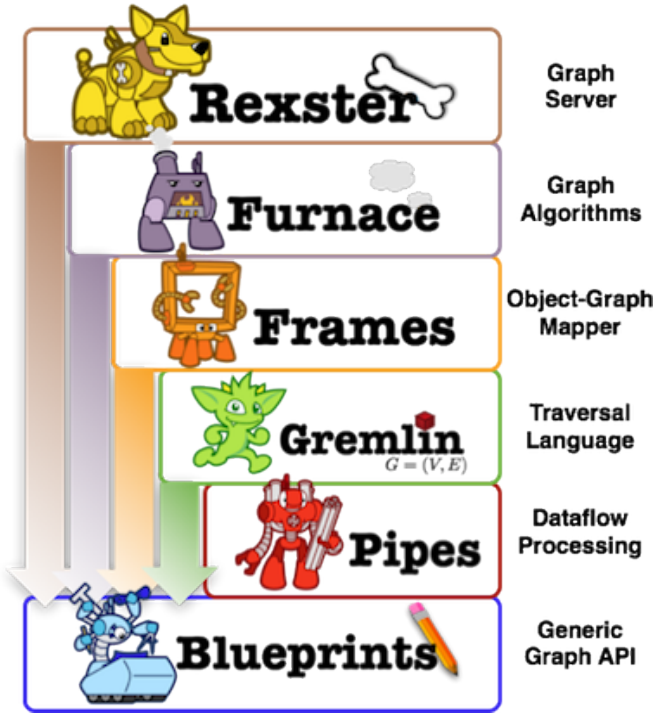


Figure 1.2: TinkerPop system architecture

Applications, such as Manta Flow, are able to interact with Titan in two ways: either **executing Gremlin queries** directly against the graph within the same JVM or interact with a local or remote Titan instance by **submitting Gremlin queries** to the server [8].

In Titan you can use both external index interface as well as the standard index. The indexing in Titan and its specific implementation in Manta Flow is explained further in detail in the section 1.9.

1.2.2 Physical data model

Titan stores a graph in any storage backend that supports the **BigTable data model** [12]. BigTable is a distributed storage system for managing structured data that is designed to scale to a very large size: petabytes of data across thousands of servers [13].

In the BigTable data model, each table is a collection of rows. Each row is uniquely identified by a key. Each row contains an arbitrary number

[1]Blueprints is an open source, community developed Java interface for graph databases that expose a property graph data model [11].

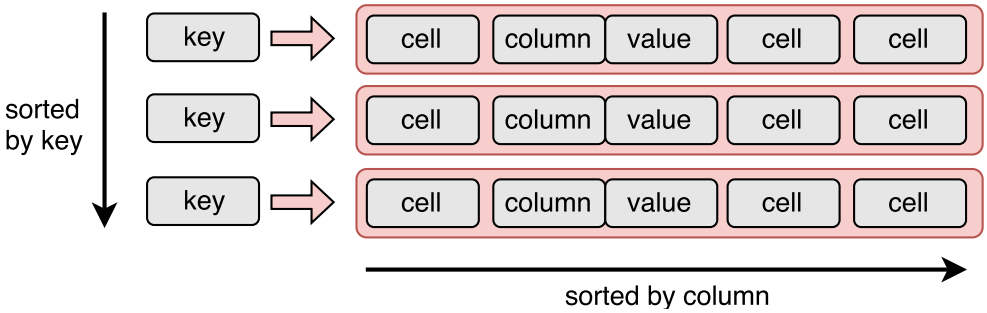


Figure 1.3: BigTable data model

of cells. A cell is composed of column and value. Titan requires that cells must be sorted by their columns. Additionally, depending on implementation of BigTable by the storage backend, rows may be kept sorted in the order of their key [12].

Using BigTable data model as a storage concept, Titan stores graphs in the **adjacency list format**. In other words, a graph is stored as a collection of vertices with their adjacency list. The adjacency list of one vertex contains all of the vertex' incident edges and vertex' properties. Each edge and property is stored as one cell with a predefined structured such that the byte order of the column respects the sort key of the edge label (see figure 1.5) [12].

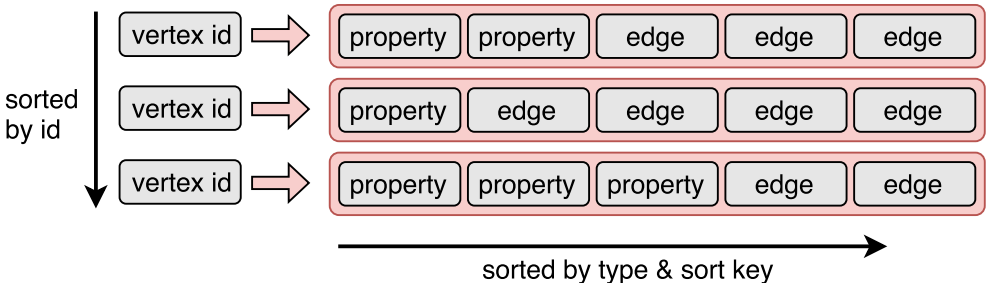


Figure 1.4: Titan data layout

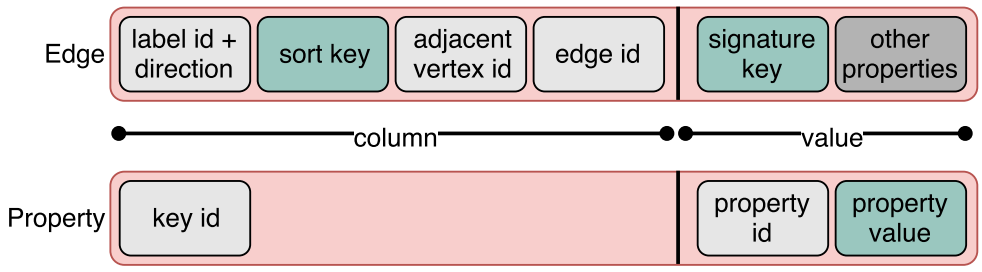


Figure 1.5: Edge and property layout

The disadvantage of the adjacency list format is that every edge is stored twice – in vertex where the edge starts and in vertex where the edge ends. On the other hand, the adjacency list format allows Titan to effectively traverse large graphs, perform efficient insertions and deletions or use, for instance, vertex-centric indexing (for more information about vertex-centric indexing see section 1.9.3).

1.2.3 Logical data model

The logical (abstract) data model in Titan is a **property graph**. Since Titan implements the Blueprints interface, we can entirely define the data model accordidng to the Gremlin's[2] definition of a property graph [14]. Gremlin defines property graph as a **key/value-based**, **directed** and **multi-relational graph**. The first term, key/value-based graph, refers to the fact that both vertices and edges can have any number number of properties associated with them. The second term, directed graph, refers to the fact that each edge is oriented. Finally, the third term, multi-relational graph, describes the fact that between two vertices can exist more than one edge. Hence, between two vertices can exist more than one relationship. An example of a property graph is in the figure 1.6.

1.3 Data model in Manta Flow

Using the property graph as a data model, Manta Flow defines its own structure and hierarchy of vertices and edges. In the following section we will describe all types of vertices and edges in Manta Flow and explain their relationships.

1.3.1 Vertices

As mentioned above in the description of the abstract data model, vertices can have any number of properties. Properties can also be used as a vertex label. In other words, we can reserve one property to be defined in all vertices to distinguish or categorize types or groups of vertices. This approach is used in Manta Flow and this property is named `vertexType`. Currently, in Manta Flow are defined **nine vertex types**, each with their own corresponding properties:

- super root
  - superRoot
- resource
  - resourceName

[2]Gremlin is a graph traversal language of Blueprints TinkerpPop framework.

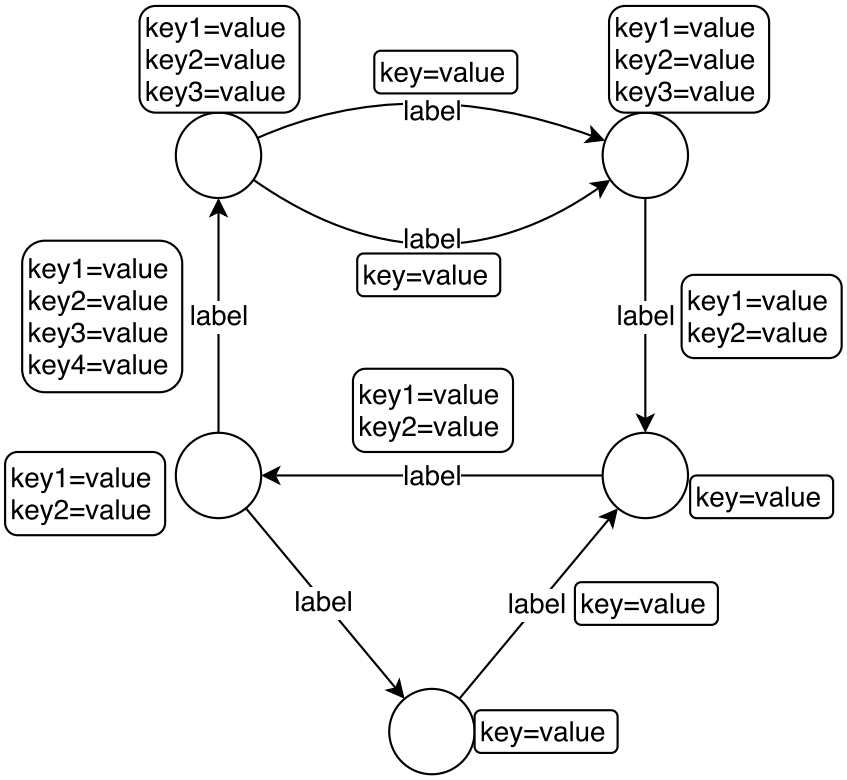


Figure 1.6: Example of a property graph

  - resourceType
  - resourceDescription
- layer
  - layerName
  - layerType
- node
  - nodeName
  - nodeType
- attribute
  - attributeName
  - attributeValue
- revision root
  - revisionRoot
- revision node
  - revisionNumber
  - revisionCommitted
  - revisionEnd
- source root
  - sourceRoot

- source node
  - sourceNodeLocal - local name of the source code file
  - sourceNodeId - unique name of the source code file
  - sourceNodeHash - hash code of the content of the source code file

> **Vertex vs. Node**
>
> It is important to distinguish between the terms *vertex* and *node* in the Manta Flow terminology. *Vertex* is every vertex in the database. *Node* is a type of vertex. It may be confusing due to the names of vertex types *revison node* and *source node* that are not nodes but only vertices.

Super root is the root of the stored metadata. Hence, all user's metadata are below super root in the vertex hierarchy. In the database exists only one vertex of this type. Super root has only one property `superRoot`. This property has no other usage than as a single-property index.

Resource represents a specific resource of metadata or its part in the repository. It is a technology. Resource can be for instance Oracle, Teradata, Hive or just a Filesystem. Every *node* belongs to a certain resource. To which resource a node belongs to is defined either directly or indirectly. For more information see section 1.4.

Layer represents a logical level of metadata stored in the database. Purpose of the layer is to distinguish between different views of the modeled reality. Thus, users are allowed to view their stored metadata from different perspectives. Especially, users are allowed to view the data flows within a specific layer. For example, user has a modeled data on two levels: physical and business layer. Then, user is able to view the same data flows on the physical level (e.g. flows between tables) and on the business level (e.g. flows between entities corresponding to the tables from the physical layer).

Node represents an object in the data flow graph. A node may be for instance a database, schema, table or column. Node can also represent a procedure, PLSQL script, file, directory or any other user defined object in the data flow graph.

Node has two properties `nodeName` and `nodeType`. For instance node representing a table has properties `nodeName=EMPLOYEES` and `nodeType=Table`. Property nodeName is also indexed for a full-text search.

Attribute represents a supplementary information of a node. For instance, when a column has a data type `VARCHAR(20 BYTE)`, then in the data model a node of `nodeType=Column` is connected with an attribute vertex and this attribute vertex has two properties: `attributeName=COLUMN_TYPE` and `attributeValue=VARCHAR(20 BYTE)`.

Revision root is the root of the tree with revision nodes. In the database exists only one revision root. Revision root has only one property `revisionRoot`. It is a single-property index for quick search of the revision root.

Revision node is, in the Manta Flow terminology, not a node but a vertex, however this is the official name of this type of vertex, hence, we will keep using this name. Revision node represents a revision (see section 1.10.1 for more information about revisions).

Revision node contains three properties. Property `revisionNumber` stands for the number of revision, `revisionCommitted` is a boolean property and its value is true, when the revision is committed, otherwise its value is false. Finally, property `revisionEnd` represents time when the revision was committed.

Source root is the root of the tree with source nodes. In the database exists only one source root. Source root has only one property `sourceRoot`. It is a single-property index for quick search of the source root.

Source node is, in the Manta Flow terminology, not a node but a vertex, however this is the official name of this type of vertex, hence, we will keep using this name.

Source node represents a source code file. Source code files are typically database scripts modifying user's database, and thus affecting the data flows in user's database.

Source node contains three properties. Property `sourceNodeLocal` contains name and path to the source code file in the user's file system. Second property, `sourceNodeId`, is a unique, randomly generated, name of the source code file in Manta Flow. SourceNodeId is used for identification of the source code file within the Manta Flow. Finally, property `sourceNodeHash` is a hash code of content of a source code file. This hash code is used for check of change during the asynchronous loading of source code files from user's file system to Manta Flow.

For more information about management of source nodes and source code files in Manta Flow see section 1.7.

1.3.2 Edges

Edges in Titan can have both labels and properties. The only constraint is that one edge can have at most one label. Manta Flow defines **nine different edge labels** (i.e. edge types), each with their corresponding properties:

- hasResource
  - childName
  - tranStart
  - tranEnd
- hasParent
  - childName
  - tranStart
  - tranEnd
- hasAttribute

  - tranStart
  - tranEnd
- directFlow
  - targetId
  - interpolated
  - tranStart
  - tranEnd
- filterFlow
  - targetId
  - interpolated
  - tranStart
  - tranEnd
- inLayer
  - tranStart
  - tranEnd
- mapsTo
  - tranStart
  - tranEnd
- hasRevision
  - tranStart
  - tranEnd
- hasSource
  - sourceLocalName
  - tranStart
  - tranEnd

> **tranStart and tranEnd**
>
> Every label has two properties **tranStart** and **tranEnd**. These properties specify the validity of each object in time. It specifies, when the object was created and removed. These properties are also indexed (see section 1.9 for more information). Since storage of the time validity of objects in the database is the core of our thesis, these properties are described in detail in the section 1.10.2.

HasResource edge connects node with its resource. This edge directly defines node's resource. The edge starts from the node vertex and ends in the resource vertex. For more information about node's resource see section 1.4.

The edge with label `hasResource` has one more special usage. It is also connecting resource with super root. The edge starts from the resource vertex and ends in the super root vertex.

In both cases described above, the edge with label `hasResource` has a property `childName`. This property contains name of the source vertex, i.e. a name

of the vertex from which the edge starts. This property is also indexed for a quick search.

HasParent edge connects child node with its parent node (predecessor). The edge starts from the child node and ends in the parent node. Every node has at most one predecessor. For example, a child node Column has a parent node Table and node Table has a parent node Schema.

The edge with label `hasParent` has, as well as the edge `hasResource`, a property `childName`. This property contains name of the child node. This property is indexed for a quick search.

HasAttribute edge connects node with its attribute. The edge starts from the node and ends in the attribute vertex.

DirectFlow edge represents a direct data flow from the source node to the target node. A direct data flow is defined as a data flow from the source node to the target node. These data may be transformed (filtered, sorted, changed format etc.) on their way to the target, however, unlike the `filterFlow`, this data flow does not affect another data flow.

The edge `directFlow` connects only nodes on the lowest level of the graph hierarchy (e.g. database columns or function parameters).

`DirectFlow` edge has two properties, `targetId` and `interpolated`. `TargetId` property contains the automatically generated vertex id of the target node. This property is indexed for a quick search. The second property, `interpolated`, is a boolean property indicating whether the edge was interpolated (`true`) or not (`false`).

FilterFlow edge represents an indirect data flow from the source node to the target node. Unlike the direct data flow, in case of the filter (indirect) data flow, the source node affects, indirectly, what data flows to the target node. For example, source node represents a part of the script where is the `IF` statement. This statement decides, whether some data will flow to some target node.

On the `filterFlow` edge are applied the same constraints (connecting nodes on the lowest level) and properties (`targetId` and `interpolated`) as on the `directFlow` edge.

InLayer edge connects a `resource` (source vertex) with a `layer` (target vertex). This edge specifies, to what layer belong metadata from the same specific resource (for more information see vertex `layer` in the section 1.3.1).

MapsTo edge represents relationship between nodes from different layers. This edge specifies that the source node from layer X is mapped to the target node from layer Y. For edges `mapsTo` applies the same restriction as for the edges `directFlow` and `filterFlow` that this type of edge can be present only on the lowest level of the hierarchy in both layers.

HasRevision edge connects `revision root` (source vertex) with a `revision node` (target vertex). This edge has no other purpose than to only keep the `revision nodes` accessible from the `revision root`.

HasSource edge connects `source root` (source vertex) with a `source node` (target vertex). This edge has no other purpose than to only keep the `source nodes` accessible from the the `source root`. Additionally, this type of edge has a property `sourceLocalName`. This property contains local name of the source code (name of the source code file in the graph database). This property is indexed for a quick search.

1.4 Determination of a node's resource

As mentioned in the section Vertices, every node has some resource. It is defined either directly or indirectly. Direct definition is performed simply by the edge with label *hasResource*. When the node is not connected directly, then its resource is derived indirectly by its predecessor. This is the complete pseudo-algorithm of determining a node's resource:

Algorithm 1 Find resource of a node

```
1: function getResource(node)
2:     if node has hasResourceEdge then
3:         resource ← hasResourceEdge.targetVertex
4:         return resource
5:     else if node has hasParentEdge then
6:         parent ← hasParentEdge.targetVertex
7:         return getResource(parent)
8:     else
9:         ERROR
10:    end if
11: end function
```

As we can see from the algorithm 1, if a node does have a *hasResource* edge, then its resource is represented by the end node of the *hasresource* edge. Otherwise, we search for a resource of its parent node. Since every node has an edge either to a parent node or to a resource, every node has one and only one resource. In other words, if a node does not have a *hasResource* edge, then some of its predecessors have this edge and we only need to find this predecessor using edges with label *hasParent*.

For instance, in the figure 1.7 node B does not have a *hasResource* edge. Therefore, we proceed to its parent node A. There we find out, parent node A has a *hasResource* edge to the *resource* 1. Hence, node B has a resource *resource*1.

However, it can happen a node has a different resource than some of its predecessors. For example, in the same figure 1.7, node D has a resource different from its parent node C.

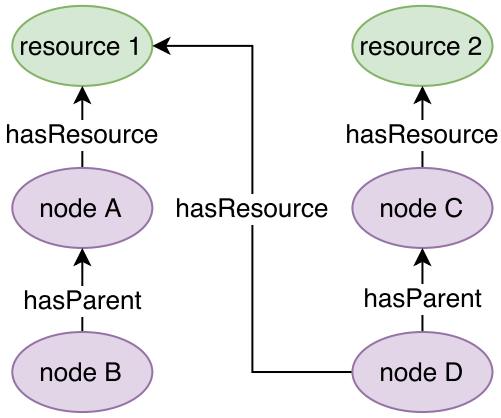


Figure 1.7: Example of a relationship between a node and resource

1.5 Graph structure

In this section, we will describe the overall graph structure composed of the previously defined vertices and edges. The graph structure in Manta Flow is split into three unconnected graphs: **main graph**, **graph with source nodes** and **graph with revisions**.

1.5.1 Main graph

Main graph contains all user's stored metadata and all data flows between them. The hierarchy of vertices and their relationships in the graph is shown in the figure 1.8.

In our visualization, super root is always on top of the main graph. To the super root are connected resources (e.g. Oracle or Teradata) and resources are connected to their layer (e.g. physical or business layer). To the resources are connected nodes and nodes may have some additional attributes. Finally, all the nodes on the lowest level are connected by the flow edges representing data flows between these nodes.

Depth of the main graph is derived from the complexity and granularity of user's metadata. Therefore, depth of the main graph is in general unlimited. For instance, when representing relational database data model consisting only of node types database, schema, table and column, the depth of the graph will be at most 5 (super root is at depth 0, resource is at depth 1, database at depth 2, schema at depth 3, table at depth 4 and column at depth 5). On the other hand, when representing a directory structure, depth of the graph depends on the depth of the deepest subdirectory.

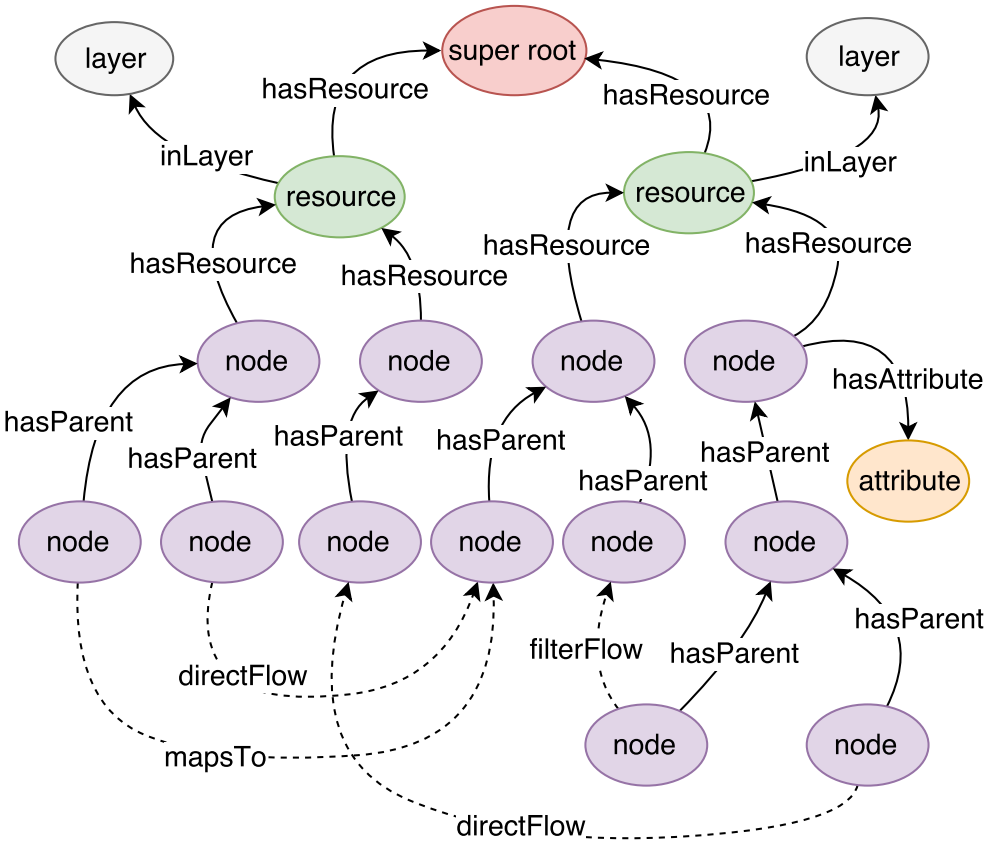


Figure 1.8: Example of a main graph hierarchy

1.5.2 Graph with source nodes

Graph with source nodes consists of a source root and source nodes connected to the source root. This simple structure is showed in the figure 1.9.

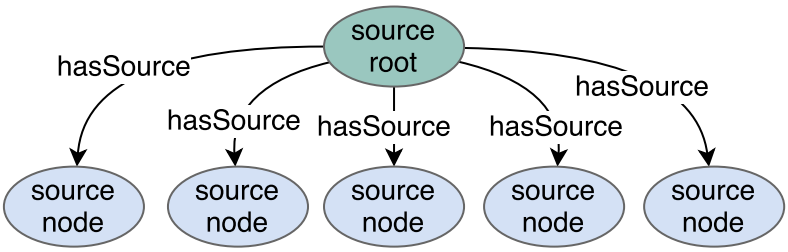


Figure 1.9: Hierarchy of a graph with source code files

Source code files are not stored in the graph database. Therefore, this graph stores references to the physical location of source code files in the source nodes.

For more information about relationship between source nodes and source code files see section 1.7.

1.5.3 Graph with revisions

Graph with revisions has a similar structure as the graph with source code files. On top of the graph is revision root. Revision root is connected with all revision nodes. Revision nodes only hold information about a specific revision, there are no other connections in this graph structure. This structure is showed in the figure 1.10.

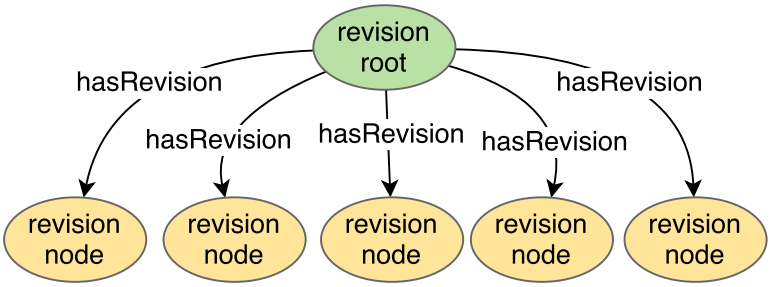


Figure 1.10: Hierarchy of a graph with revisions

Graph with revisions serves no other purpose than storing list of revisions in Manta Flow. Version control itself is implemented in the main graph using properties tranStart and tranEnd on the edges. See section 1.10 for more information about version control in Manta Flow.

1.6 Graph creation

To visualize data flows of user's metadata, we have to first create a graph of metadata in Manta Flow Titan database.

First, on user's side has to be created a **local graph structure** of his or her metadata and data flows. Then this structure is linearized into the **CSV file** and this file is sent to Manta Flow server. The received CSV file is parsed to single vertices and edges and these vertices and edges are **merged** one by one into Titan graph database.

Merging starts from the root and proceeds to the leaves as the processor goes through the CSV file line by line. In the CSV file one line represents one edge or vertex.

Merging is performed by database **transactions**, i.e. each edge or vertex creation is performed by a single database transaction. It was empirically observed the optimal number of transactions during graph creation is 500. Therefore, during the creation of graph in the Titan database, 500 vertices and edges are sent together to be created.

Besides creating the metadata graph structure in Titan database, Manta Flow also stores user's **source code files** on the disk. User's source code files are sent asynchronously to reduce the server load and during the creation of the graph structure in Titan, Manta checks whether these source code files are already received or not. See the following section 1.7 for more information about source code files and their storage in Manta Flow.

1.7 Source code files

Source code files are essential for visualization of data flows in Manta Flow. Typically, source code files are user's scripts determining data flows between objects. For instance, user uses Oracle database. Then, source code files are SQL scripts defining creation of tables, columns or views, insertions, selections, imports and other usual database operations and procedures. Manta Flow is able to visualize all these operations as data flows in user's database.

Source code files are not stored in the Titan database. Source code files are stored aside on the disk. Manta Flow stores in Titan only names of these files in the source nodes (property `sourceNodeId`) in the source node graph. However, source code files are for the purposes of data flows split into smaller parts and these parts are represented by nodes in the main graph. For referencing to the source code file and identification of source code files within Manta Flow is used the randomly generated sourceNodeId. SourceNodeId is assigned to the source code file when it is imported to Manta Flow.

1.7.1 Example

For example, in the figure 1.11 we can observe how a script is represented in the graph database. The whole script as a source code file is represented by source node under the source root. From this source node we can find out what substitute name of the source code file is used in the database (`60CD105CA0AB416`), in what folder at user's storage the script was stored before the import to Manta Flow (`C:\Manta\Procedures`) and what hash of this script was used during import to Manta Flow (`93416a03a0d749186`).

All scripts stored in Manta are located in one folder on a disk. If we knew location of this folder, then we would find there a file (script) with name `60CD105CA0AB416`.

This script (`60CD105CA0AB416`) is interpreted as a procedure. Therefore, a node of type `Procedure` was created in the main graph. This procedure is part of a package of procedures. Hence, a node of type `PLSQL Package` was also created in the main graph and this package was set as its parent.

Furthermore, the Procedure node has two attributes: `sourceEncoding` and `sourceLocation`. Thanks to these attributes, Manta Flow knows in what file (script) can be found content of this Procedure.

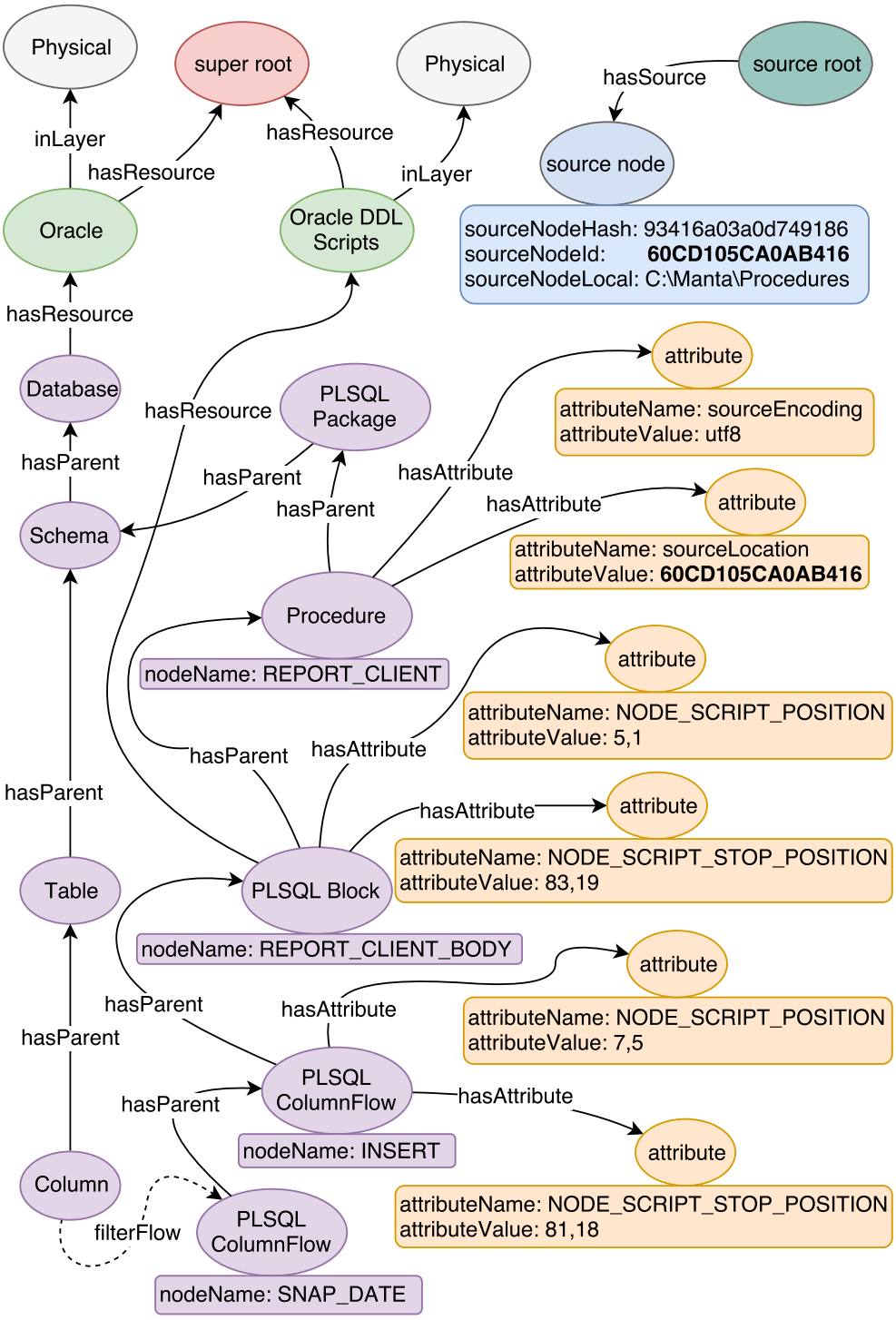


Figure 1.11: Example of the source code file management

In Manta Flow, every script is partitioned into smaller logical parts and these parts are ordered into a hierarchy. In this example, procedure consists of blocks of codes (PLSQL Block) and each block consists of single operations (PLSQL ColumnFlow). In Titan database is stored position in the file where these blocks and operations start and end. In this example, the insert operation starts at line 7, column 5 and ends at line 81, column 18.

Finally, by partitioning scripts into smaller logical parts, Manta is able to describe what operations are affecting what nodes, i.e. describe data flows. In this example, there is an insertion into column `SNAP_DATE`. This insertion is indirectly affected by the column `PARTY_KEY`. Therefore, there is created a filterFlow from `PARTY_KEY` column to `SNAP_DATE` column.

1.8 Interpolation

According to the Manta Flow documentation [15], interpolation is a process of creating new data flow edges in one layer derived from data flows in another layer. Moreover, this process can create not only data flow edges but also new nodes.

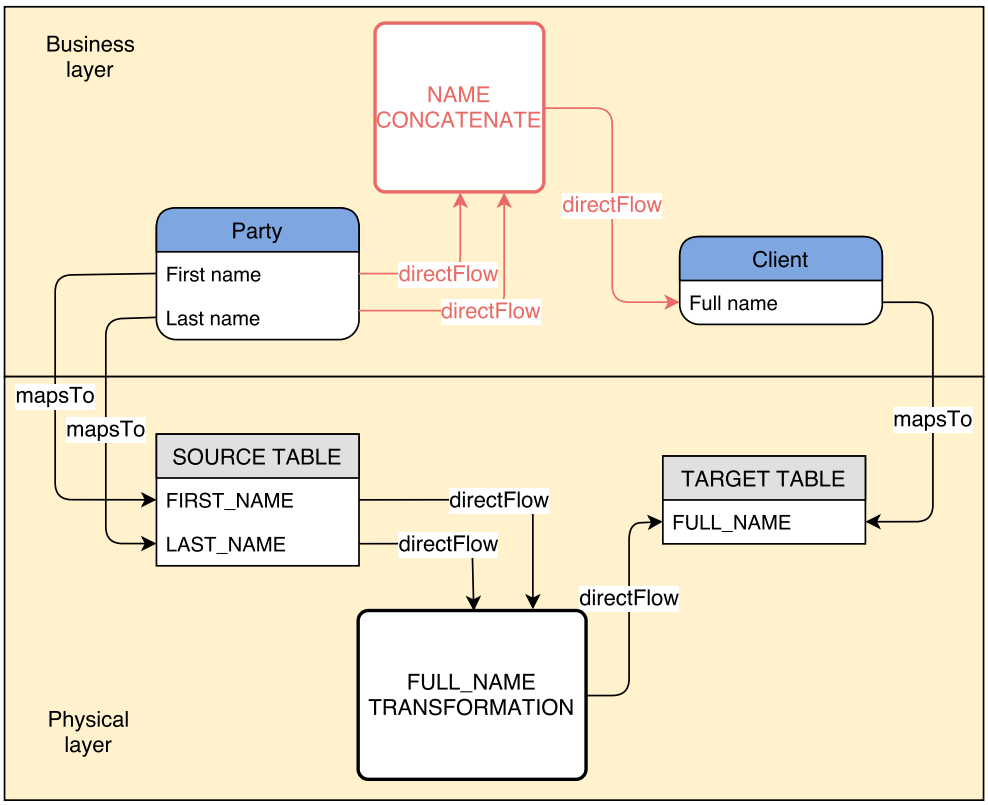


Figure 1.12: Example of interpolation

For example, in the figure 1.12 were before the start of the interpolation only two entities in the business layer: Party and Client. These entities were mapped to their counterparts in the physical layer: SOURCE TABLE and TARGET TABLE. The interpolation analyzed data flows in the physical layer and created new data flows in the business layer according to the data flows in the physical layer. The newly created data flows have the property `interpolated` set to `true`. Additionally, the interpolation process is able to create new nodes (in the example a process NAME CONCATENATE) in the interpolated layer to truly mirror the mapped relationships.

The purpose of interpolation is to completely mirror data flows between two layers of the identical metadata. Users do not want to analyze data flows on the lower levels (e.g. physical layer) since data in these layers are less readable than data on the higher levels (e.g. business layer). Therefore, interpolation allows to display data flows in another layer in order to easy analysis of metadata for users.

1.9 Indexing

Index is a data structure allowing a fast retrieval of elements by specified property key/value pair(s). In Titan, it is possible to index both vertex and edge property keys. Titan supports three types of indices: **standard index**, **external index** and **vertex-centric index**. In this section, we will learn about these index structures and then we will describe their usage in Manta Flow.

1.9.1 Standard index

Standard index can be used for indexing either vertices or edges by indexing one or more of their property keys.

Standard index is fast and available without any further configuration. It is implemented directly against the configured storage backend. However, standard index is limited only to the exact index matches. In other words, standard index can only retrieve vertices and edges by matching one of their properties exactly [16].

Standard index is also closely related to the **uniqueness of a property key**. A unique property key is a key whose property value is always uniquely associated with a vertex. In other words, it is impossible to set the same property value of a unique property key for two different vertices. However, declaring a unique property key *requires* a defined standard vertex index [17].

1.9.2 External index

External index is more flexible than standard index. It supports retrieving vertices and edges by bounding their geo-location, properties allowing search

by numeric range or matching tokens in full-text search [16]. As mentioned in the section 1.2.1, Titan supports an arbitrary number of external index backends.

However, unlike the standard index, the external index backends need to be configured in the graph configuration before they can be used.

Titan in the version 0.4.4 supports two indexing systems: **Lucene** and **Elasticsearch**. Lucene has a slightly extended feature set and performs better in small-scale applications compared to Elasticsearch. Nevertheless, unlike Elasticsearch, Lucene is limited to single-machine deployments [18]. Therefore, Lucene is currently in use in Manta Flow since Manta Flow runs only on a single-machine.

1.9.3 Vertex-centric index

Vertex-centric index is an index structure **specific to a vertex**, i.e. index for a quick retrieval of adjacent vertices of a specific vertex. The purpose of vertex-centric index is to sort and index the incident edges of a vertex according to incident edges' labels and properties [19]. Simply put, vertex-centric indexing sorts edges of a vertex by their properties making it a label as well. Sorting the edges may speed up lookup of next vertex from current vertex, since there may be no need to go through all incident edges.

Unlike graph indices[3], which are **global** to the graph (designed for fast global lookups), vertex-centric index enables fast graph traversals by avoiding linear scanning of incident edges at every visited vertex in a vertex query.

Vertex-centric indexing reduces time complexity of a adjacent vertex search from linear complexity $O(n)$, (n stands for the number of incident edges of a vertex) to at most a logarithmic complexity $O(\log n)$. Therefore, vertex-centric indexing is suitable, especially, for graphs with a large number of edges [19].

Vertex-centric indexing is possible thanks to the storage of the graph in the adjacency list format (see section 1.2.2. It allows to store the adjacency list of each vertex in a concrete sort order defined by sort-key configurations of the edge labels and properties.

1.9.4 Indices in Manta Flow

Following is the list of indices implemented in Manta Flow:

- Standard indices
  - superRoot
  - revisionRoot
  - sourceRoot
  - childName

[3] Both standard and external index are graph indices.

  - targetId
  - sourceLocalName
- External indices
  - nodeName
  - tranStart
  - tranEnd
- Vertex-centric indices
  - hasResource sorted by {childName, tranEnd, tranStart}
  - hasParent sorted by {childName, tranEnd, tranStart}
  - directFlow sorted by {targetId, tranEnd, tranStart}
  - filterFlow sorted by {targetId, tranEnd, tranStart}
  - hasSource sorted by {sourceLocalName, tranEnd, tranStart}

SuperRoot index is a standard *unique* index created on vertices using boolean vertex property superRoot. This index is created for fast retrieval of super root vertex.

RevisionRoot index is a standard *unique* index created on vertices using boolean vertex property revisionRoot. This index is created for fast retrieval of revision root vertex.

SourceRoot index is a standard *unique* index created on vertices using boolean vertex property sourceRoot. This index is created for fast retrieval of source root vertex.

ChildName index is a standard index created on edges using edge property childName. This index helps to faster look up a child node from its parent node by the child's name.

TargetId index is a standard index created on edges using edge property targetId. This index helps to faster retrieve a target node (end node) of the connecting edge.

SourceLocalName index is a standard index created on edges using edge property sourceLocalName. This index helps to faster retrieve source nodes with a specified name of the source code file.

NodeName index is an external, full-text search indexed created on vertices using String property nodeName. This external index is using external backend Lucene.

TranStart and **tranEnd** indices are external indices created on edges using edge properties tranStart and tranEnd. The external index was created for the purpose of interval search.

Vertex-centric indices are created as labels and are sorted by one special property key and tranEnd and tranStart property keys in this order. For instance, source root vertex has sorted all its outgoing edges first by property sourceLocalName, then by property tranEnd and finally by property tranStart. Therefore, finding the latest version of a source code is faster when using vertex-centric indexing, since the latest source node is reachable by the first outgoing edge.

1.10 Version control

Based upon analysis of Petr Holeček [1], a **concept of revisions**[4] (temporality) was implemented in Manta Flow to capture changes in time, i.e. version control.

The implementation of temporality allows users to store changes of objects in time. That means, we can track changes of user's metadata and data flows in Manta Flow. For instance, user can find out what were the data flows in his or her database in a specific revision or what was the impact of adding a new database script in another specific revision.

1.10.1 Revisions

Version control system is based on the term of **revisions**. Revision represents a point in time capturing an exact state of all objects in database.

Revisions are stored in the revision tree and each of them contain in their vertex properties basic information about a specific revision.

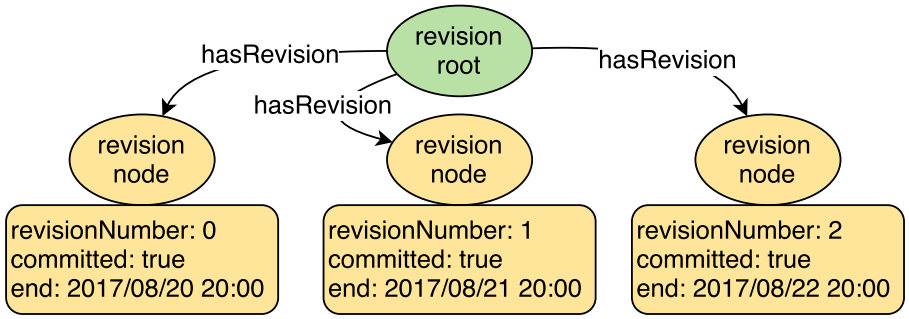


Figure 1.13: Example of revisions in revision tree

In the figure 1.13 we can see three revisions with their properties (see section 1.3.1 for description of the properties). Revision with number zero is also called a *technical revision*, all other revisions are non-technical. Technical revision is created at the beginning when no user's data is imported. Therefore, no data is tracked within this technical revision, i.e. technical revision has no usage for data tracking.

1.10.2 Revision tracking in the graph

Manta Flow supports version control. When version control is enabled, all objects in Manta Flow have specified their time validity by two points in time: **tranStart** and **tranEnd**. TranStart is number of the revision in which the object was created and tranEnd is number of the latest revision in which the object still existed.

[4] In the Manta Flow terminology terms "version" and "revision" are interchangeable.

According to the design of version tracking by Petr Holeček [1], revision validity of vertices is not stored in the vertices but in the **control edges**. Each vertex has defined a direction and label of its control edge. Super root, revision root and source root are the only exceptions. Root vertices have undefined revision validity because these vertices are always present. Following is the summary of control edges for each type of vertex used in Manta Flow:

Vertex	**Control edge**
super root	N/A
revision root	N/A
source root	N/A
resource	outcoming hasResource
layer	incoming inLayer
node	outcoming hasParent or outcoming hasResource
attribute	incoming hasAttribute
revision node	incoming hasRevision
source node	incoming hasSource

Table 1.1: Summary of vertices' control edges

Each vertex should have defined only one control edge. For instance every node has at most parent node (therefore, only one outcoming hasParent edge) or every attribute is connected with only one node (therefore, only one incoming hasAttribute edge).

We can track revision validity not only for vertices (e.g. nodes representing tables, columns, attributes etc.) but also for edges (e.g. data flow edges). Nevertheless, revision validity is always stored in the edges via the properties tranStart and tranEnd.

In the figure 1.14, we can see an example of revision tracking in the main graph. We suppose, there are created three non-technical revisions with revision numbers 1, 2 and 3. Hence, the latest (current) revision is revision number 3. All revisions are committed.

In this example, vertices resource, layer, node A, node B and node C were all created in the revision number 1 (tranStart=1) and are still present in the latest revision number 3 (tranEnd=3). As mentioned above, revision validity of vertices is defined by the properties tranStart and tranEnd on their control edges. For instance, node A has defined revision validity by its outcoming edge `hasResource` and node B and node C have defined revision validity by their outcoming edges `hasParent`.

Furthermore, attribute X was created in the revision number 1, however, it was removed in the revision number 3. Thus, its latest valid revision is revision number 2 (tranEnd=2). Similarly, directFlow between node B and node C was created in revision 1 (tranStart=1) and removed in revision 3

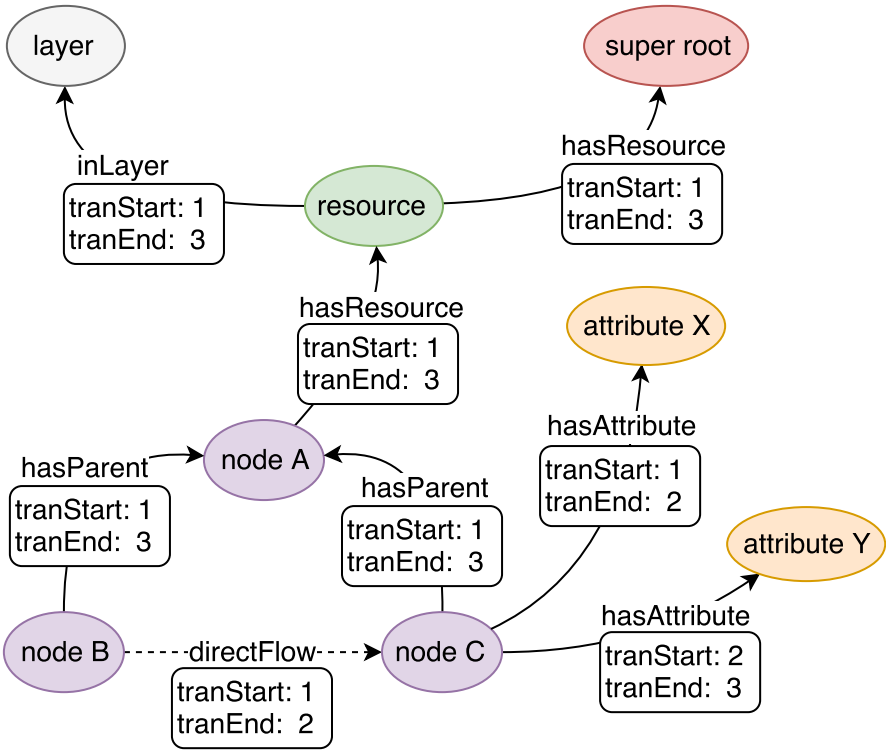


Figure 1.14: Example of revision tracking in the main graph

(tranEnd=2). Finally, attribute Y was created in revision 2 (tranStart=2) and is still present in the latest version (tranEnd=3).

1.10.3 Operations with revisions

Users are allowed to manipulate with revisions in order to track history of their data. Manta Flow currently supports these operations with revisions: **create a new revision**, **commit a revision**, **rollback a revision**, **prune to a revision** and **prune the oldest revisions**. Usage of these operations is identical to usage of any other version control system. We will shortly explain how these operations are implemented in Manta Flow.

Create new revision operation creates a new revision node in the revision graph and sets its revision number greater by one than the previous (highest, latest) revision number. The newly created created revision is uncommitted ($revisionCommitted \leftarrow false$). The newly created revision node is connected to the revision root by hasRevision edge.

Commit revision operation only sets appropriate revision node proper-

ties to new values:

$$revisionCommitted \leftarrow true$$
$$revisionEnd \leftarrow currentDate()$$

The only restriction is you can commit only an uncommitted revision.

Rollback revision operation removes all changes from an uncommitted revision and removes this revision node. Rollback can be performed only on the uncommitted revision. Rollback operation traverses through the whole graph and performs revision validation over all visited objects. Revision validation checks object's tranStart and tranEnd properties and performs appropriate operation:

- If the object was created in the last uncommitted revision, i.e.:

  $$tranStart = tranEnd = last\ uncommitted\ revision$$

  then the object is removed from the database.

- If the object was created in any previous committed revision and is valid in the last uncommitted revision, i.e.:

  $$tranStart < tranEnd \wedge tranEnd = last\ uncommitted\ revision$$

  then tranEnd is set to the previous value:

  $$tranEnd \leftarrow tranEnd - 1$$

- If the object was created in any previous committed revision but was also removed in some of the previous committed revisions, i.e.:

  $$tranStart \leq tranEnd < last\ uncommitted\ revision$$

  then do nothing.

Prune to revision operation removes old revisions up to a specified revision, including the specified revision. Removing a revision in this context means not only removing a revision node. Especially, it means physically removing from the database all vertices and edges valid only in the revisions to be pruned. In other words, if $tranEnd \leq latest\ pruned\ revision$, then the object is removed from the database. Otherwise, the object is left unaffected.

For instance, we have ten revisions 1, 2, 3, ... 9, 10. If you want to remove all old revisions up to revision 5 (including), you call `pruneToRevision(5)`. After this operation is performed, only revisions 6, 7, 8, 9 and 10 remain.

More specifically, if an object from the example above had $tranStart = 1$ and $tranEnd = 5$, it is removed. If another object had $tranStart = 1$ and

tranEnd = 6, it remained unaffected. Revision nodes 1, 2, 3, 4 and 5 were removed from the database as well.

Prune oldest revisions operation is similar to the prune operation described above. We only specify how many newer revisions should remain. For instance, we have ten revisions 1, 2, ... 9, 10. We want to have only 3 newest revisions. Then we call `pruneOldestRevisions(3)`. After this operation is performed, only revisions 8, 9 and 10 remain.

1.10.4 Merge

Except the above described operations with revisions, user is allowed to propagate his or her local changes, i.e. to save changes made within the last uncommitted revision. It is analogical process to the *commit*[5] command in SVN or *push* command in Git.

Merging process is same as the process of graph creation (see section 1.6). As well as in case of graph creation, user first creates a local linearized graph structure of his or her metadata. Then user merges this whole graph structure into the current graph stored in Titan. Thus, the only difference is, user does not merge into an empty graph (except root vertices and other supporting structures) but into the graph containing all the previous versions of user's metadata.

When merging an object, two cases may happen: object exists in the database or object does not exist in the database. Whether the object already exists in the database or not is checked by **equality criteria**. Equality criteria is defined for each object and may differ. For instance, two nodes are equal when their properties nodeName and nodeType are equal and they have the same parent vertex. Additionally, **revision validity** is considered for all objects as well. If the same object existed in the database but was removed, i.e. *tranEnd* < *latest committed revision*, it is also considered non-existing.

Hence, when a merged object does not exist in the database, i.e. the object was either removed before or was never created, it is created: the object itself is created in the database, in case a vertex is created an edge or edges are created as well, and object's revision properties are set to these values:

$$tranStart \leftarrow new\ revision\ number$$
$$tranEnd \leftarrow new\ revision\ number$$

When the merged object already exists in the database and was not removed, i.e. *tranEnd* = *last committed revision*, we only update the current object's tranEnd property:

$$tranEnd \leftarrow new\ revision\ number$$

[5]Commit in SVN is different from commit in Manta. Commit in SVN checks local changes into the repository while commit in Manta only sets property revisionCommitted to true in a revision node without checking data into the repository

1.11 Analysis of the current implementation

A major ineffectiveness occurs in the current implementation. When a change is made on the user's side and user wants to see the impact of this change (i.e. user wants to visualize a data flow diagram based on the new data), these changes have to be updated in the Manta Flow graph database. However, in the current implementation, it has the effect of traversing through the whole graph to update all objects' revision validity.

Let us show it on a simple **example**:

1. User changes an attribute in one column. For instance a column name was changed
2. User wants to save this change in Manta Flow
3. The latest revision 3 is committed. Thus, a new revision 4 has to be created first
4. A linearized local graph structure of user's new metadata is created and sent in the form of CSV file to Manta Flow server
5. The linearized graph structure is merged into the current graph structure in Titan database
6. User commits the newly created revision 4

In this example (see figure 1.15), user disliked American English and wanted to rename column with name *Color* to British English version *Colour*. Therefore, only one vertex property value was changed in the whole graph (property nodeName in the column 2) in the new revision 4.

Thanks to version control in Manta Flow, column 2 with the old name *Color* remains (indicated by control edge hasParent with $tranStart = 1$ and $tranEnd = 3$) and the changed name of column 2 is represented by the newly created vertex (indicated by control edge hasParent with $tranStart = 4$ and $tranEnd = 4$). For instance, if we wanted to know what name of column 2 was in the revision 3, we would get the old vertex with nodeName *Color* and if we wanted to know name of column 2 in the revision 4, we would get the new vertex with nodeName *Colour*.

However, to keep all information up-to-date, we had to update other tranEnd properties for all other objects. If we did not update their tranEnd property from 3 to 4, it would look like all of them were removed in the new revision 4. This update was performed by the merge process (see section 1.10.4 for detailed information about merging).

To summarize it, a whole graph structure had to be created and merged into the current graph structure in Titan to update only one vertex property. We can call this process a **full update**. In general, full update refers to

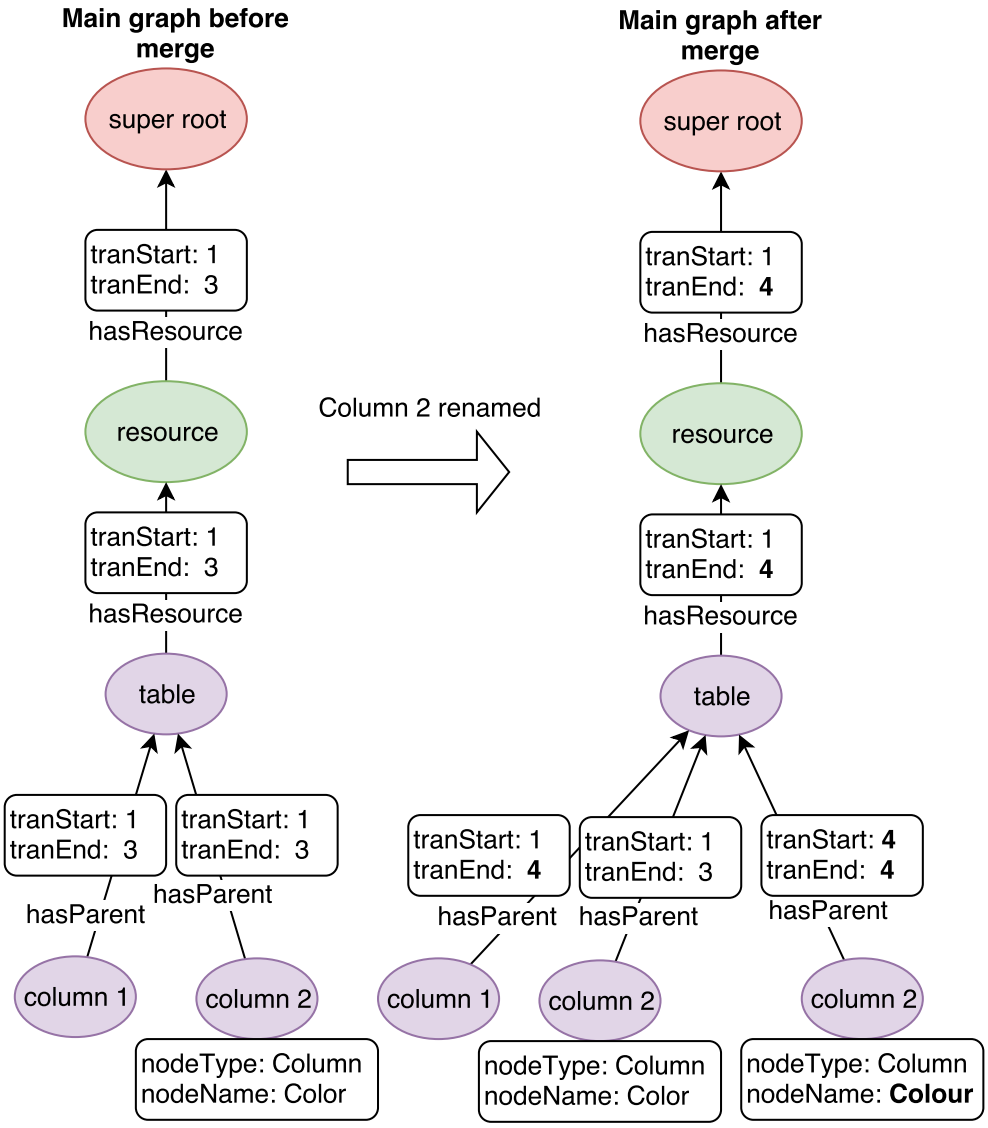


Figure 1.15: Example of merged changed graph

a load of **all data** into a data storage, including data not changed since the last data load. Contrary to the full update, **incremental update** refers to a data load into a data storage **based on changes** made since the last data load.

The advantage of incremental update is reduction of the amount of data being transferred and saved in the data storage. Consequence of reducing the amount of data is **reduction of time complexity** for performed updates. This advantage may not be obvious for small amount of data. However, it may become crucial when managing a large amount of data, where the time

complexity between full update and incremental update may differ by orders of magnitude.

Currently, in Manta Flow is implemented only full update and our task is to implement the incremental update in order to speed up updating data in the graph database Titan.

1.12 Summary

In the first chapter, we learned about the Manta Flow project, especially, about the persistent data flow storage module. This chapter not only describes how Manta Flow works but also unifies terms used in the current implementation and documentation and gives a detailed summary of Manta Flow data lineage storage. Therefore, this chapter can be used as well as a comprehensive documentation of the persistent data flow storage module in Manta Flow.

The last part of this chapter was dedicated to the analysis of a current implementation of version control in Manta Flow. We found out, updating changes in the graph database Titan are very ineffective since every update is performed as a **full update**, i.e. even a small change update (e.g. adding one vertex or edge) had the effect of traversal through the entire graph structure. This process can take up to hours or even more, depending on the data size. Hence, we introduced the idea of implementing **incremental update** to reduce the time complexity of the update. We will study different methods of incremental update in the following chapters and based on these studies, we will design an implementation of incremental update suitable for Manta Flow.

Chapter 2

Related work and inspiration

In this chapter, we will study the **version control systems** and **incremental backups** in databases. We will focus on how these systems perform updating their data. Purpose of this analysis is to find an inspiration for a design of incremental update in Manta Flow.

There is a plenty of version control systems. Therefore, we have chosen two entirely architecture different version control systems: **Subversion** (centralized VCS) and **Mercurial** (distributed VCS). Primarily, we will analyze the way these VCSs store metadata and how they perform updates from revision to revision. We will not study the way these version control systems deal with simultaneous collaboration of multiple users (such as branching, conflicts solving, shared files locking, transactions, etc.).

2.1 Subversion

Subversion is an open-source centralized version control system (CVCS). Subversion contains two essential objects that users interact with: **working copy** and **repository**. Working copy is an ordinary directory tree on the user's local system containing some project's files. Repository is a shared storage of all users working on the same project. Repository contains the complete history of the project. Hence, it contains all changes made locally in the working copies that were afterwards published into this repository [20].

Subversion uses **revisions** for tracking the data history. Revision represents one snapshot of the versioned data in a specific time. Each revision is assigned a unique natural number, one greater than the number assigned to the previous revision [21]. The same approach is used in Manta Flow (see section 1.10.1).

The **basic work cycle** in Subversion can be divided into these steps:

1. Update local working copy according to the repository (update command)

2. Make changes in the local working copy

3. Publish changes from the local working copy to the repository (commit command)

This work cycle is similar to all version control systems. User first updates his or her working copy to be up-to-date with the latest revision in the repository. Then, user makes some local changes in his or her working copy. When user is finished with changes, he or she publishes (commits) these changes to the repository.

Except these basic steps, users have to deal also with changes of other users. Therefore, sometimes user has to merge changes (merge command) and resolve potential conflicts, even before committing the changes back to the repository, since user's working copy has to be up-to-date with the latest revision in the repository before publishing (committing) changes to the repository. Subversion will not let you commit anything new until you deal with any conflicts [22]. The basic operations in Subversion and their usage in the Subversion schema is described in the figure 2.1.

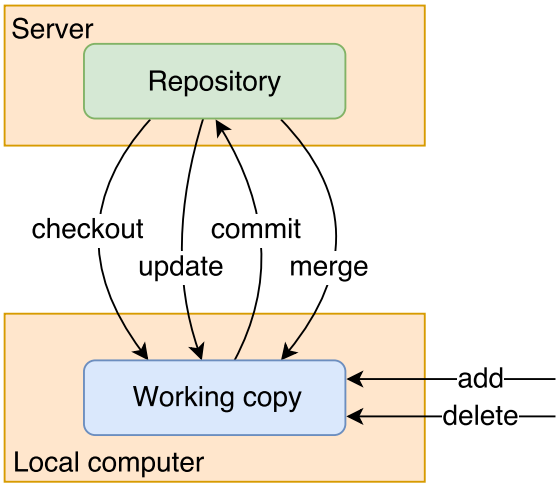


Figure 2.1: Subversion basic operations

Now, when we have a basic knowledge how Subversion works, we will study how the data is stored in both repository and working copy and we will learn how both repository and working copy are designed internally.

2.1.1 Working copy

Working copy is a local copy of the versioned data in one specific state. It is an ordinary directory tree on the local system containing a specific snapshot of the versioned data.

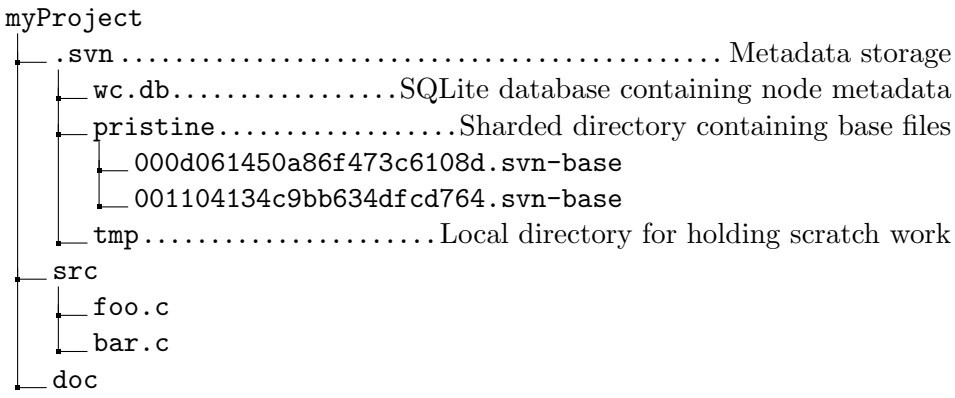


Figure 2.2: Example of a working copy

In the example of a working copy directory tree (see figure 2.2), we can see a typical project (myProject) with its content (directories src and doc, source code files foo.c and bar.c).

Every working copy contains an administrative directory **.svn**. This directory is responsible for tracking the changes made in the working copy.

Prior to version 1.7, the .svn directory used to be in every subdirectory. The idea behind this model was to keep track of directories' own revision information. It was thought to be advantageous to the basic idea of Subversion, where every file and directory can be versioned, thus enabling more effective operations on the level of single files or subdirectories [23]. One disadvantage of this old approach was that when a directory was deleted in the working copy, files inside the directory were deleted. However, the empty directory remained until this change was committed. This behavior was necessary since Subversion held the information about the deleted directory right inside this directory in its .svn directory.

The **wc.db** is a self-contained SQLite-based database containing all Subversion metadata. Every versioned item can be described by one or more **nodes**. There are four types of nodes: **BASE**, **WORKING**, **NODE_DATA** and **ACTUAL**. These nodes are saved in the SQLite database under a complex schema [24] [25].

BASE node represents the original item checked out from the repository and is used for offline comparison of the changed data with the original content without the need of connecting to the repository. BASE node corresponds to a particular repository URL and revision.

WORKING node describes structural changes to BASE node. For instance, when a new item is added to the working copy (e.g. created a new file), a WORKING node is created. However, this node does not describe changes in the content of this item.

NODE_DATA node describes layered structural changes to WORKING

node. For instance, directory A is replaced, another directory B is copied into this replaced directory A and a file is added to the directory A. Then the WORKING node describes replacement of directory A, one NODE_DATA describes the copied directory B into the directory A and second NODE_DATA describes the added file into the directory A.

ACTUAL node describes property changes and text changes of a modified file content.

The **pristine** directory contains all the base files (represented by the BASE nodes). These files are stored under the name of the hash of their content. In this example (figure 2.2, the two C files have their original content from the repository stored in the pristine directory. These base files have *.svn-base* extension and contain the original text.

The .svn directory may contain also another support files. However, the most important is the database file wc.db containing all the metadata and the original content files in the pristine directory.

Unfortunately, there is missing a proper documentation how Subversion performs data changes in the working copy and the source code structure is too messy[6] to grasp the key principles of data updates without a deep code analysis. However, we may use, at least, the information it is possible to version not only files but also directories. This versioning may be performed by one centralized database or the metadata can be saved in each directory using flat files.

2.1.2 Repository

Repository is a central store of all the versioned data on the server side. Repository consists of two main parts: **Subversion file system** and **repository logic** wrapping the implemented file system.

Subversion provides two options for the underlying data storage each repository uses: **Berkeley DB** and **FSFS**. Berkeley DB is a database environment initially used in the first versions. FSFS is a versioned filesystem using the native OS file system directly. Both of these data stores have their advantages and disadvantages. For instance, FSFS is platform independent, has better scalability, has smaller disk usage and is insensitive to interruptions. On the other hand, Berkeley DB is extremely reliable since it supports backups and auto-recovery [27].

Regardless of the data storage used, we will now describe the **repository structure**. When the repository structure is defined, we will explain the **bubble-up method** which performs updates in the revisions file tree.

[6] As Greg Stein, one of the founding developers of Subversion, mentions [26], the working copy library (*libsvn_wc*) had grown over years into an unorganized bunch of code since there was no proper design. Although, there were made fundamental changes in the design in the version 1.7, no proper, publicly accessible documentation was created yet.

2.1.2.1 Repository structure

Repository structure is basically only a list of directory trees (one tree for one revision) and a table containing links to all versions of files and directories stored in these directory trees. Following is the list of objects defined[7] in the repository [23]:

- text string
- string of bytes
- property
- property list
- node
- node number
- node table
- file
- directory entry
- directory
- revision
- history

The lowest data object used in all of the other objects is a **text string** (a string of Unicode characters) and a **string of bytes**.

Property is a pair *(name, value)*, where name is a text string and value is a string of bytes. Property typically refers to some file or directory attributes. For instance, files may have executable permissions or EOL style defined.

Property list is an unordered list of properties. No two properties in an unordered list have the same name. In other words, property name is a unique key within a property list. Each revision, file, directory and directory entry has its own property list. Property lists are versioned as well. Therefore, Subversion can detect changes not only in the file content or directory structure but also changes in file or directory attributes.

Node is either a file or a directory. Every node has assigned a node number. **Node numbers** are stored in the **node table** and are mapped onto single nodes in this table. In other words, all files and directories are represented by a node and are accessible by their node number from the node table.

File consists of a property list (file attributes) and a string of bytes (content). **Directory** is an unordered list of directory entries and a property list. **Directory entry** is a triple *name, property list, node number*. Hence, content

[7] This structural definition is taken from the initial version of Subversion, since there is no newer documentation. Although, the structure could have changed, the substantial part of this structure probably remained. Moreover, the bubble-up method, which is supposed to be still up-to-date according to the developers' notes, is also explained by using this inital structure definition.

of directory (files and subdirectories) is represented by an unordered list of directory entries.

History is an array of revisions, indexed by a contiguous range of non-negative integers containing 0. **Revision** is a node number and a property list. Every revision has a *root directory*. The revision's node number is therefore node number of the root directory represented by a node. Revision 0 always contains an empty root directory.

To sum it up, repository consists of history and node table. History is an unordered list of revisions. Each revision refers to a root directory of a directory tree representing a snapshot of files and directories in the specific revision. Node table is a mapping of node numbers to all nodes (i.e. mapping to all versions of files and directories) in the repository.

Moreover, directories can have more than one parent. Hence, the term *directory tree* is incorrect and we should use term *directed acyclic graph* instead. However, we will stick with the term directory tree, since it would be distracting and unhelpful to replace this familiar term used everywhere in the documentation.

2.1.2.2 Bubble-up method

Bubble-up method is a method used for adding new revisions (directory trees) into the repository. The key idea of this method is to create only the nodes affected by the change and link the newly created nodes with the unchanged nodes. Let us show it on some examples from the Subversion developer notes [23].

In the following examples, all boxes represent nodes. Blue boxes represent directory nodes, green boxes represent file nodes. File nodes have a byte string content, while directory nodes have a list of directory entries as a content. Line represents relationship between a child node and a parent node. Node's name is stored only in its parent because a node with multiple parents may have different names in different parents.

Assume, we have only one revision of the project (one commit) in the repository (see figure 2.3). Hence, revision 1 refers to the root directory. This root directory contains two directories A and B. Directory A contains a directory fish and directory fish contains a file tuna. Directory B is empty.

In our **first example**, we want to modify file tuna and commit this change into the repository (second commit). This triggers a series of steps (see figure 2.4):

1. A new file node is created. This node contains the modified content of file tuna. This node is not connected to anything at the moment.

2. A new revision of parent directory (fish) is created and the previously created file node for tuna is connected to new parent directory (fish).

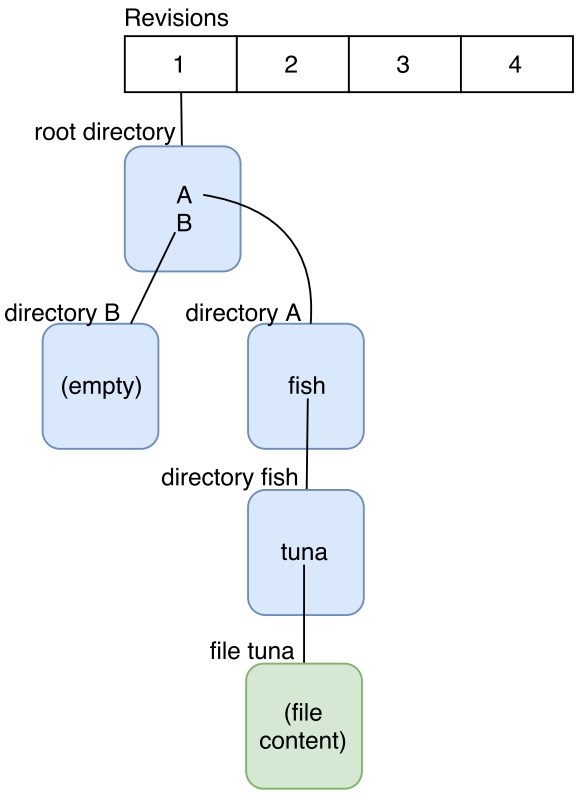


Figure 2.3: Initial state of repository

3. A new revision of parent directory (A) is created and the previously created directory fish is connected to the new parent directory (A).

4. Finally, we reached the root directory. Hence, a new revision of root directory is created and the directory A is connected to this root directory. However, directory B wasn't changed at all. Therefore, directory B from revision 1 is linked also to the new root directory.

5. Finally, all new nodes are created and connected. Thus, we link this new directory tree to the revision 2 and the process is finished.

As we can see from the first example (figure 2.4), it is **easy to locate revision of a specific file or directory**. For instance, to locate revision 2 of file tuna, we simply first find the revision 2 in the list of revisions (repository history), locate its root directory and walk down to the file tuna.

Another advantage of this approach is **writers do not interfere with readers**. When writer is trying to commit a new revision, first, he or she is creating new nodes aside and the concurrent readers cannot see the work in progress. The newly created tree becomes visible to readers after the writer

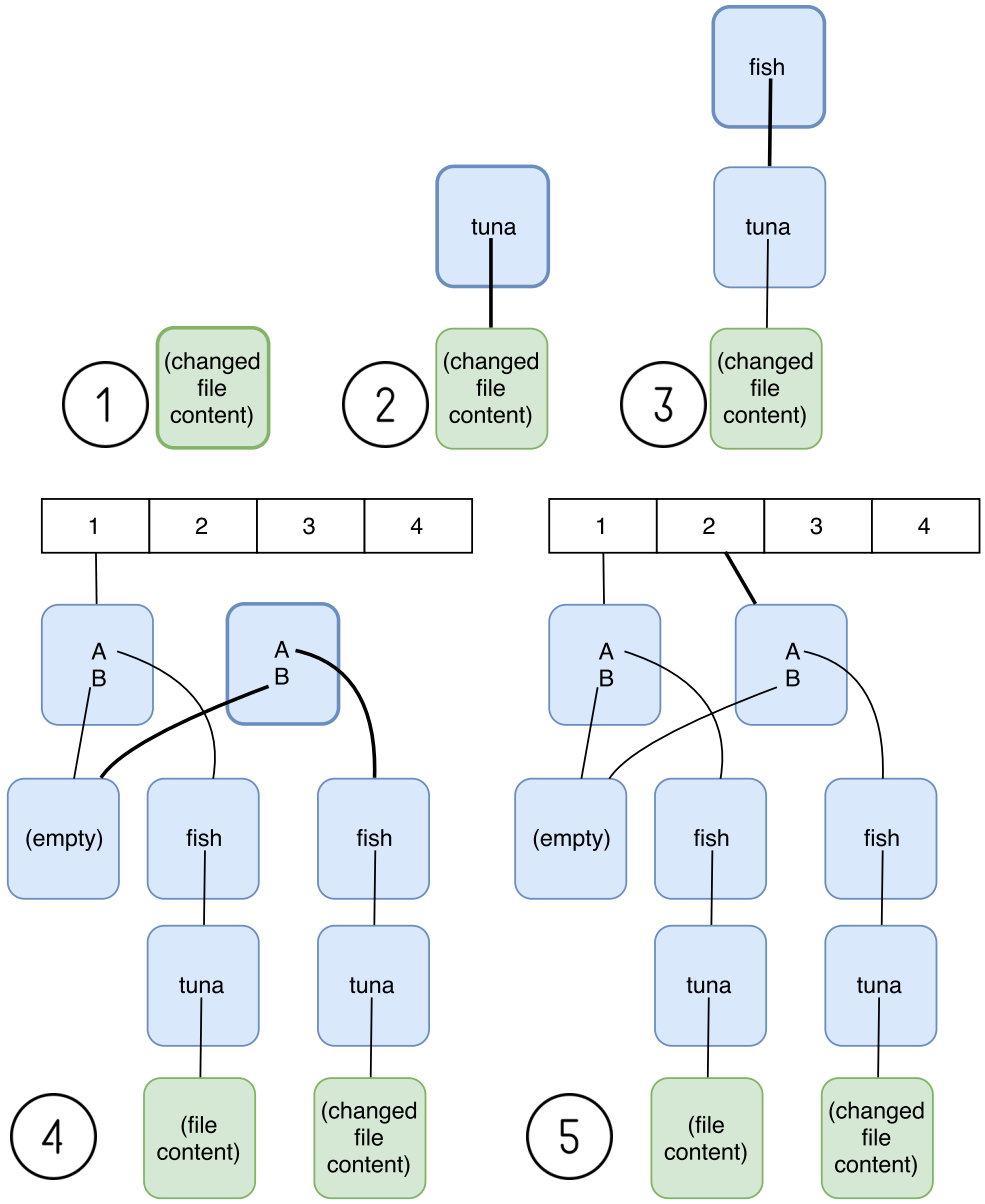


Figure 2.4: Bubble-up method - example 1

makes its final link from the root directory to the specific revision in the repository history.

In the **second example** (figure 2.5), we will demonstrate a structural change. What happens if we rename file *tuna* to *book* and commit it into the repository? We will bubble-up again from the changed node up to the root directory following these steps:

1. First, we create a new revision of directory fish. In the newly created directory node we create a new directory entry *book* and we connect this directory entry with the file node from previous revision. Thus, we will not create a new file node, since content of the file has not changed.

2. We create a new revision of directory A and connect it with a new revision of directory fish

3. We create a new revision of root directory and connect its directory entry A with new revision of directory A. The directory entry B is connected to the revision of directory B from the revision 1 because it has not changed since then.

4. We link the new revision of directory root to the revision 3.

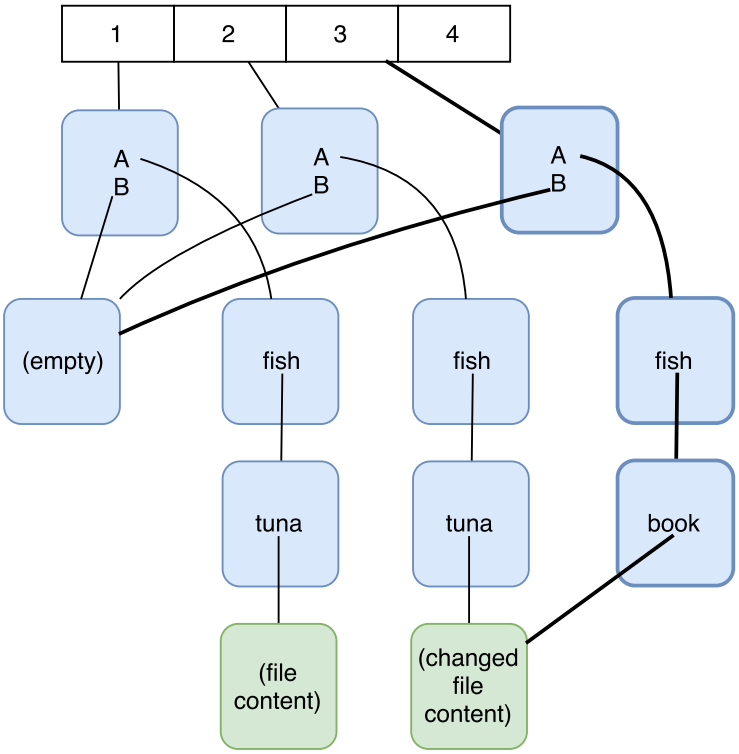


Figure 2.5: Bubble-up method - example 2

In the second example (figure 2.5), we can see why name of the node is not saved in the node itself but only in its parent nodes. This way, it is possible to track different names (tuna/book) of the same file.

The second example also demonstrates that Subversion is able to **version the file and directory structure**. All file and directory modifications (rename, move, add, delete) are part of the repository history.

As mentioned in the Subversion notes [23], it would be space wasteful to create an entire line of new nodes, including their content, in every commit. Hence, **Subversion stores changes as differences**. Subversion does not create entire copies of nodes. Instead, Subversion stores the latest revision as a full text and previous revisions are stored as a succession of reverse diffs. These diffs are called **deltas** and Subversion defines three kinds of deltas: **tree delta**, **text delta** and **property delta**. Thus, when a file or directory from an older revision is required, the appropriate deltas are applied on the latest node version and the previous node version is returned.

2.2 Mercurial

Mercurial is a distributed version control system (DVCS). Hence, Mercurial has no concept of a central repository. All users have their own local repository and share their changes via a remote repository. Technically speaking, remote repository and user's local repository are equal and users are free to define their own topology for sharing.

User's repository consists of two logical parts: **working directory** and **local repository**. In the working directory are stored user's data such as project files and directories, i.e. working directory contains data users manipulate directly with. In the local repository is stored the complete history of these tracked data from the working directory [28].

Repository vs. local repository

In Mercurial, the term repository may have different meanings, depending on the context. First, the term repository may be used for all the data and metadata on a user's computer, i.e. a directory containing both working directory and local repository. Second, strictly speaking, only local repository (metadata files stored in the .hg directory) is the real repository, since there is stored the complete history.
We will not bind the term repository neither to all files and directories in the user's local computer (i.e. local repository + working directory) nor only to the local repository (only metadata files). It will be always clear from the context or we will explicitly specify what we mean.

The **basic work flow** in Mercurial could be described by these five steps:

1. Check for changes in the remote repository using **pull** command
2. Update working directory using **update** command
3. Make some local changes in the working directory.
4. Commit changes to the local repository using **commit** command

5. Push changes from local repository to the remote repository using **push** command

We can notice, the basic work flow is similar to the work flow in Subversion. The only difference is, in Mercurial it takes two steps instead of one to propagate changes to or from the remote repository. For instance, in Subversion command *update* updates local working copy. In Mercurial, this functionality has to be performed by the combination of two commands *pull* and *update* to first propagate changes to the local repository and then to the working directory. Similarly, when propagating local changes to the repository, in Subversion, you only need command *commit* while in Mercurial, *commit* is a purely local operation whose effects are not shared with anyone else until a *push* command is performed to distribute these changes [29].

Except these operations, Mercurial provides another operations such as **clone** command for cloning the remote repository to the local repository, **merge** command for merging two branches into the local repository, **add** command for adding unversioned files in the working directory to the version control, **forget** command for removing versioned files from version control and many others. The overview of the basic operations is visualized in the figure 2.6.

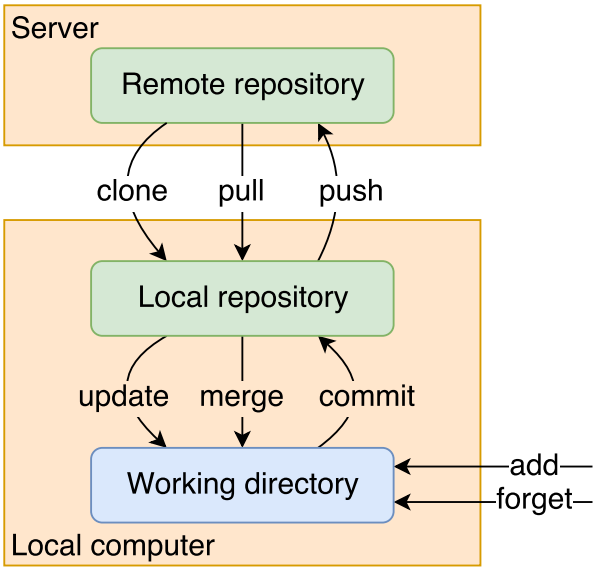


Figure 2.6: Basic operations in Mercurial

Now, when we grasped the basic concept how Mercurial works, we will study in detail, how the version control is implemented internally.

2.2.1 Metadata structure and relationship

Mercurial defines three metadata objects for tracking the history: **filelog**, **manifest** and **changelog**. These files are stored in the local repository.

Filelog stores history of a single file. Each entry in the filelog contains enough information to reconstruct one revision of a tracked file. Filelog contains two kinds of information: revision data and an index to help Mercurial find a revision efficiently. Hence, every tracked file in the working directory has its history tracked in a filelog [30].

Manifest collects together information about files it tracks, i.e. manifest describes content of the repository at a particular changeset[8]. Primarily, manifest contains a list of file names and revisions of these files [30].

Changelog contains information about all changesets. Each revision records who committed a change (committer), why was the change committed (changeset comment), when was the change committed (commit date and time), some other pieces of changeset-related information and finally, what revision of manifest to use [30].

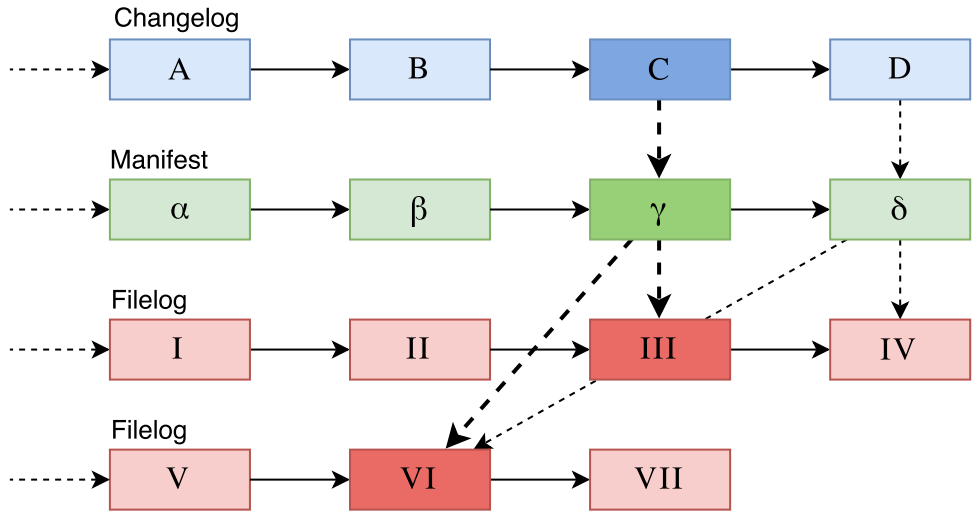


Figure 2.7: Metadata relationships

In the figure 2.7, we can see the visualized relationship between these metadata obejcts. Names of the boxes in this example are used only for distinguishing different revisions of the same object. Therefore, in this example is only one changelog file in four revisions (A, B, C and D) or there is one manifest represented by its four revisions (α, β, γ and δ).

Within a changelog, a manifest or a filelog, each revision stores a pointer to its immediate parent (or to its two parents in case of two merged revisions into one revision) [30]. This relationship is represented by the horizontal arrows.

[8] Changeset is a collection of changes to files in a repository.

For instance, a changelog's revision B has its previous revision A, or a filelog in revision II has its previous revision I.

For every changeset in the repository, there is exactly one revision stored in the changelog. Each revision of the changelog contains a pointer to a single revision of the manifest. A revision of the manifest stores a pointer to a single revision of each filelog tracked when the changeset was created [30]. Relationship between changelog-manifest and manifest-filelog is represented by the vertical dashed arrows. For example, changelog in revision C stores pointer to the manifest in revision γ and manifest in revision γ points to two filelogs, one in revision III and second in revision VI.

Notice, there is not a one-to-one relationship between changelog-manifest and manifest-filelog. For instance, manifest revisions γ and δ both point to the same revision VI of the second filelog. This means, a file has not changed between the two revisions, hence, the entry for that file in the two revisions of the manifest will point to the same revision of its filelog [30].

2.2.2 Revlog

All metadata objects in Mercurial (changelog, manifest and filelog) are using the same inner structure called **revlog**. Revlog represents all revisions of a specific file in the repository. Each revision of a file is either stored compressed in its entirety or it is stored as a compressed binary delta (difference) relative to the preceding revision [31].

More detailed, every metadata object consists of two parts[9] an **index file** and a **data file**. Index file contains a list of fixed-size records for each version of the metadata object and each record in the index file determines where exactly can be found the specific revision of the meatadata object in the data file. Hence, a data file contains all revisions of one file (metadata object) in the form of compressed deltas and compressed contents.

An example of the revlog layout is in the figure 2.8. In this example, Mercurial registers five revisions of a file in the index file and these revisions are stored in the data file. To reconstruct a specific revision, you must first read the closest snapshot and then apply all the left deltas up to the target revision. For instance in the figure 2.8, if you want to retrieve revision 5 of the tracked file, you need to read the snapshot from revision 4 and then apply the delta from revision 4 to revision 5.

The decision whether to store a full version or a delta is decided by how much data would be needed to reconstruct the file. That means, once the cumulative amount of deltas stored since the last snapshot exceeds a fixed threshold, a new snapshot is created instead of a delta. This system allows Mercurial to effectively store files since it does not require huge amounts of data to reconstruct any version of file [31] [30].

[9] When a file is small, the index file and the data file are combined into one file.

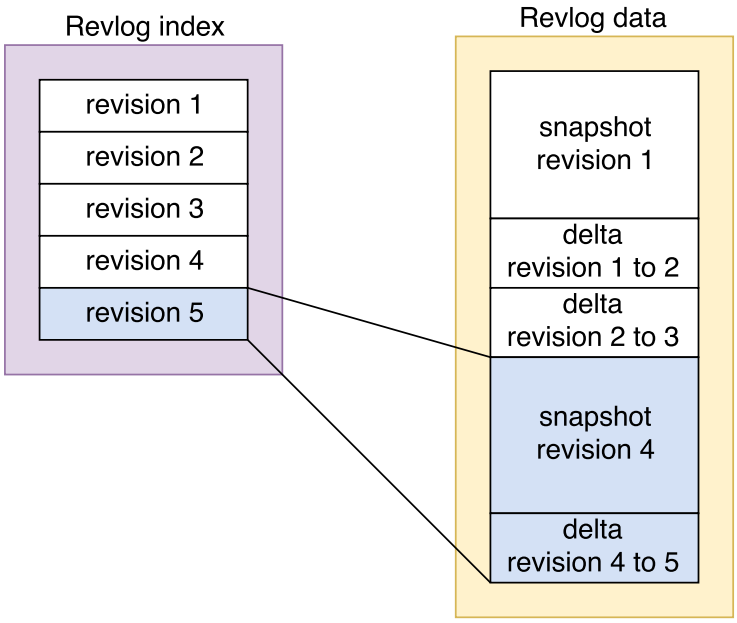


Figure 2.8: Revlog layout

Mercurial defines a **format of a revlog index record** in the following way[10]:

- offset (4 bytes)
- compressed length (4 bytes)
- base revision (4 bytes)
- link revision (4 bytes)
- nodeid (20 bytes)
- first parent nodeid (20 bytes)
- second parent nodeid (20 bytes)

Offset specifies, where to begin read in the data file. **Compressed length** defines, how much data we will read from the offset position to retrieve the delta or snapshot. **Base revision** specifies the last revision where the entire file is stored. **Link revision** points to the corresponding changeset. **First parent nodeid** is nodeid of the first parent, **second parent nodeid** is nodeid of the second parent. Finally, **nodeid** is the nodeid of the revision itself [31].

Nodeid is a unique identification of a revision. Nodeid is computed by the SHA-1 function which generates 160 bits (20 bytes). This hash is computed using content of the file and its position in the project history. Position

[10]Mercurial introduced a new format *RevlogNG*. This new format is eliminating some deficiencies in the original Revlog format (see [32] for more details). However, we will describe the original Revlog format since the same logic remains and it is easier to understand.

in the project history means using the *first parent nodeid* and the *second parent nodeid*, since parent revisions define the position of the revision in the project history. Thanks to this approach, when a file is modified, the change is committed, and then modified back again to restore the original content and committed, the content of the file is same but the parent revisions are different. Hence, a file with the same content will get a new nodeid [33].

In case of the first revision, both parent nodeids are empty (all zeros, also called a *nullid*), i.e. there is no previous revision. In case of revision created by merge of two previous revisions, both parent nodeids are non-empty, i.e. revision has two parents. Otherwise, only first parent is non-empty and the second parent is nullid, i.e. revision has only one parent.

rev	offset	length	base	linkrev	nodeid	parent 1	parent 2
0	0	467	0	10	a7bdd2379	000000000	000000000
1	467	168	0	12	692932a95	a7bdd2379	000000000
2	635	173	0	15	f1d9cb420	692932a95	000000000
3	808	476	0	17	d238a6113	f1d9cb420	000000000
4	1284	491	0	18	b71d29927	f1d9cb420	000000000
5	1775	470	0	19	4a7ebb32f	b71d29927	d238a6113
6	2245	64	0	20	6b99ca4dd	4a7ebb32f	000000000
7	2309	177	0	21	33557d969	d238a6113	000000000
8	2486	213	0	22	e4d67566a	6b99ca4dd	33557d969
9	2699	102	0	23	ab4bcfb96	33557d969	000000000
10	2801	384	0	24	86d19e47e	e4d67566a	000000000
11	3185	88	0	25	4969c00e0	86d19e47e	ab4bcfb96

Table 2.1: Sample content of an index file in the revlog format

In the table[11] 2.1, you can see a sample content of an index file in the revlog format. In this example, when the file was created in its revision 0, its offset in the data file is 0 and length is 467. Therefore, if you want to retrieve revision 0 of the file, you would only need to go to the beginning of the data file and read 467 bytes. Also notice, when at the revision 0, file has no parents, because it was newly added to the repository, and thus, it has no previous history at this point.

Moreover, in the example above, all revisions have a base revision 0. That means, only one snapshot was created in the revision 0 and all other revisions are deltas. Also notice, link revision points to the changesets with higher numbers. The implication is file was not changed during each commit. For instance, the first revision of file was added to the repository within the tenth changeset (commit).

[11] A keen reader will surely notice nodeids (SHA-1 160-bit hashes) in the table are significantly shorter. These hashes were shortened only for the aesthetic purposes to fit the width of the table within the page.

Finally, we can observe in the table 2.1 usage of branching and merging. In the revision 4, file has the same parent as in the revision 3. It is rather obvious, a branch was created in the revision 4. Afterwards, in the revision 5, file has two parents from revision 3 and revision 4, meaning it was merged back after the previous one-night branch.

Now, when we understand in detail the Mercurial metadata objects, we can put it all together and see how it works. In the figure 2.9, we can see how all the metadata objects are linked together using the revlog structure.

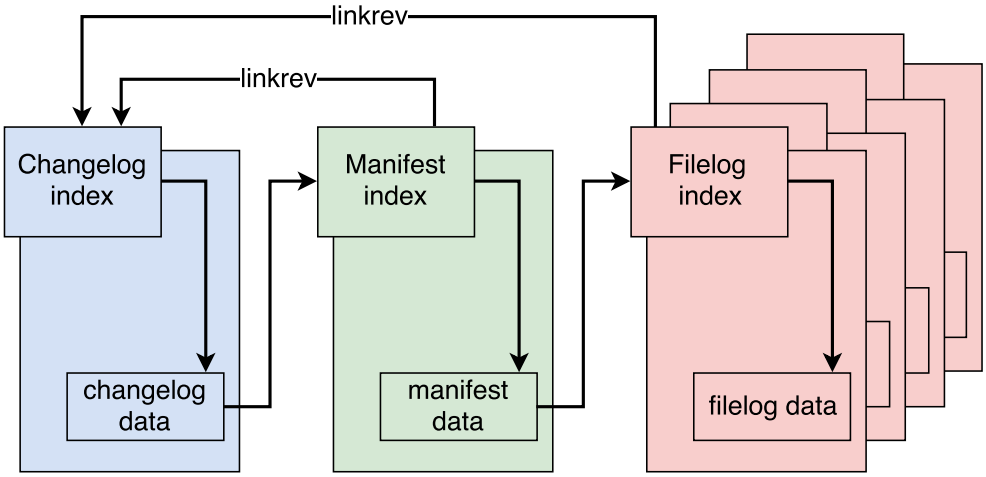


Figure 2.9: Metadata hierarchy using the revlog structure

If you want to look up a given revision of file, you need to proceed through these steps [34]:

1. Look up the changeset in the changelog index
2. Reconstruct the changeset data from the data file
3. Look up the manifest nodeid from the changeset in the manifest index
4. Reconstruct the manifest for that revision from the data file
5. Find the nodeid for the file in that revision
6. Look up the revision of that file in the filelog index
7. Reconstruct the file revision from the data file

On the other hand, if we want to find a changeset associated with a given file revision or a manifest, we simply follow the linkrev.

2.2.3 Committing changes

How does Mercurial saves changes from the working directory to the repository? In other words, what happens, when you want to **commit changes** from the working directory to the local repository? Short answer: committing changes to the repository involves updating all three types of the metadata objects: all modified files (their filelogs), the manifest and the changelog.

Changeset commit is a **two-stage process**. The first stage walks from top to bottom, from the changelog, to the manifest and to the filelogs. The second stage goes back up from filelogs, to the manifest and finally ends at the changelog [35]. Let us examine it in detail.

2.2.3.1 First stage - top to bottom

Purpose of the first stage is to retrieve the parent revision of the repository. Parent revision is a specific revision checked out (updated) from the local repository to the working directory. Parent revision is the initial state of the working directory, i.e. a state before no file was modified in the working directory.

Changes in the working directory are tracked in the file *dirstate*. This file is stored in the local repository. In the figure 2.10, you can see a working directory that was checked out from the revision 2 (initial state) and some changes were performed. Revision 2 is the parent revision then.

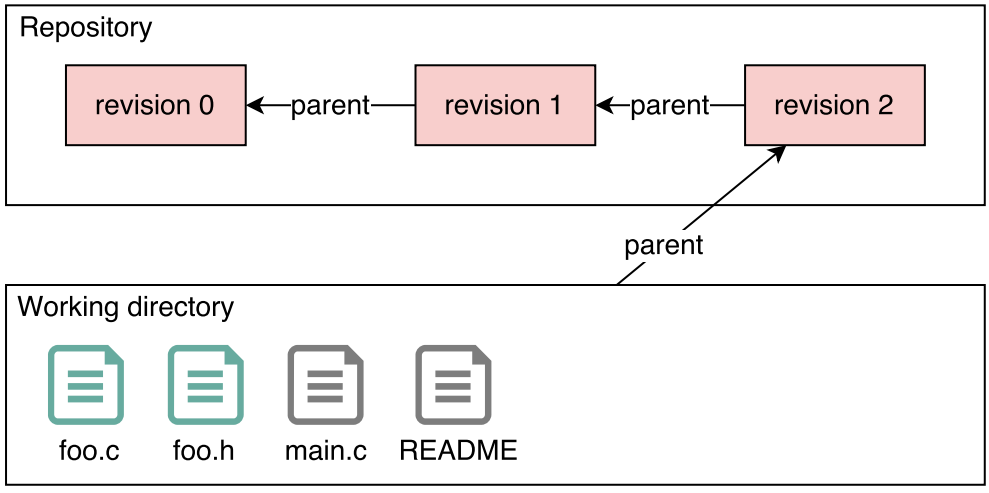


Figure 2.10: Parent revision of a working directory

The **first step** is to get the changelog from the parent revision. As we mentioned before, every metadata file is stored in the revlog format. Thus, even a changelog is versioned and we need to retrieve the content of the changelog from the parent revision. Hence, we need to first search in the changelog index

file how to retrieve the specific parent revision content of the changelog from the changelog data file.

Every changelog entry (a changeset) contains basic information about the commit such as name of the committer, when it was committed, why it was committed, list of modified files and a corresponding manifest nodeid. In the listing 2.1, you can see a sample content of a changeset from a specific revision.

```
changeset:   3:5024abf2219fa1f3715a0e3b3c065bfffadbb
parent_1:    2:58af277f6ffa709a04264613add2466c36508
parent_2:    -1:0000000000000000000000000000000000000000
user:        jsykora
date:        Thu Dec 14 10:55:48 2017 +0100
description: Added foo functionality.
manifest:    3:67ef48ead0a1749cdc20f695018255eb45dba
files_added: foo.c
             foo.h
```

Listing 2.1: Sample content of a changeset

The **second step** is to get the manifest related to the retrieved changeset from the first step. Once we extracted the changelog of the parent revision and got the specific changeset, we get a *nodeid* of the specific **manifest** revision. This manifest nodeid is used for searching in the manifest index file to retrieve the content of the manifest in the parent revision from its manifest data file.

A specific version of manifest describes content of the repository at a specific revision. Manifest contains a list of filenames and revisions of files that are present. In other words, manifest describes the project state by listing each file and its nodeid and by specifying a particular version of each file.

In the listing 2.2 you can see a sample content of a manifest refering to a specific changeset. According to this manifest, there were these four files present in the repository in the specific revision.

```
foo.c            b80de5d138758541c5f05265ad144ab9fa86d
foo.h            45e6ecbd6b23f0a7fbe854e07e7f6af6a76bf
main.c           d9e856ee2adeb1f8442ce5523d2879cd6b4a6
README           393afd3c554af1676a684268f0be33c55bdef
```

Listing 2.2: Sample content of a manifest

The third and the **last step** is to retrieve the files specified in the manifest that have been modified in the current working directory. We will get parent versions (the unmodified version) of the currently modified files.

To sum it up, purpose of the first stage is to retrieve the parent versions of changelog, manifest and all modified files. These parent versions are saved in the memory.

2.2.3.2 Second stage - bottom to top

Purpose of the second stage is to update revlogs of the changelog, manifest and all the modified files (their filelogs). Thus, the second stage constructs new entries for each of the affected revlogs.

Every revlog entry (metadata object) is determined by its nodeid. Nodeid of a metadata object is constructed by hashing the nodeids of its two parent versions and the content of the new version of the file. This way of nodeid construction is the reason, why *Mercurial cannot update repository during the first stage*. For instance, to create the new nodeid of the changelog, you need to have not only its parent nodeids (which you have) but you also need its new content. New content of the changelog has to contain a new changeset. The new changeset has to contain the new manifest nodeid (see listing 2.1). However, you do not have the new manifest nodeid yet. Therefore, you cannot generate the new changelog nodeid, and thus, you cannot create a new version of the changelog. Similarly, to create a new version of manifest nodeid, you also need its new content. However, manifest has to contain new nodeids of the new (modified) files (see listing 2.2). Thus, you are not able to generate nodeid of a manifest until you create new nodeids of the modified files. Hence, you are forced to start updating from the bottom (from the filelogs).

The **first step** is to generate new nodeids of all new metadata. As we explained, we have to start by generating nodeids of the modified files. Then the manifest is updated with these new nodeids and a new version (content) of the manifest is created. With the new version of manifest prepared, a new manifest nodeid can be computed. Finally, this allows us to generate a new version of the changelog and then the new nodeid of that changelog.

The **second step** is the actual commit, i.e. the update of revlogs by adding new entries of the modified files. We could not perform the actual update of revlogs before generating the new changelog nodeid because the changelog nodeid is present in every revlog index entry as a changeset ID (as the old good attribute *linkrev*[12]).

Thus, after the new changelog nodeid is created, we update all affected filelogs, manifest and changelog. For each affected metadata object, we simply append a new record into the revlog index file and append the new delta (or the whole compressed content of the file) to the revlog data file. We start from the logfiles and proceed up to the manifest ending up with the changelog.

Updating in the order from bottom to top ($filelogs \rightarrow manifest \rightarrow changelog$) is more safe because the newly added revision is valid if only and only if the changelog contains the new changeset entry. Therefore, when the new revision is in the process of committing (creating nodeids, append-

[12]In the example table 2.1 you can see a linkrev is not a 160-bit hash but a regular revision number. This is because a revision number can be used as a shorthand within a single repository. However, revision number may not be unique within other users' repositories and only the 160-bit hash is unique across all repositories.

ing entries to the revlog files, etc.), no one knows about this revision and the process of committing does not affect any other previous data (it is appending information only at the end of revlog files). Similar behavior can be observed in the bubble-up method used in Subversion (see section 2.1.2.2), where the whole tree structure is not visible as a new revision to anyone, until the root directory is connected with a revision from the list of revisions.

2.3 Incremental backups in databases

The purpose of an incremental backup is to back up only data changed since the previous backup. The reason of this concept is to save not only space but also time needed for the backup.

There are 2 basic approaches to the incremental backups in databases - **differential incremental backup** and a **cumulative incremental backup**. You can also combine these two approaches in a **combined incremental backup**. Let us examine in the following sections the way each of these approaches perform the backup.

2.3.1 Differential incremental backup

Differential incremental backup backs up all changes since the last backup, regardless of the type of the last backup. For instance, you make make a full backup of your database every week, let us say every Sunday, and you are applying a differential incremental backup. Then, on Monday, you back up only changes since Sunday's full backup. On Tuesday, you back up only changes since Monday's backup. On Wednesday, you back up only changes since Tuesday's backup and so on, until you reach Sunday. On Sunday, you make a full backup and start repeating the whole cycle [36]. See figure 2.11 for the overview of differential incremental backup.

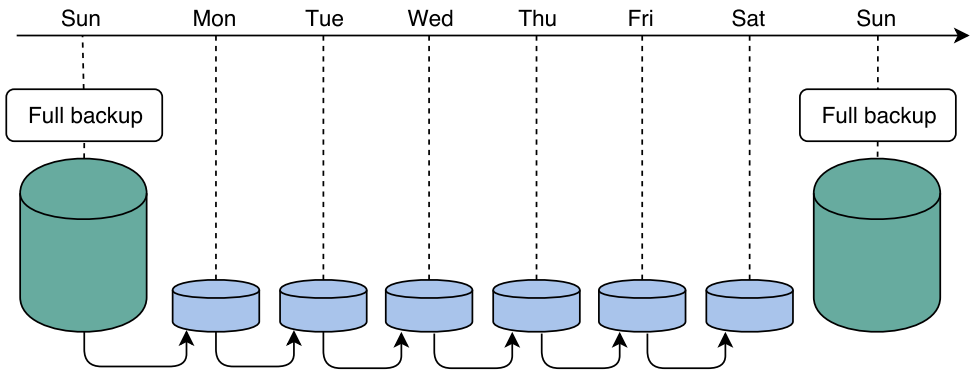


Figure 2.11: Differential incremental backup

Differential backups are created quickly because less data is backed up. However, restoration from differential backups takes longer since you have to compute all the changes starting from the latest full backup. Another disadvantage is a potential impossibility of a data restoration in case a differential backup is missing. For instance, it is Saturday and we lose a differential backup from Wednesday (assume, we are still applying the backup plan from above, where a full backup is performed every Sunday). It is possible to recover data from Tuesday by applying the incremental backups from Monday and Tuesday on the full backup from Sunday. Nevertheless, it is impossible to recover data not only from Wednesday but also from Thursday and Friday. To recover data from Thursday, you need all previous backups from the last full backup, i.e. from Monday, Tuesday and Wednesday.

2.3.2 Cumulative incremental backup

Cumulative incremental backup contains all changes since the last full backup. For example, you make a full backup of your database every Sunday. Then, on Monday, you back up all the changes since Sunday (until here, it is a same logic as in the differential backup). On Tuesday, you back up all changes from Sunday and Monday. On Wednesday, you back up all changes for Sunday, Monday and Tuesday. And so on, until the full backup on next Sunday [36]. See figure 2.12.

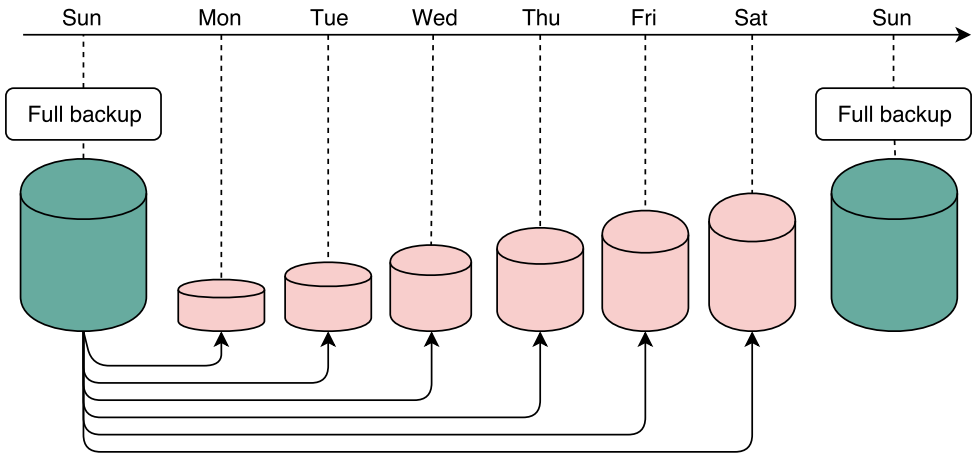


Figure 2.12: Cumulative incremental backup

Cumulative backups are slower and take more space than differential backups since you have to backup more changes. On the other hand, recovery is faster because you do not have to combine all the previous backups. Another advantage is also less vulnerability to the loss of incremental backups. For instance, it is Saturday and you lose Wednesday's backup. You can still restore the database to Monday and Tuesday, but most importantly, you can restore

database also to Thursday or Friday, since you have accumulated changes made before Thursday and Friday, i.e. changes included in the Wednesday's lost backup.

2.3.3 Combined incremental backup

You can also combine differential and cumulative incremental backup for greater flexibility. For example, you can perform a full backup monthly, a cumulative backup weekly and a differential backup daily.

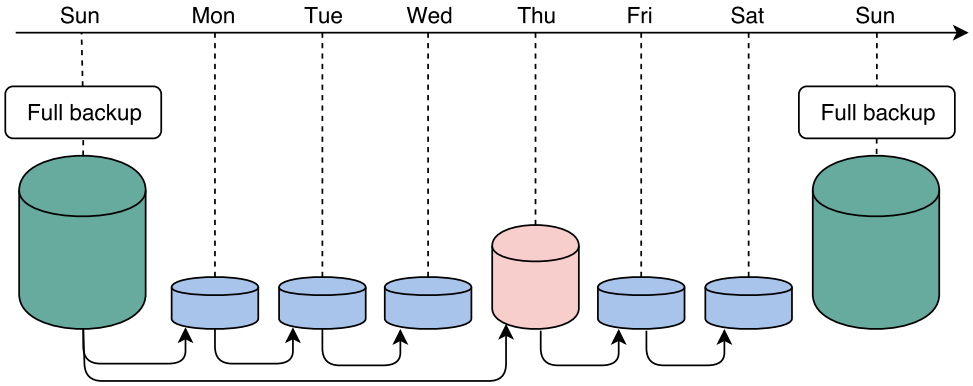


Figure 2.13: Combined incremental backup

For instance, in the figure 2.13, you can see a combined incremental backup. Full backup is performed every Sunday, on Monday, Tuesday and Wednesday is performed a differential backup, on Thursday a cumulative backup and on Friday and Saturday again a differential backup. Hence, if you want to recover database to the Saturday state, you apply the cumulative backup from Thursday on the full backup from Sunday and then you apply the following differential backups from Friday and Saturday.

2.4 Observation

We have studied two version control systems, Subversion and Mercurial, and incremental backups in databases. In this last section, we will observe how these systems update data and evaluate results of this observation for an inspiration in the design of incremental update in Manta Flow.

2.4.1 Data changes

There is an essential conceptual difference between Manta Flow and these analyzed systems. Version control systems or databases are saving directly the user's data they work with, while Manta Flow does not only save the user's

data but, primarily, **Manta Flow generates new data** (data flows) from the user's data and these generated metadata have to be stored as well.

Version control systems need to only manage and store content of the user's project files and the directory tree. On the other hand, Manta Flow has to save not only the content of files and their data structure, but also has to generate data flows, depending on the content of these files. This brings a lot more complexity. For instance, one file change in a version control system produces only one change in the file itself (or one structural change in the directory tree). In Manta Flow, one change in a file produces not only a change on the file level but also generates changes in the data structure and new data flows. For instance, adding a single insert statement into a database script results into the creation of a new node for the insert statement, new nodes for the variables, columns and tables used in this insert statement and new data flows representing the insertion of data from source table to the target table[13].

In consequence of this fundamental difference of the managed data, **Manta Flow client does not currently track changes of the users data**. To understand, why it is not so simple to track the changes on the client's side, we have to explain how the Manta Flow client works. According to the documentation [37], Manta Flow client application does sequentially these three steps for each source system:

1. Extracts database dictionaries, DDL scripts, workflows and configurations from the source systems

2. Performs data flow analysis on both of the extracted and the provided user's data

3. Exports the resulting metadata in the form of a linearized graph in the form of the CSV files

If we wanted to track the changes on the client's side, Manta Flow would have to not only check for the changes in the scripts, but it would also have to connect to the databases to retrieve, for instance, changes in the database schema. What more, Manta Flow would have to perform an *impact analysis* to check what impact the particular changes had on the data flow.

Consequently, Manta Flow cannot automatically produce an input representing changes between single revisions. Therefore, changes for the incremental update in Manta Flow have to be **specified manually** and from these manually specified changes will be created an input in the form of a subgraph for the incremental update.

[13]See in the section 1.7, how the source code files are stored in Manta Flow

2.4.2 Repository update

We can observe, the methods of the incremental update in Subversion and Mercurial are **creating the changes aside**, avoiding conflicts with other users reading the repository.

In Subversion, the *bubble-up method* is creating a new revision as a directory tree aside the repository structure. The unchanged parts of the new directory tree are linked to the existing repository structure. Until the new directory tree is linked to the list of revisions, the changes in the repository are invisible for all other reading users.

In Mercurial, changes are written at the end of the metadata files. Thanks to the structure of these metadata files ($changelog \rightarrow manifest \rightarrow filelog$), changes written at the end of these files are also invisible for all other reading users until a new changeset (i.e. a new revision) is written to the changelog.

Manta Flow metadata structure allows similar behavior. Current unchanged data revision validity is either extended or their revision validity is not changed at all and a newly added data is appended to the current structure with a revision validity starting from the newly committed revision. In any case, users reading any previous, already committed, revision are unaware of the changes being made within the new revision thanks to the revision validity represented on the edges.

Another observation of the methods used by the above described version control systems reveals, the changes are applied from **bottom to top**. However, we cannot directly apply some of these methods due to the nature of entirely different data used by Manta Flow. Nevertheless, when designing a new method of the incremental update, we may analyze the possibility of merging a subgraph into the main graph from bottom to top, instead of the current way of updating from top to bottom.

2.4.3 Incremental backups

The second part of this chapter belonged to the analysis of the methods of incremental backups in databases. There are two basic methods, differential and cumulative incremental backup. Additionally, these methods can be combined.

Incremental backups aim to save the changes made in some specific time period and when needed, these changes are applied on the latest full backup to reconstruct a particular state. However, as we explained in the section 2.4.1, it is difficult and currently impossible to save all the changes made within a specific time period. Moreover, the idea of saving changes made between revisions, having only one revision in the database and applying these changes when another revision is required, is unsuitable for the data structure in Manta Flow. The data structure in Manta Flow is built for versioning the data using revisions, not for applying changes.

2.4.4 Summary

Manta Flow works with a complex data where one change in a file may invoke more than one change in the whole structure. Consequentially, we are currently unable to track all the changes and automatically create inputs representing these changes on the client's side without an impact analysis. The current way-out is to limit the variety of changes for incremental update and users will be forced to select these changes manually. Nevertheless, it still disallows us to directly apply any of the methods of incremental update in version control systems or any of the methods of incremental backup in databases.

The only useful inspiration for the incremental update in Manta Flow is using the method of update from bottom to top applied in the version control systems instead of the expected update from top to bottom. We will analyze usefulness of this approach in the new design of the incremental update in Manta Flow.

Chapter 3

Analysis and Design

Purpose of this chapter is to **analyze the possibilities of incremental update** in Manta Flow and subsequently **design a new method of incremental update** based on this analysis.

Important criteria of the analysis is the number of necessary changes in a graph, the ability to find data flows related to a certain time interval and the amount of time necessary for querying.

3.1 Revision data representation

Petr Holeček, author of the current design of version control in Manta Flow [1], studied different revision data representations in Manta Flow. Petr Holeček examined three options of storing revision data in the graph database Titan used by Manta Flow:

- Revision data stored on the data vertices
- Revision data stored on separate vertices
- Revision data stored on the edges

First option, storing revision data on the data vertices, allows effective revision tracking only of the data vertices but not of the data flows. Tracking revision validity of the data flows is impossible without application of another support structures, such as vertex duplication. However, application of these structures leads to an unacceptable redundancy and querying complexity. Therefore, storing revision data on the data vertices was rejected due to the ineffective data flow revision trackability.

Second option, storing revision data on separate vertices, is even more ineffective than the first option. Storing revision data on separate vertices also requires extra support structures for data flow revision tracking. Moreover, storing revision data on the special vertices requires additional connection of

data vertices and data flows to these separate vertices. That leads to an even higher data redundancy and querying complexity. Hence, neither this option was accepted.

The third option, storing the revision data on the edges, was accepted as the most suitable. This option does not generate redundant vertices or edges, since it uses the already existing edges. Also, storing revision data on the edges allows a natural data flow revision tracking. On top of that, thanks to the usage of vertex-centric indexing, querying on the database and searching for the valid vertices and edges is effective as well.

Considering results of Petr Holeček's analysis, **we will keep the revision data representation on the edges**. Moving revision data either to the data vertices or to the separate vertices would not bring any advantage. However, the current revision data representation is incapable of performing the incremental update. Hence, we will analyze different revision data representations on the edges in the graph database Titan and we will choose the most suitable one.

3.1.1 Closed end revision

In the current Manta Flow version, the end revision, i.e. the last valid revision of the tracked metadata object, is represented by the parameter tranEnd. This parameter always holds an **explicitly specified revision number**. Therefore, this type of end revision is called a *closed* end revision.

We have partially examined the closed end revisions at the end of the first chapter, in the section 1.11. We observed, adding changes to the main graph within a new revision results into the update of end revisions of all other valid nodes and edges from the latest revision in order to keep the revision history consistent. In the following text, a deeper analysis of this revision data representation will be performed.

In the figure 3.1, is an example demonstrating a **small change** in a versioned graph. This graph holds metadata history from revisions 1, 2 and 3. We decide to remove a node D (red color) and add a new node α (green color). Therefore, a new revision 4 is created and an update is performed at all affected nodes and edges.

Notice, the first number on the edge represents the first valid revision, i.e. the parameter tranStart. The second number stands for the latest valid revision, i.e. the parameter tranEnd. Also notice, root node of the whole graph, node A, is always present in all revisions, i.e. it is a *super root*. Therefore, we may ignore this node when committing a new revision.

We can see in the figure 3.1, we had to update end revisions of all nodes and edges remaining in the revision 4 (orange color). If we did not update their end revisions from 3 to 4, they would be absent in the new revision 4. For instance, if we did not update end revision of node G from 3 to 4, this node would be absent in the revision 4.

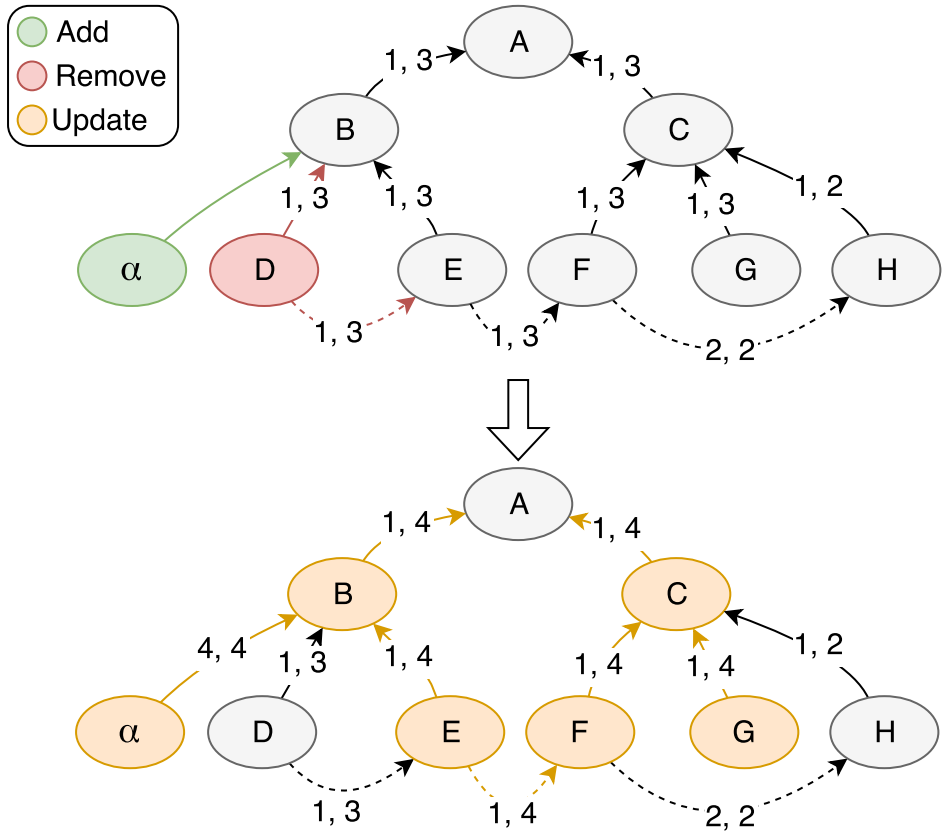


Figure 3.1: Small change in the graph with closed end revisions

On the other hand, we did not have to update any of the nodes or edges absent in the revision 4. Naturally, we ignored the node H since this node was already absent in the revision 3. Moreover, we also ignored the node D. Thus, node D was *indirectly removed* in the new revision 4.

In the next example, we will perform a **substantial removal**. We will use the same graph from the first example. This time, we will remove all nodes and edges but the node B. Additionally, we will add again the node α. See the figure 3.2.

Obviously, in the figure 3.2, only nodes B and α had to be updated. Rest of the nodes remained in the revision 3. Thus, rest of the nodes was indirectly removed in the revision 4. For instance, node C is valid only up to the revision 3 (including), since its tranEnd parameter remained set at the revision number 3.

To summarize it, when using revisions representation with closed ends, revision data has to be updated at all nodes and edges present in the new revision. On the other hand, revision data of nodes and edges either remaining in the latest revision or already removed in some of the previous revisions can

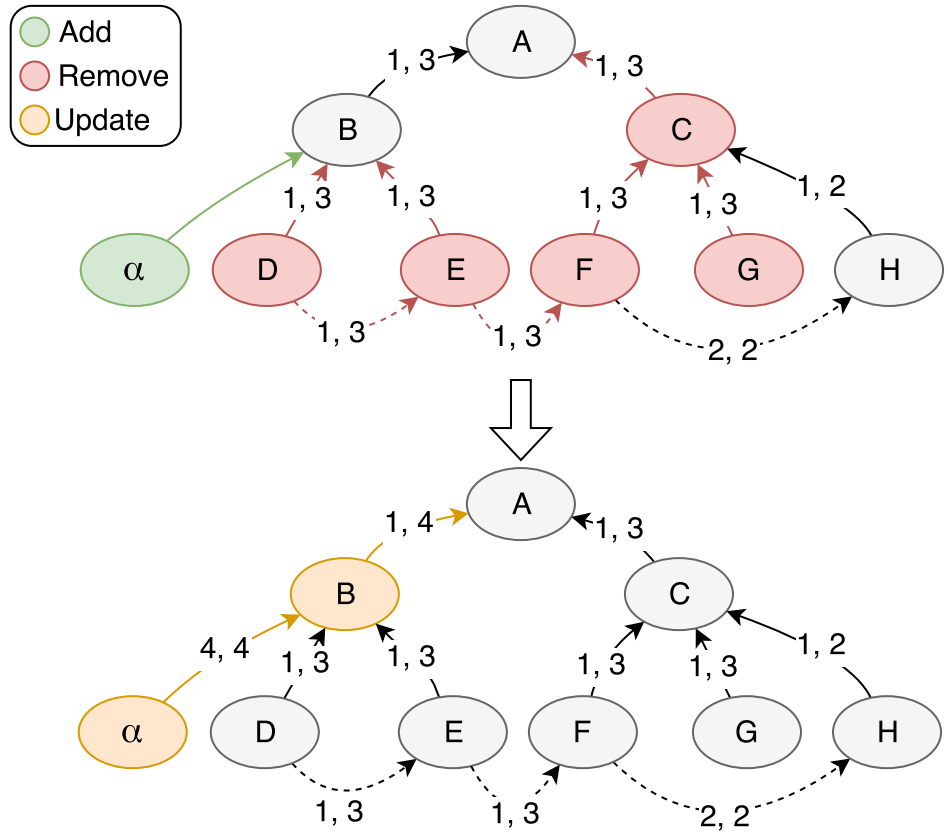


Figure 3.2: Large removal in the graph with closed end revisions

be ignored. In other words, we do not have to set the end revisions of nodes and edges absent in the new revision. We only have to update end revisions of nodes and edges present in the latest revision and remaining in the new revision as well. And, of course, we also have to create the newly added nodes and edges and set their revision data.

3.1.2 Unclosed end revision

Unclosed end revision represents the **latest revision of all revisions**. That means, a node or an edge with an unclosed end revision is definitely present in the latest revision, even though, the latest revision number is not explicitly specified. Let us study the impact of unclosed end revisions closely.

Assume, we have three revisions 1, 2 and 3. Main graph holds the revision data on the edges, using the unclosed end revisions. The unclosed end revision is represented by the ∞ symbol. Again, we decide to perform a **small change**. We will remove the node D and add a new node α. Hence, a new revision 4 is created and an update is performed within the newly created revision 4. See the figure 3.3.

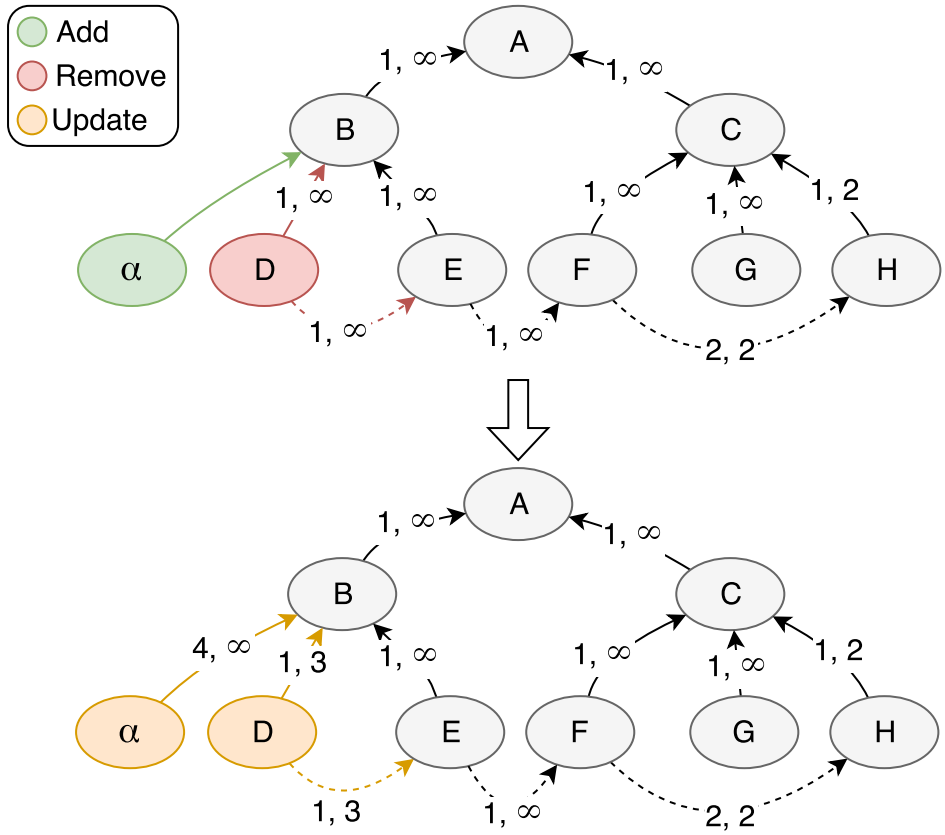


Figure 3.3: Small change in the graph with unclosed end revisions

We can immediately observe the difference when using unclosed end revisions instead of the closed end revisions. We only had to add the node α, set end revision of the node D to the revision number 3 and also set end revision of its outgoing flow edge to the number 3. That means, we had to update only the changed nodes and edges.

Apperently, only the changed nodes and edges had to be updated. As we know, incremental update is built on the idea of updating only the changed parts. Therefore, using unclosed end revisions seems as an effective revision data representation for incremental update.

Nevertheless, using only the unclosed end revisions may be ineffective when performing a **substantial removal** in the graph. We will demonstrate it in the next example. Assume, we have three revisions 1, 2 and 3. We have still the same graph, using the unclosed end revisions. We decide to remove all nodes and edges except the node B. Additionally, we also append the node α to the node B. As in the previous examples, we create a new revision 4 and perform update of the graph. See the figure 3.4.

This time, update had to be performed at all the removed nodes and edges.

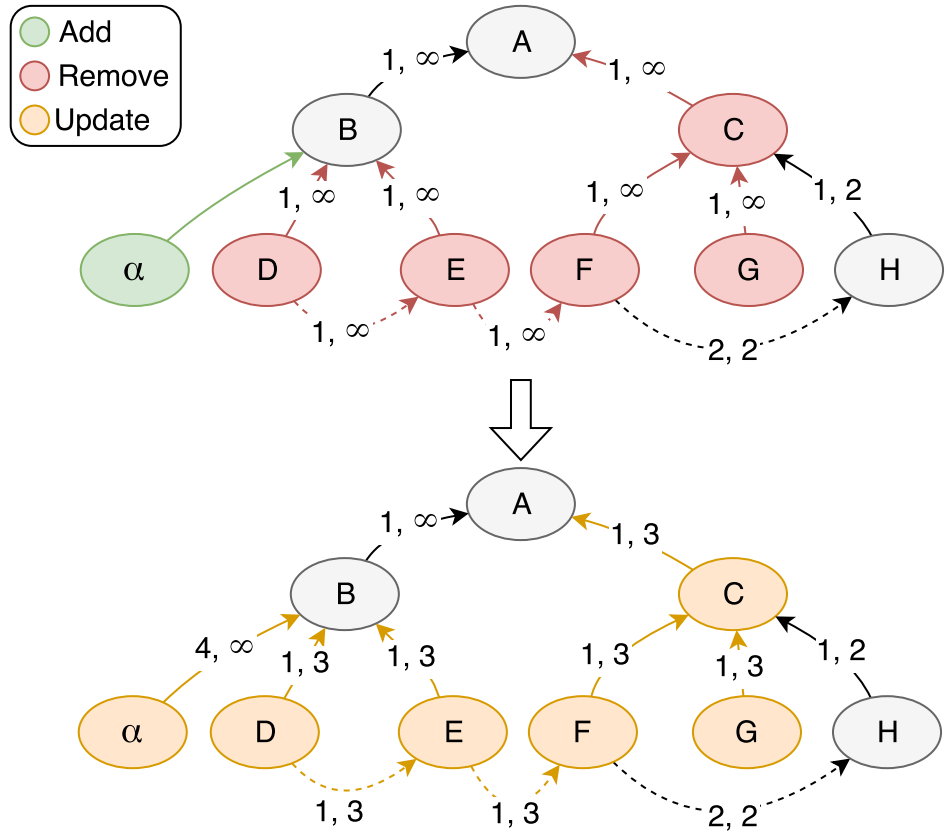


Figure 3.4: Large removal in the graph with closed end revisions

End revisions of all the removed nodes and edges had to be set to the revision number 3, i.e. they were set to their latest valid revision. If the end revision of any removed node was left unchanged, i.e. at the value (∞), this node would seem still present in the revision 4.

In conclusion, when using the unclosed end revisions, the number of necessary updates is directly proportional to the number of changed nodes and edges. Therefore, unclosed end revisions allow effective small updates. However, the greater the change, the more steps has to be performed. In the extreme cases, such as removal of all nodes and edges, the update has to be performed at all nodes and edges. Degrading incremental update into a full update.

3.1.3 Two-level revision

We observed in the previous examples, closed end revisions are suitable when a large portion of the graph is removed. However, closed end revisions are unsuitable for the small updates. On the other hand, unclosed end revisions are suitable when the low number of changes is performed but they are un-

suitable when a large portion of the graph is removed. Obviously, these two approaches are *complementary*. Therefore, we **combine closed and unclosed end revisions** in such a way to benefit from both of their advantages and avoid the drawbacks natural for each of them.

Instead of two revision parameters tranStart and tranEnd, there will be four revision parameters[14] *majorStart*, *majorEnd*, *minorStart* and *minorEnd*. Parameters majorStart and majorEnd represent **major revisions**. Parameters minorStart and minorEnd represent **minor revisions**. Major revisions will keep the system of the closed end revisions. Minor revisions will use the unclosed end revisions.

We can interpret major revisions as integers and minor revisions as their decimal part. Hence, now we will not have only integer revisions $1, 2, 3, \ldots$ but we will have revisions $1.0, 1.1, 1.2, 2.0, 2.1, \ldots$ and so on.

When a **full update** is performed, then it is a transition from one major revision to another major revision. For instance, when performing a full update from a revision 1.5, the new revision will be 2.0. On the other side, when an **incremental update** is performed, it is a transition from one minor revision to another minor revision. For example, performing an incremental update from a revision 1.5 results into a new revision 1.6.

Let us demonstrate this two-level revision data representation on the following examples. As you would expect, the same graph from the previous examples will be used. Nevertheless, this time, our graph contains revisions $1.0, 1.1, 1.2, 2.0$ and 2.1. We will add new node α and we will remove node D within an **incremental update**. Thus, a new revision 2.2 will be created. See the figure 3.5.

We can see in the figure 3.5, only the changed nodes had to be updated. According to the revision data on the edges[15], node α was added in the revision 2.2 and its last valid revision is also 2.2. How do we know its latest revision is 2.2? Because ∞ stands for the latest subrevision within its major end revision (i.e. majorEnd parameter). And since the latest subrevision within the major revision 2 is .2, the last valid revision of node α is 2.2.

To complete our list of examples, we will demonstrate usefulness of major revisions when a substantial part of a graph is removed. We will remove all nodes except the node B. On top of that, we will append α node to the node B. Again, assume, we have revisions $1.0, 1.1, 1.2, 2.0$ and 2.1. But this time, we will perform a **full update**. Hence, a new revision 3.0 will be created. See the figure 3.6.

Again, we can clearly state, we only had to update the two remaining nodes B and α. Since major revisions are superior to minor revisions, we had to update only major revisions of the changed nodes and we could have left all

[14]For the simplicity, we will shorten the name of parameters by removing the "Tran" word

[15]On the first line are major revisions. On the second line are minor revisions.

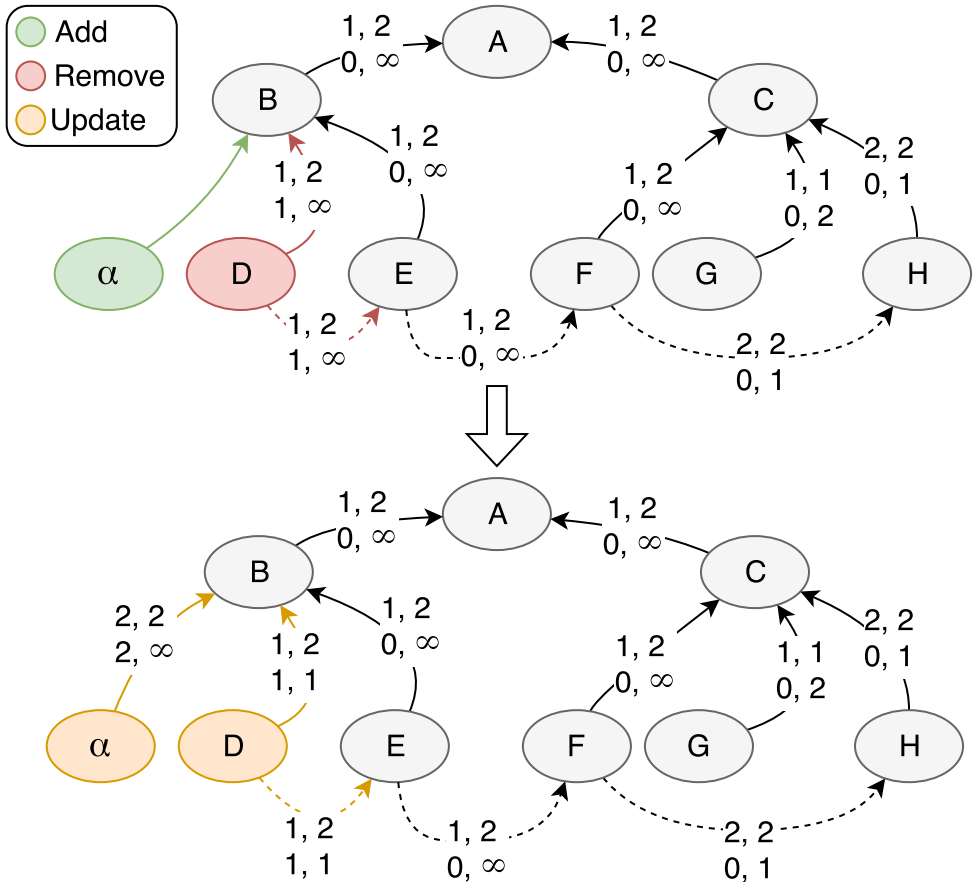


Figure 3.5: Small change in the graph - incremental update

the other minor revision data unaffected. Therefore, removing a substantial part of a graph, using the major revisions, requires the minimum of update steps to keep the history of the graph database consistent.

When a node is removed, its minor end revision (parameter minorEnd) is set to a specific value. For instance, when a node H was removed in the sub-revision 2.2, its minor end revision was set to 1, i.e. its end revision was set to 2.1. However, in case of transition to the major revision 3, we left the minor end revisions of the removed nodes unclosed. For example, node D was removed but its minor end revision is left unclosed (i.e. ∞). Nevertheless, its major end revision was left at the value of 2. Thus, there was no reason to close the minor end revision. In other words, leaving the minor end revision of a removed node unclosed when performing full update did not bring any inconsistency to the graph since minor revision is inferior to the major revision.

To sum it up, combining closed and unclosed end revisions in the system of major and minor revisions allows us updating only the necessary nodes

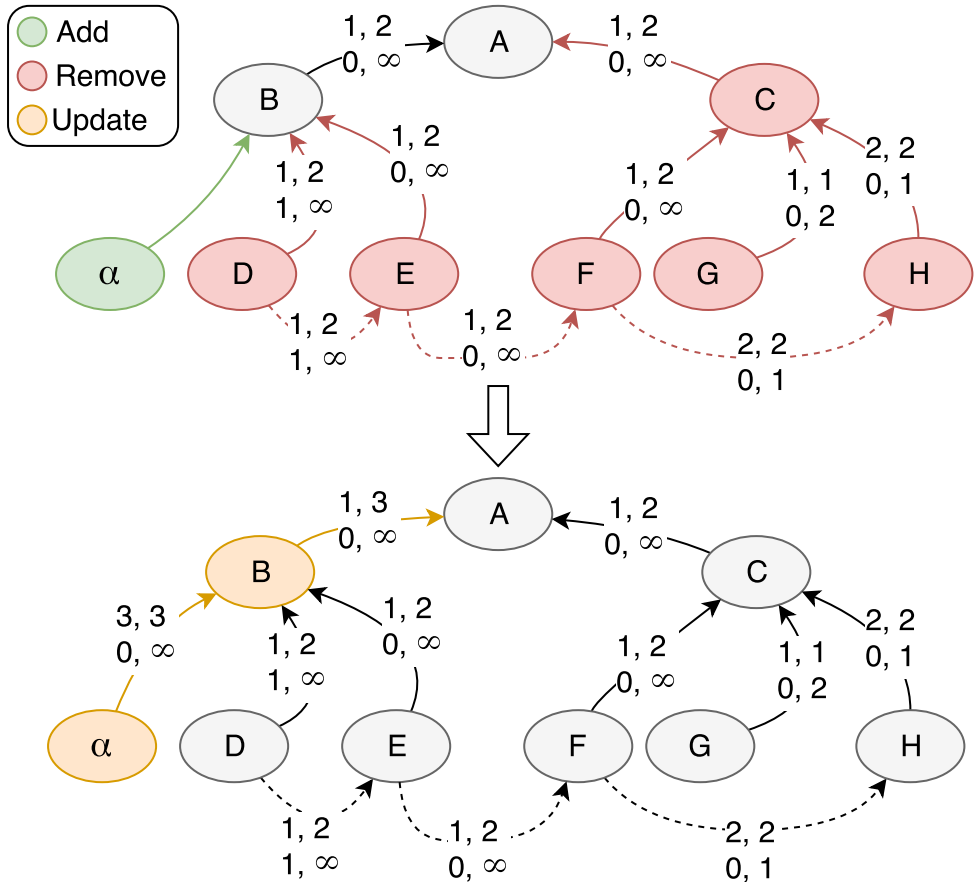


Figure 3.6: Large removal in the graph - full update

and edges. Using this system of subrevisions allows us to perform effectively both full update and incremental update. Therefore, we will use the two-level revision data representation for the new method of incremental update as well as for the current method of full update in Manta Flow.

3.1.4 Integer vs Decimal

Representing two-level revision by four attributes (majorStart, majorEnd, minorStart and minorEnd) brings unnecessary complexity when searching for the nodes and data flows related to a certain revision interval.

For instance, if we want to find all child nodes of a specific node that are valid in the revision interval 2.3–4.5, we have to filter only nodes, whose revision properties are fulfilling the following condition:

$$[[majorStart < 4] \vee [majorStart = 4 \wedge minorStart \leq 5]] \\ \wedge \\ [[majorEnd > 2] \vee [majorEnd = 2 \wedge minorEnd \geq 3]]$$

This expression cannot be more simplified due to the nature of the two-level integer system. However, when filtering child elements in a vertex query, we cannot directly apply a disjunction within one query. Hence, we have to convert this expression into a disjunctive normal form (DNF), where each clause represents one filter query:

$$[majorStart < 4 \wedge majorEnd > 2] \vee \\ [majorStart < 4 \wedge majorEnd = 2 \wedge minorEnd \geq 3] \vee \\ [majorEnd > 3 \wedge majorStart = 4 \wedge minorStart \leq 5] \vee \\ [majorStart = 4 \wedge minorStart \leq 5 \wedge majorEnd = 2 \wedge minorEnd \geq 3]$$

Obviously, we have to perform four queries. Performing four queries would take definitely longer time than performing only one query. If we wanted to perform only one query, we would have to create a new class representing this two-level revision (containing all four revision attributes), store it as an edge property, implement its own comparison method and use these property objects for revision comparison.

Titan allows storing custom objects as properties on both edges and vertices. However, Titan supports indices allowing interval search (mixed indices) only for a certain set of data types. For instance, Titan supports indices allowing interval search for integers, floats, doubles, dates etc. Custom data types can be used as index keys only for the standard (composite) indexing, which support searching based only on the equality.

That means, Titan would not be able to perform interval search when using our custom object property as an index key. Since interval search is fundamental to the revision validity querying, we cannot use custom class to avoid the issue of multiple queries when filtering elements according to their revision validity.

Nevertheless, there is an elegant solution to this issue. We can use two **numbers with decimal part** instead of four integers. That means, we will have again only two revision properties representing start and end revision. The integer part represents the major part and the decimal part represents the minor part.

The update logic remains the same. When performing full update, only the integer (major) part is affected, when performing incremental update, only the decimal (minor) part is affected.

Moreover, querying is more effective since mixed indices can be used and also only one query has to be performed when searching for elements within a specific revision interval. For example, when searching for the child nodes valid within the revision interval 2.3–4.5, the following condition has to be fulfilled:

$$start \leq 4.5 \wedge end \geq 2.3$$

The only disadvantage of this solution is in the **maximum number of minor revisions** within one major revision. The maximum number of minor revisions within one major revision is defined by the maximum value in the decimal part. The maximum value in the decimal part is derived from the data type used for this property.

When using integer (as in the rejected option with four revision attributes), the maximum number would be `INTEGER.MAX_VALUE`. However, when storing float or double data type to the database, Titan cuts their decimal part during serialization. Floats are stored with up to three decimal digits (maximum value is .999) and doubles are stored with up to six decimal digits (maximum value is .999999) [38]. Thus, when using float data type, we can perform up to thousand incremental updates within one major revision. Analogically, when using double data type, we can perform up to million incremental updates within one major revision.

Titan allows storing floats and doubles in their entirety without any decimal digits cuts using special attributes FullFloat, resp. FullDouble. However, these attributes do not allow vertex-centric indexing. Since vertex-centric indexing is essential for us when querying, we cannot use these special attributes. Also notice, the reason why floats and doubles are cut of their decimal parts when serialized, is due to their potential usage as index keys. Cutting decimal parts of floats and doubles allows more effective vertex-centric indexing.

Since we expect some users using only incremental update, we will choose the **double data type** because it allows creation of million of minor revisions (within one major revision) in contrary to the maximum of one thousand minor revisions when using the float data type.

3.1.5 Revision tree

In the current implementation, revisions are stored in a standalone graph, in the revision tree. Revision tree has on top a **revision root**. From this revision root lead **hasRevision** edges to the **revision nodes**.

We have to change the data type of revision number, saved in the revision node, from Integer to Double. Rest of the logic, such as creation of a new revision or committing the latest revision, remains the same. We only have to distinguish between an incremental update and a full update, to always create a correct revision number.

Additionally, we will **add new properties** to the revision root and to the revision node to speed up querying. We will add properties latestCommittedRevision and latestUncommittedRevision to the revision node and we will add properties previousRevision and nextRevision to the revision nodes.

Next, we will define **technical revision** (the first revision created at the beginning) as a revision with number 0.000000. Depending on the next type of update, either a non-technical revision 1.000000 or a non-technical revision 0.000001 is created next.

Finally, we will keep tranStart and tranEnd properties on the hasRevision edges. These properties are useful when the whole revision node needs to be retrieved while knowing only its revision number. For instance, when we want to know when a specific revision was committed, we can quickly find its revision node specified by its revision number and retrieve the commit time property from this revision node.

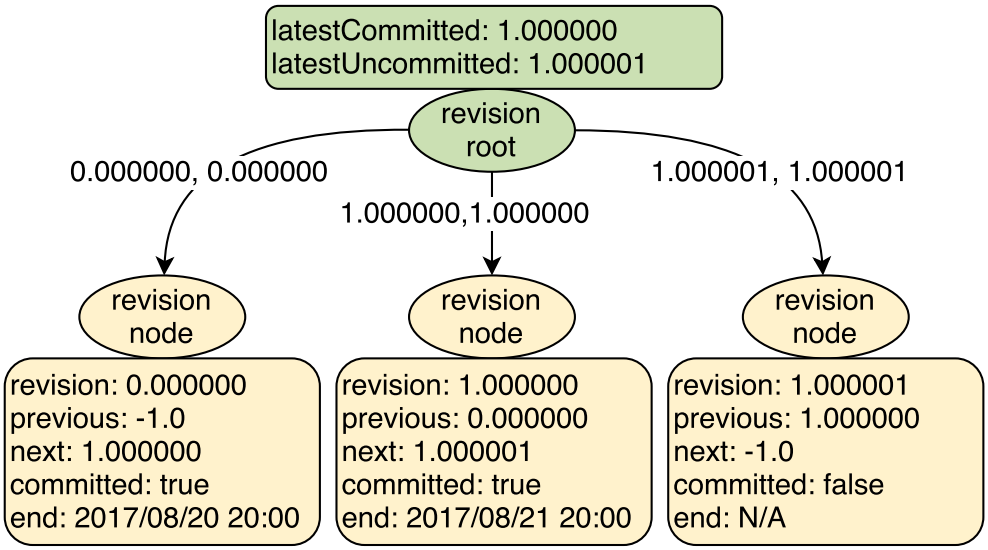


Figure 3.7: Example of a revision tree

In the figure 3.7 is an example of revision tree. In this example are three revisions. Technical revision 0.000000 and two non-technical revisions 1.000000 and 1.000001. Clearly, revision 1.000000 was created as a new major revision (full update) and revision 1.000001 was created as a new minor revision (incremental update).

Since the latest revision 1.000001 is not committed yet, revision root registers revision 1.000000 as the **latest committed revision** and revision 1.000001 as the **latest uncommitted revision**. If all revisions were committed, i.e. no uncommitted revision exists, a special value -1.0, indicating an empty (null[16]) value, would be saved as the latest uncommitted revision in

[16]Null cannot be set as a property value. Therefore, a value -1.0 is used instead.

the revision root.

Obviously, when searching for the latest committed revision, we have to retrieve only one property from the revision root instead of retrieving all revision nodes and searching for a committed revision node with the highest revision number.

The special value -1.0, indicating an empty (null) value, is used also in the properties storing **previous revision** and **next revision**. Previous revision is an immediately preceding revision and next revision is an immediately following revision.

In the figure 3.7, revision 0.000000 has no previous revision -1.0 and the next revision is 1.000000. Analogically, revision 1.000000 has previous revision 0.000000 and next revision 1.000001 and revision 1.000001 has previous revision 1.000000 and no next revision.

Saving number of the previous and next revision is important, because the new system of mixed revisions does not follow any **sequence**. The old revision system (with integer revision numbers) followed an arithmetic sequence with difference 1. Hence, it was possible to determine previous and next revision from the current revision number. For instance, when you have revision 5, the previous revision has, without doubts, number 4 and the next revision has definitely number 6.

The new system of major and minor revisions does not follow any sequence. When you have revision 1.000000, you cannot determine from this revision number, whether the next revision is 1.000001 or 2.000000. Without saving previous and next revision number directly in the revision node, we would have to iterate through a subset of revision nodes to find the one with maximum or minimum revision number, to find the previous or next revision number.

For example, when searching for the previous revision number of the revision node 2.000000, we would have to retrieve all revision nodes that are less than 2.000000 and find the one with the maximum number. The more revisions we have, the longer this operation may take. Therefore, we introduced new properties previousRevision and nextRevision solving this issue.

The previous and next properties can be used especially when **pruning old revisions**. For instance, when we want to remove all revisions and keep only the three latest committed revisions, we need to know number of the latest revision that is to be removed as well. We will get this number the following way:

1. Get revision root and get the latest committed revision number

2. Find the latest committed revision node (using tranStart and tranEnd edge properties)

3. Get previous revision node number and find the the previous revision node

4. Get previous revision node number and find the the previous revision node

5. Get previous revision node number → this is the revision number we are looking for

Once we have the revision number of the latest revision to be removed, we simply perform removal of all revisions less than or equal to this revision number.

3.1.6 Full update vs incremental update

When using the two-level revision data representation, we always have to decide, whether the update should be performed on the level of major revisions or on the level of minor revisions. In other words, we always have to choose between the full update or the incremental update. Luckily, we can unambiguously define what level the update should be performed on, if we know how many nodes and edges is present in the latest revision and how many nodes and edges will be removed in the new revision.

Let *closed* be the number of nodes and edges necessary to update when closed end revisions are used and *unclosed* be the number of nodes and edges necessary to update when unclosed end revisions are used. Next, we want to compare these two functions. Let us find out, when it is more effective to perform an update using closed end revisions than using unclosed end revisions. That means, we want to find out when the number of updates using closed end revisions is less than the number of updates using the unclosed end revisions:

$$closed < unclosed \tag{3.1}$$

Let *before* be the number of nodes and edges valid in a graph before the update, *add* be the number of nodes and edges we want to add to the new revision and *del* be the number of nodes and edges we want to remove in the new revision.

Next, closed end revisions require to update only nodes and edges valid in the new revision. Those are nodes and edges valid in the original revision, except the removed nodes and edges but also additionally extended by the newly added nodes and edges. On the other hand, unclosed end revisions require to update only newly added or removed nodes and edges. Hence, substituting to the inequality 3.1, we get:

$$before + add - del < add + del \tag{3.2}$$

Removing the *add* variable on both sides of the inequality 3.2, followed by a simplification, we proceed to:

$$before < 2del \tag{3.3}$$

$$\left\lfloor \frac{before}{2} \right\rfloor < del \tag{3.4}$$

When simplifying the inequality 3.3, we divided it by 2 to clearly demonstrate in the final form 3.4 the following claim:

Claim 1 *Removing more than half of nodes and edges from a graph, when transitioning to a new revision, requires less updates when using closed end revisions than unclosed end revisions.*

Notice, the effectiveness of both representations is unaffected by the number of added nodes and edges. It is a logical implication, since creating a new node or an edge always requires an update. Moreover, if we only add new nodes or edges, update using the closed end revisions would be still less effective due to the already existing nodes and edges before the update.

Also notice, we did not consider the already removed nodes and edges. Already removed nodes and edges do not need to be updated. Therefore, we could ignore these.

Finally, in the last form of the inequality 3.4, we used the floor function. We could have also left this inequality without the floor function and it would be still correct. However, we wanted to emphasize we work with integers, with the number of updates. If we used the ceil function (higher integer), the inequality would be incorrect in case of removal the "smaller half" from graphs with odd number of nodes and edges.

Nevertheless, the inequality 3.4 is only a **rough estimation**. We considered only the number of necessary changes. We did not distinguish between updating nodes and updating edges. We did not include the complexity of creation of the input, although, each approach requires different input. And most importantly, we did not consider how these changes would be applied.

3.2 Update

Having chosen a suitable revision data representation allows us to proceed to the most essential part of this analysis, the update. Update is responsible for propagation of user's new revision data into Manta Flow graph database.

Update in Manta Flow consists of these three steps:

1. Create a new revision
2. Update user's changed data within the new revision
3. Commit the newly created revision

Creation and commit of a new revision will not differ from the current implementation. A new revision node is created in the revision tree, its revision number property is set to a proper value and once the update of user's changed data is finished, its commit attributes are set to proper values as well. Since logic of these operations in the revision tree remain the same and are pretty straightforward, we will focus only on the update of user's changed data.

We will describe two types of updates, **full update** and **incremental update**. Our primary focus should be directed only towards the incremental update. However, due to the change of revision data representation, we have to also slightly rework the full update. Therefore, in the following text, we will describe changes in the full update and then we will describe a new method of the incremental update.

3.2.1 Full update

Full update receives on the input the whole new version of the main graph. This new graph is **merged** to the versioned graph in Manta Flow in the order order it comes from the input, i.e. first resources and layers are merged, then nodes and attributes and the data flows are merged last. When source code files are sent to be stored as well, they are being send asynchronously via multiple threads.

Merging single vertices and data flows is almost the same as explained in the first chapter, in the section 1.10.4. We only have to incorporate changes of revision data properties on the edges.

When a metadata object is merged, it is first checked, whether it exists in the latest revision or not. As we already know, comparison of two metadata objects is performed via their **equality criteria**.

If the merged object does *not* exist in the latest revision, it is newly created. Its start revision parameter is set to the new revision number (both integer and decimal part). Its end revision parameter is set to the new revision number (integer part) but its minor part (decimal part) is set to the maximum value. Hence, the new object will have these revision properties:

$$start \leftarrow newMajor.000000$$
$$end \leftarrow newMajor.999999$$

If the merged object does exist in the latest revision, its start revision parameter (both integer and decimal part) remains unaffected, major part (integer part) of its end revision parameter is incremented by one and the minor part remains unaffected. Hence, the updated object will have these revision properties:

$$start \leftarrow unchanged.unchanged$$
$$end \leftarrow newMajor.unchanged$$

Let us demonstrate the description of full update on an example. Notice, we will use shorter form (two digits) of the minor revisions .00 and .99. Assume, in the database is only one revision 1.00. A new input is merged by full update within a revision 2.00. Below is described the merge process step by step in single figures 3.8, 3.9, 3.10 and 3.11.

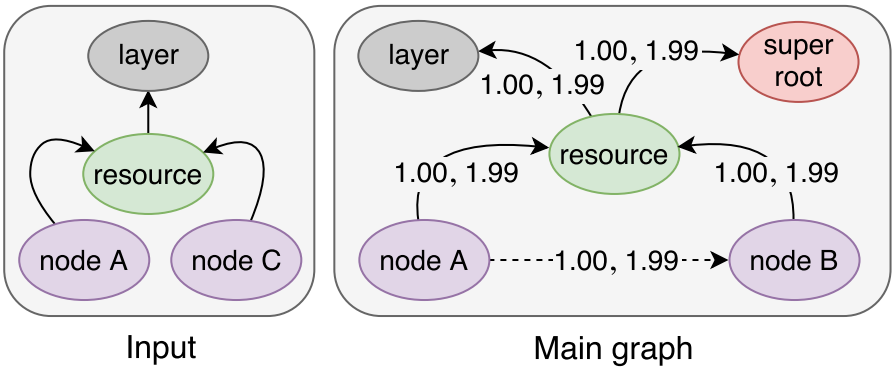


Figure 3.8: Full update - initial state

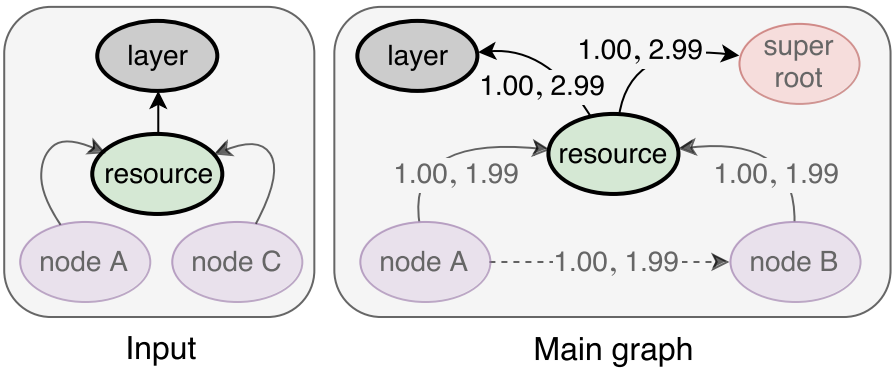


Figure 3.9: Full update - merging resource and layer

Obviously, full update performed transition to the revision 2.00 as expected. All objects from the input had their revision parameters updated according to the description from above. Node B was missing in the input. Therefore, it was ignored during the merge. Consequentially, node B was removed together with its data flow and is no longer valid in the revision 2.00.

Also notice, resource and layer were merged at once. Layer vertex has no parent vertex. Therefore, it has to be merged together with one of its resources. All other vertices have their parents clearly defined. Hence, all other vertices are merged individually.

To sum it up, full update will be performed almost identically as it has been performed until now. The only difference is, we update exclusively the major parts (integer parts) and we leave minor parts (decimal parts) unaf-

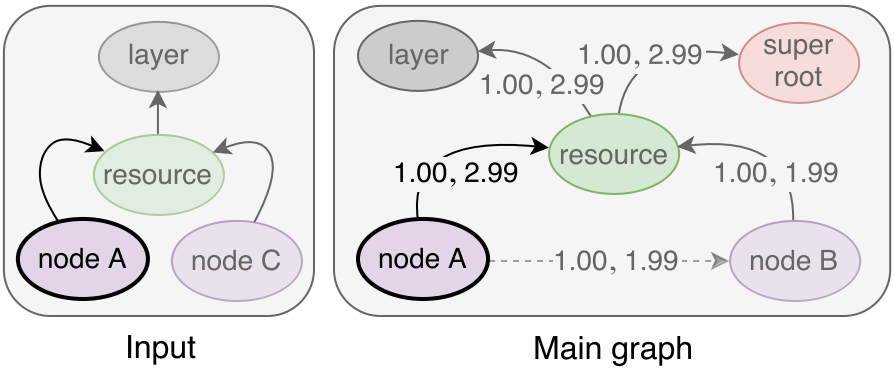


Figure 3.10: Full update - merging node A

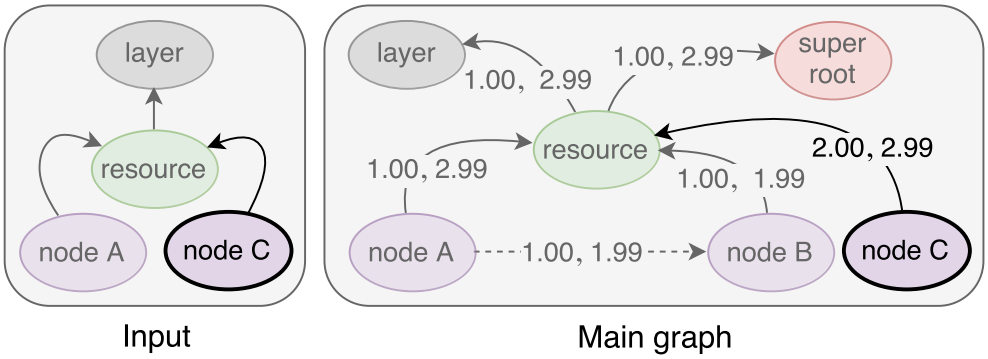


Figure 3.11: Full update - merging node C

fected. The only exception is an object creation. The start revision is set to the newly created revision always starting with minor revision zero (*major*.00, and the end revision is set to the maximum value in the new major revision.

3.2.2 Incremental update

Subgraph merge within incremental update is a **combination of full-update-like graph merge and subgraph removal**. The input subgraph[17] is merged node by node until a specially marked node is reached. This marked node indicates, a change has been performed somewhere in the subgraph starting with this marked node.

When this marked node is reached, the whole subgraph starting from this marked node is removed. The removal is performed simply by setting the end revision property to the latest revision:

$$start \leftarrow unchanged.unchanged$$
$$end \leftarrow unchanged.latestMinor$$

[17]Input for the incremental update is described in detail in the section 3.3.

Once the subgraph is removed in the main graph, the merging process continues from this marked node. Hence, the new version of this removed subgraph is merged to the main graph. If the removed node or an edge is absent in the new revision, it will remain removed. If the removed node or an edge is present in the new revision, i.e. it will be in the input subgraph, it will be merged back by setting the minor end revision back to the maximum value .99. This way, we properly update in the main graph only the referring marked part of the input subgraph.

Let us demonstrate it on an example. Again, we will use the shorter, two-digit form of the minor revisions for the simplicity. Assume, in the Manta Flow server is a main graph containing two subgraphs with root nodes A and B. All vertices and edges were added during revision 1.00. No other revisions were committed yet. See the main graph in the figure 3.12.

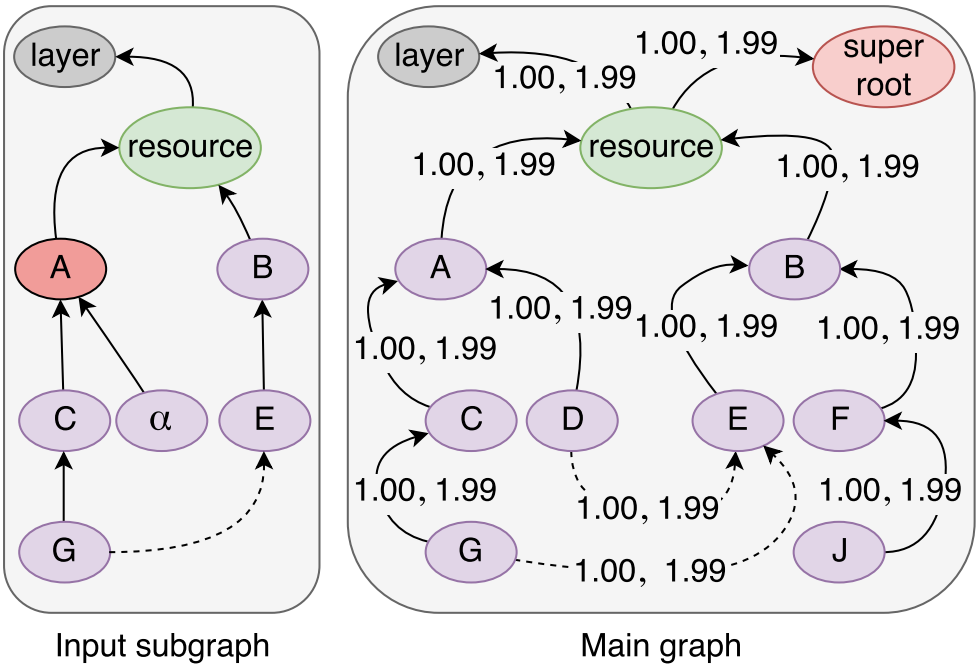


Figure 3.12: Subgraph merge - initial state

A small change was performed on user's side. As a consequence, in the subgraph with the root node A, a node D was removed and a new node α was added. However, it cannot be detected at this point, what exact nodes and edges were added or removed. The only available information is, a subgraph with the root node A was changed. Therefore, when creating an input for the incremental update, node A was marked (red color). See the input subgraph in the same figure 3.12.

Next, the input subgraph is sent to the Manta Flow server and the update to the revision 1.01 begins. First, resource and layer are merged. Since incre-

mental update is performed, we operate only on the level of minor revisions (decimal parts). Therefore, merging unchanged resource and layer will have no effect. We do not have to change neither minor nor major end revision to any specific value. See the figure 3.13.

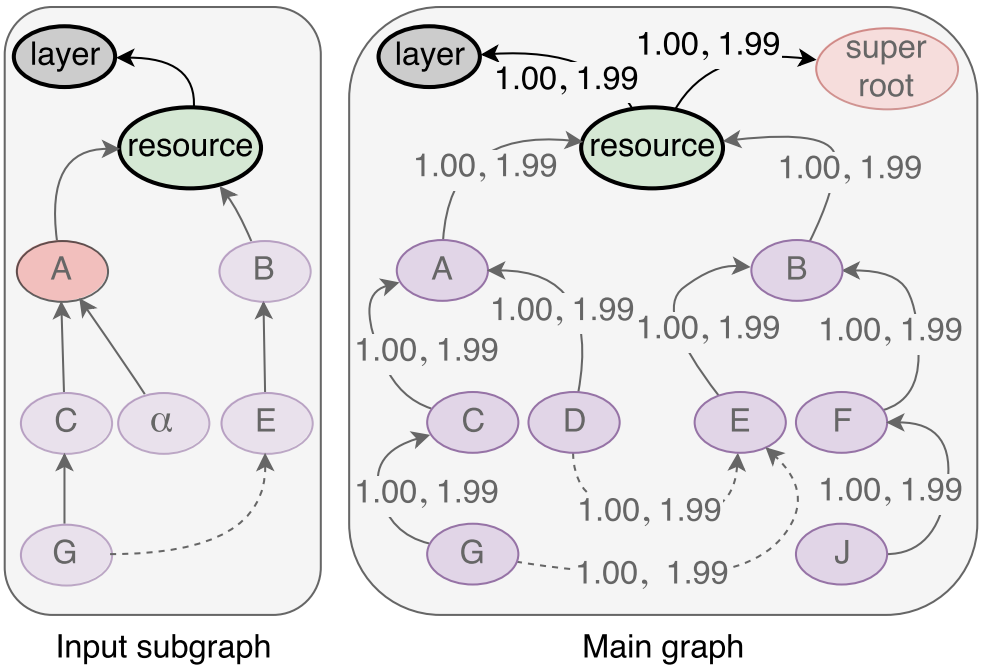


Figure 3.13: Merging resource and layer

Depending on the *topological order* of the input, either the subgraph with the root node A or the subgraph with the root node B is merged next. Without loss of generality, assume, the subgraph with the root node B is being merged next. That means, the node B and its child node E are being merged, in this order.

Again, merging unchanged nodes B and E is having no effect as well. Their start revisions (1.00) remain unchanged as well as their end revisions (1.99). Notice, in the input subgraph are missing child nodes F and J. These nodes have nothing in common with the change in the subgraph with root node A. Therefore, nodes F and J are missing in the input subgraph. Thus, they are simply skipped. See the figure 3.14.

Next, the node A is reached. Manta Flow detects, the node A is marked. Therefore, the whole subgraph of the root node A is removed. Latest revision is revision 1.00. Hence, all nodes and their data flows have their minor end revision parameters set to the latest minor revision .00. See the figure 3.15.

Notice, after the subgraph was removed (figure 3.15), we did not have to remove the root node A. Node A is already in the input. Thus, node A is, without doubts, present in the new revision and removing it would be

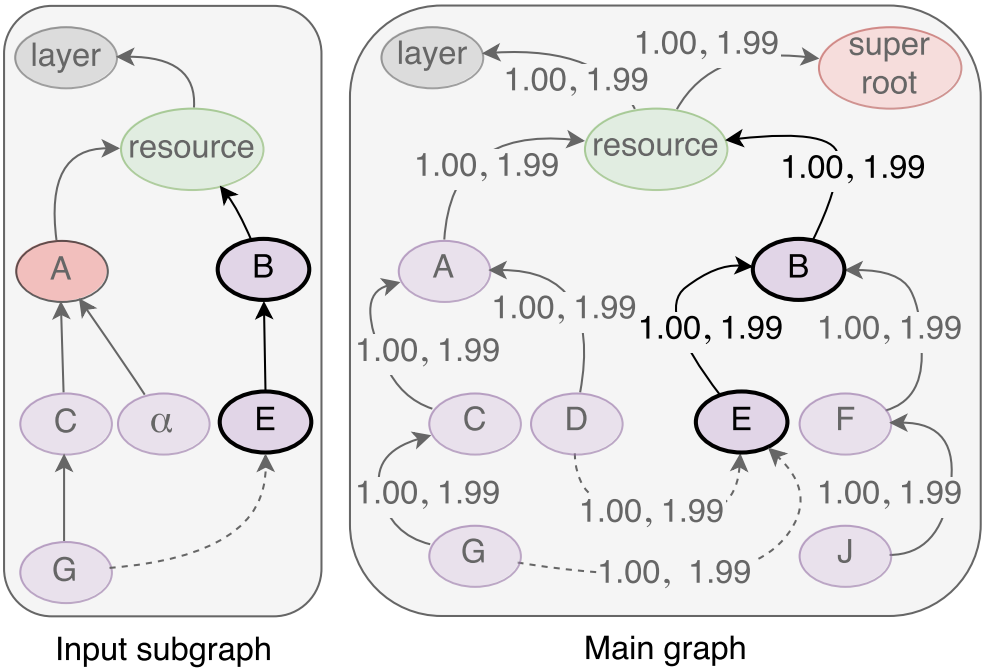


Figure 3.14: Merging nodes B and E

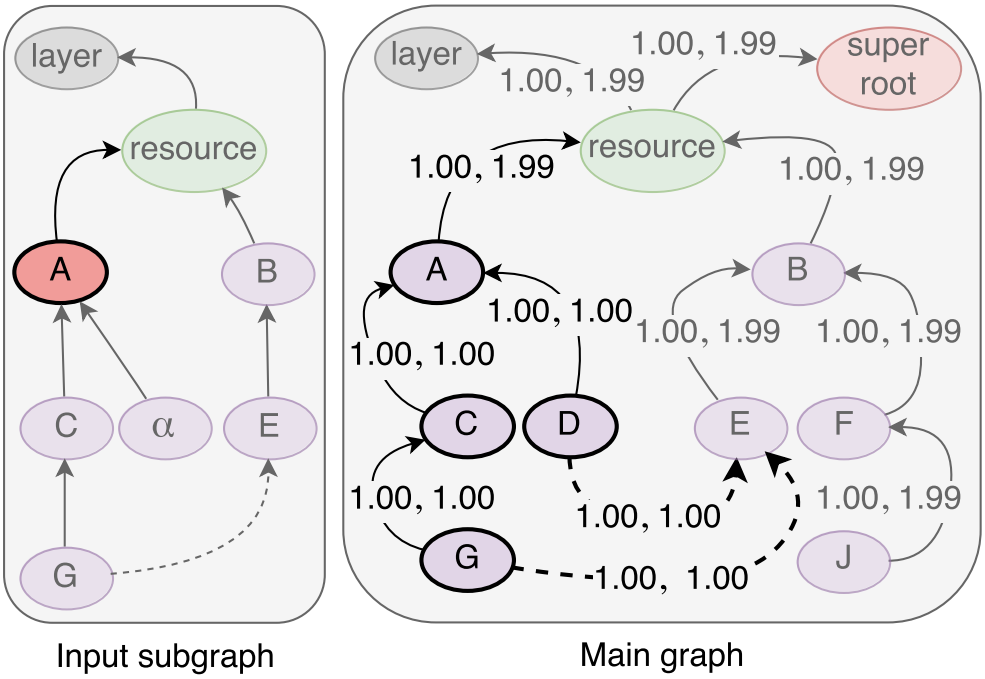


Figure 3.15: Removing subgraph with the root node A

meaningless. Nevertheless, node A may have some attribute nodes. If some attribute of the node A was absent in the new revision (i.e. the attribute was removed), it would be absent in the input subgraph as well. If we did not remove the node A and we also forgot to remove all attributes of the node A, the removed attribute would remain in the new revision. The database would be inconsistent. Node removal operation guarantees, all connected nodes and edges are removed as well. Therefore, if we decide to not remove the marked root node (as in this example), we have to make sure, all its connected nodes and data flows, including attribute nodes, were removed.

Once the subgraph is completely removed, merging process can continue. Nodes C, G and α are merged to the main graph. Nodes C and G were not changed. Therefore, their minor end revision parameters are set back to the maximum value .99. Node α is new. Hence, its start revision is set to 1.01 and end revision is set to 1.99. See the figure 3.16.

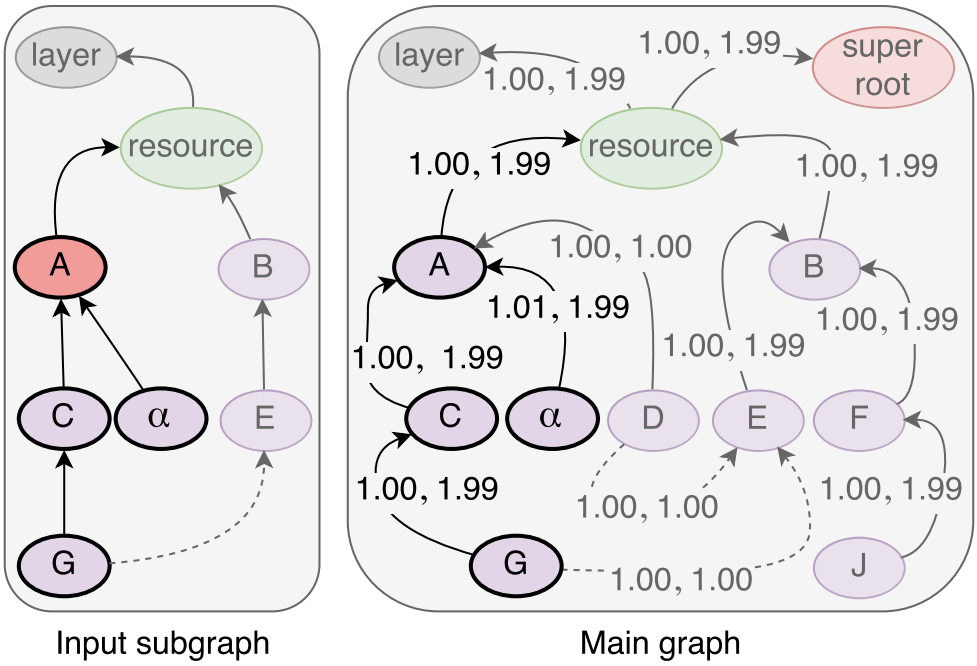


Figure 3.16: Merging subgraph with the root node A

Notice, when the merge is performed, we always check for the nodes and edges valid in the latest revision. If the merged node or edge has its end revision set as the latest revision and is also in the input, it is not newly created because it would lead into a redundancy. Instead, its end revision is set back to the maximum value .99.

As we can see in the figure 3.16, node D and its data flow to the node E remained removed in the new revision 1.01. This is the desired behavior, since the node D and its data flow to the node E are absent in the new revision.

We can also observe in the figure 3.16 (after the subgraph with the root node A was merged), the data flow from the node G to the node E is absent in the new revision 1.01, although, this data flow should be present in the revision 1.01. This is not a mistake. We only wanted to emphasize that **data flows are merged at the end**.

That means, first are merged all layers, resources and nodes. Hence, if we remove a marked subgraph during the traversal and then we continue in merging, the incoming and outcoming data flows to or from this subgraph are *not* merged together with nodes of this subgraph. In this case, data flow from node G to node E is not merged together with the nodes C, G and α.

Since we merged all nodes from the input subgraph, our last step is the merge of data flows from the input subgraph. In the input subgraph is only one data flow from the node G to the node E. This data flow is merged the same way as any other node. Its minor end revision is set back to .99. See the figure 3.17.

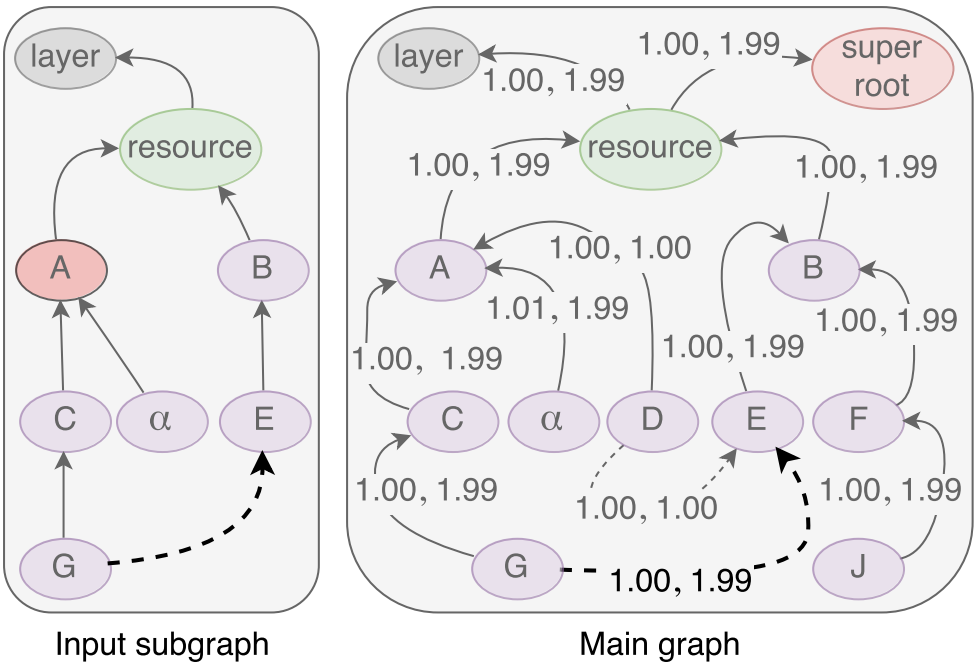


Figure 3.17: Merging data flow from the node G to the node E

Finally, we finished the whole merging process. All nodes and edges were successfully updated and the database now holds history of revisions 1.00 and 1.01.

The newly designed merge method is not the optimal method for the incremental update. The optimal method would traverse solely to the changed nodes and edges and would change only their minor revision data. However, the inability of producing the optimal input for the update limits us in imple-

menting the optimal method.

We are only able to know what subgraph has changed. This limited knowledge makes us to first remove the old revision of the changed subraph and then merge the new revision of this changed subgraph.

At first glance, it might seem, a more effective solution would be not removing the whole graph but instead, finding the difference between the old and new revision of the changed subgraph. However, this would leads us back to the problem of finding differences between two graphs. This problem can be converted to the well-known graph isomorphism problem, which belongs to the NP complexity class. On the other hand, traversal through all nodes and edges of the old and new subgraph has obviously a linear complexity. Therefore, we decided to first remove the marked subgraph (the old version) and then merge in the new version of this subgraph.

Time complexity of this method is directly proportional to the number of nodes and edges from the input subgraph (including the marked subgraph) plus the number of nodes and edges of the old version of subgraph in the main graph.

Presuming, only small changes will be performed[18], time complexity of this merge method is approaching time complexity of the optimal incremental update method.

3.3 Input

As we already know, the new method of the incremental update requires a new type of input. Since no changes need to be performed in the input for the full update, we will focus exclusively on the **input for the incremental update**.

Input for the incremental update is a **subgraph** representing changes in the new revision. However, the input subgraph does not *consist only of* the changed nodes and edges. The input subgraph only *contains* the changed data. For instance, a new line of code was added into a script. Then, we do not receive on the input only the nodes and edges representing the newly added line. We receive a graph structure of the whole script including the newly added nodes and edges.

Next, each node in the input subgraph has to have unambiguously defined a **full path to its resource**. We need to know, where exactly each node belongs to. We cannot send on the input a subgraph consisting only of the nodes and edges without their context. For instance, we want to perform an incremental update of a changed procedure. Then, we have to add to the input subgraph representing the changed procedure also a schema, database, resource(s) and layer(s), where this procedure belongs to. If the procedure affects another objects by its data flows, such as columns, we also have to

[18] According to Lukáš Hermann, number of the changed nodes and edges in the incremental update will be only a fraction of all the nodes and edges present in the main graph.

include parents of these objects up to their resource(s) and layer(s). In other words, we have to be able to find out in the input subgraph for every node its resource.

Since we receive on the input for the incremental update a subgraph containing not only the changed nodes and edges, we have to **indicate what has changed**. We already know, marking only the added or removed nodes and edges is currently impossible. We can use for the indication of changes in the input subgraph only information from the newer version of the changed object (e.g. a procedure or a script), without knowing what exactly has changed.

Therefore, we will **mark a root node** of a subgraph in the input subgraph that definitely contains the change. We do not have to mark all the nodes and edges in the changed subgraph. In general, when a node is changed, all its successors and their edges are inherently changed as well. Therefore, we need to mark only a root node of the changed subgraph.

We will most probably mark some unchanged nodes and edges. This will slightly increase complexity of the update, since we will probably update something unchanged. However, we have no other option due to the **inability of creating a graph delta** (explained in the previous chapter, in the section 2.4.

Let us demonstrate all the mentioned information from above on an example. Assume, we have a procedure assigning an input parameter into an output parameter. This procedure is enriched by another line of code, insertion of the input parameter into the LOG table. See the listing 3.1.

```
PROCEDURE proc(IN I, OUT O) {
        O = I;
        INSERT INTO LOG(MES) VALUES(I);
}
```

Listing 3.1: Procedure - insert statement added

Next, user wants to perform an incremental update of this small change. Therefore, an input subgraph representing this change had to be created first.

The input for the incremental update will be created the same way as the input for the full update. First, the extraction of user's data is performed. Second, the data flow analysis of the extracted and other provided data is performed. Finally, the resulting metadata are exported in the form of a linearized graph. The only difference is, the provided data will be only the changed data. Hence, the input subraph will contain mostly the changed part. See the figure 3.18 for the graphic illustration of such an input.

As we can see in the figure 3.18, the input subgraph meets all the requirements defined above. On the input is not only the change of the procedure (i.e. nodes INSERT and MES and their edges) but there is the whole procedure. What more, there is not only the procedure but also their parents up to

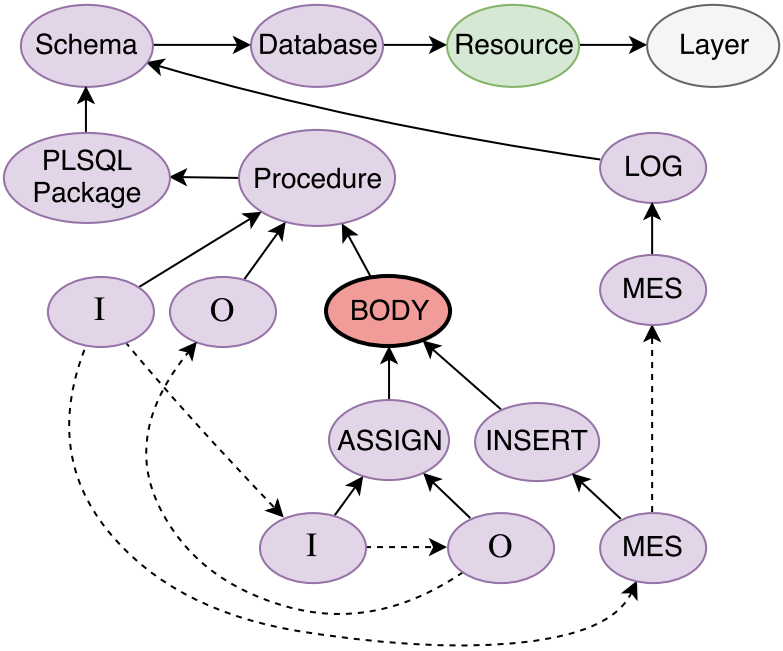


Figure 3.18: Input subgraph

the root. Each node has one and only one full path to its resource. Thus, each node is unambiguously identifiable. Finally, the changed part of the subgraph is represented by the red nodes and edges. Since we only know the procedure has changed but we cannot closely specify what nodes and edges were added or removed, we simply marked the whole procedure body subgraph, starting from the node BODY.

3.4 Customization

The newly designed incremental update requires special modifications to fit all the specifics of Manta Flow. These modifications reflect the expected use cases and the specificity of the data Manta Flow works with, in particular, the data flows.

Also, the expected usage of Manta Flow should be respected in the new design. Especially, to optimize querying in the graph database. We will also cover all these modifications in the following section.

3.4.1 Inconsistency

When designing the input for the incremental update, you probably wondered, why we did not mark in the example of the changed procedure (figure 3.18) the whole procedure subgraph, i.e. why we did not mark the node Procedure.

If we marked the node Procedure, we would remove all nodes and edges during the merge, including the incoming and outcoming **data flows**. However, it can easily happen, we do not receive on the input all data flows we removed during the merge. Hence, this operation can bring **inconsistency** to the database. We will demonstrate it in the following example.

Assume, we have the old version of the procedure from the example above. This time, we know, this procedure is called by another script. User has this procedure and script only in the form of files. See the listings 3.2 and 3.3.

```
PROCEDURE proc(IN I, OUT O) {
        O = I;
}
```

Listing 3.2: Procedure

```
BEGIN
        proc(I, O);
END;
```

Listing 3.3: Script

Procedure and script are also stored in the graph database in Manta Flow. Since there is a dependency between the script and procedure, the appropriate data flows were created between them. Assume, the graph database has only one revision 1.00 and both the script and the procedure are valid in this revision. See the snapshot of this procedure and script from the revision 1.00 in the figure 3.19.

As we can see in the figure 3.19, there two data flows between the script and the procedure. One data flow leads from the script parameter I to the procedure input parameter I. Another data flow leads from the procedure output parameter O to the script parameter O. Hence, these data flows refer to the value passing between the script and the procedure.

Next, user renames in the procedure the input parameter I to J and the output parameter O to P. See the listing 3.4.

```
PROCEDURE proc(IN J, OUT P) {
        P = J;
}
```

Listing 3.4: Procedure - renamed parameters

Since renaming parameters in one procedure is a small change, user decides to perform an incremental update to propagate this change to the graph database.

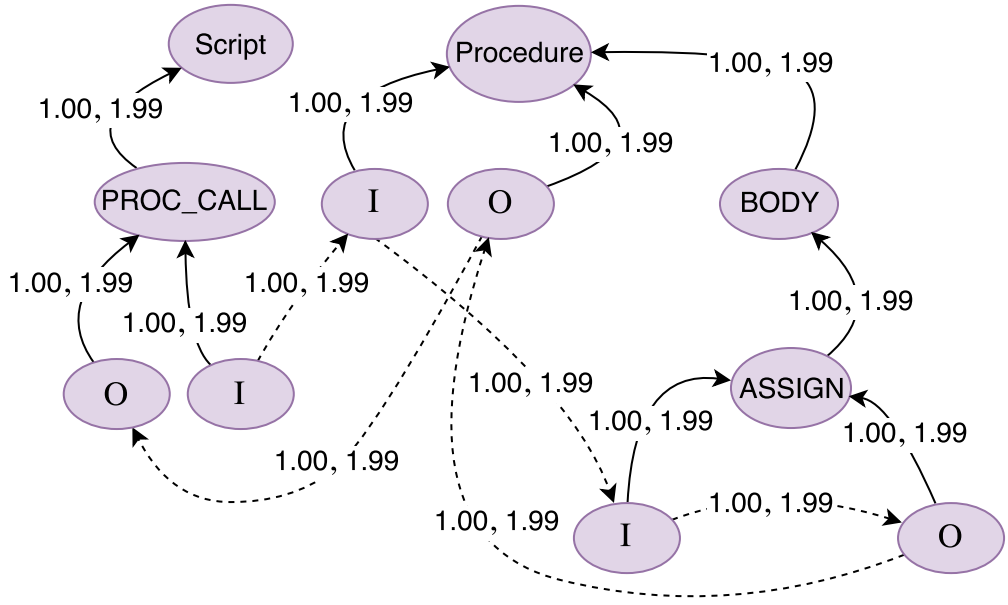


Figure 3.19: Script and procedure in the main graph

First, the input has to be created. Because we know, only the procedure was changed, we create the input subgraph based only on the analysis of the procedure. Notice, the script content has not changed at all. Therefore, we assume, there is no need in the analysis of the script. See the input in the figure 3.20.

This time, we had to mark the whole procedure subgraph, starting from the node Procedure. If we did not mark the whole procedure subgraph, we would ignore the fact that the parameters I and O were renamed also in the procedure declaration.

The input is prepared. Thus, the incremental update may begin. Server starts merging the input subgraph to the main graph. Once the node Procedure is reached, the whole subgraph, starting from the node Procedure, is removed. The data flows between the script and the procedure are removed as well.

After the procedure subgraph is removed in the main graph, server proceeds in merging the new procedure subraph from the input. When all the nodes and attributes from the input are merged to the main graph, server completes the whole process by merging the data flows from the input to the main graph. See the result of the incremental update in the figure 3.21.

We can clearly see in the figure 3.21 that the data flows between the script and the procedure are missing in the new revision 1.01. This is caused by not knowing about the call of the changed procedure from the script.

The issue described above is a problem of **the interface change**. When

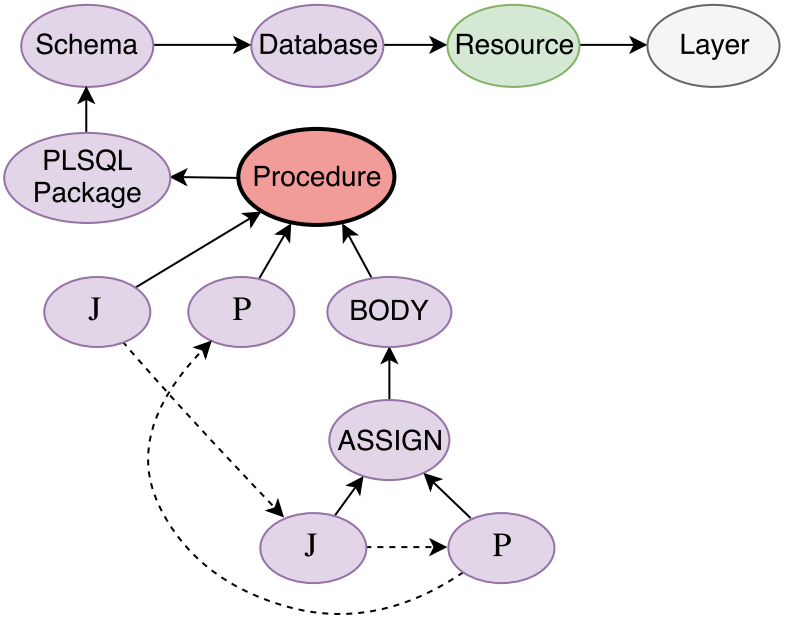


Figure 3.20: Input subgraph of the changed procedure

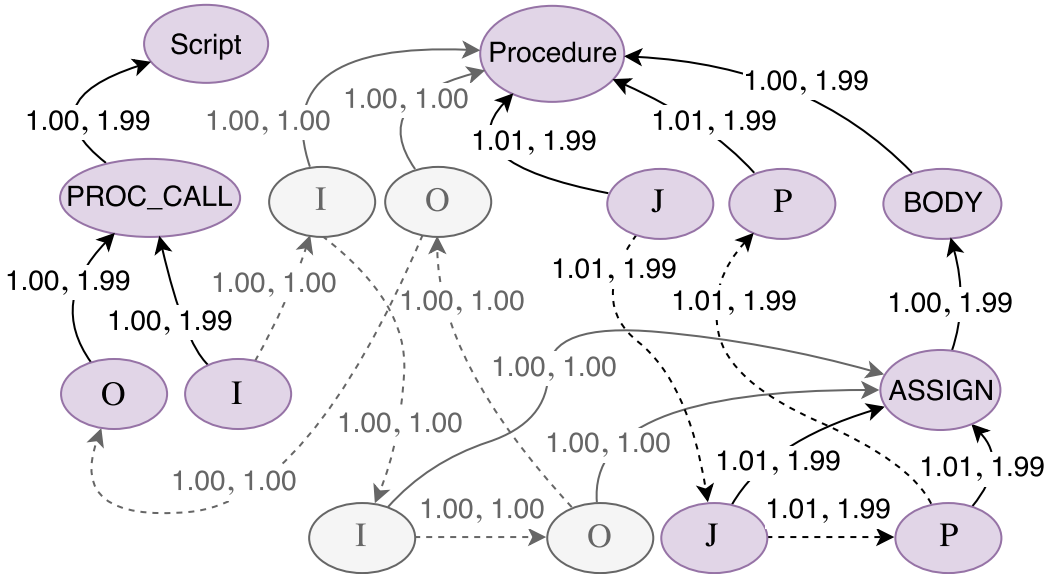


Figure 3.21: Script and procedure in the main graph after the merge

an interface is changed, there is a risk of bringing inconsistency to the system. In this case, we lost the information about data flows. We did not respect the potential **dependencies** of the changed objects.

If we wanted to perform a proper incremental update, we would have to search for all occurrences of the changed object and create an input subgraph

extended of the missing data flows. We would have to perform an **impact analysis** to cover all the dependencies of the changed database object.

We do not have currently a tool for the impact analysis. And even if we had such a tool, it could take a lot more time since the analysis would have to search through all the user's data to make sure no dependencies were ignored. Nevertheless, this is just a rough observation. A deep analysis about the usage of impact analysis in Manta Flow should be performed before making any further statements about its time complexity.

For now, we will have to deal with the incompleteness of incremental update in Manta Flow by specifying individual rules and constraints for specific cases.

One approach is not to allow users to perform incremental update when there is a dependency. For instance, when a *procedure declaration* was changed or when a script, upon which some other scripts are *dependent*, is changed.

Consequentially, we will have to customize input for the incremental update depending on the type of the changed object. For example, when a procedure is changed, only the procedure body will be marked as changed, since we presume, only the body of the procedure is allowed to be changed.

Another approach is to allow users perform incremental despite a dependency change, but a **data flow restoration** has to be performed. A suitable usage for such a case is a *column addition*. User adds a column to a table but we do not know what column that was, we only know what table has changed. Hence, we will mark the whole table node in the input subgraph. Once this node is reached during the merging process, the table subgraph is removed. As we know, this subgraph removal will remove all children nodes also with their data flows. That means, all data flows to or from the columns will be removed. However, when the table with all its columns is merged back, we will additionally not only update end revision of the column nodes, but we will update end revisions of their data flows from the latest revision as well. Hence, we will restore data flows of the columns still present in the new revision.

Clearly, data flow restoration may produce the opposite side effect. No data flows will be lost but some data flows may be accidentally added to the new revision. For instance, some script was changed and consequentially, a data flow from a column should no longer exist in the new revision. Simultaneously, in the same table was created a new column, hence, a new column node should be present in the new revision. Assume, these two changes, script change and column addition, are updated together within one revision. If the change of the script is updated first and the column addition is updated afterwards, the removed data flow remains in the new revision.

In conclusion, users have to be aware of the risk of bringing inconsistency to the database when an incremental update is performed. Anyway, users will be allowed to perform incremental update at their own risk, but they need to be either sure their change is independent on the other objects so no

inconsistency may occur or they accept the risk of bringing inconsistency to the database in the exchange for a fast update.

3.4.2 Object creation and removal

When designing the incremental update method, we considered only an object change but not its **entire creation or removal**. For instance, we studied only a change of the procedure body or procedure interface. Nevertheless, we did not consider such cases, when a new procedure is created or removed.

Object creation and its subsequent propagation to the database **does not require any modification** of the update input or of the update method. Merging a newly created object can follow the same algorithm as if only a part of the object was changed.

First, the analysis of the newly created object is performed and the input is created. During the input creation, the appropriate part of the newly created object is marked as if it was changed. Then, the incremental update merge method is performed. Due to the logic of the merge algorithm, the whole input is merged to the database, including the unmarked part of the object. Since the input contains the whole object, it is guaranteed, the whole object is stored in the database.

For instance, a whole new procedure is created and the incremental update is to be performed. The input for the incremental update contains the whole procedure structure, including the procedure interface. Also, the procedure body node in the input subgraph is marked as changed. During the merge, the interface is newly added to the main graph. When the marked procedure body node is reached, the whole procedure body subgraph is first removed from the main graph. However, there is no procedure body in the main graph yet. Therefore, there is nothing to remove and the merging can immediately continue. When the merge is finished, the whole procedure is in the main graph.

Object removal and its subsequent propagation in the database may be performed in two ways.

First option would be to follow the same algorithm as if the object was only changed. We would create an input subgraph containing the removed object and put a special mark on the root node of the removed object. Then, this input subgraph would be merged in the incremental mode to the main graph. Once the specially marked root node would be reached during the merge, it would be removed together with all its successors and their edges.

The problem is, server must not merge back the removed subgraph. Hence, either a special mark has to placed on all the nodes and edges of the removed subgraph so the server did not merge them back or during the input creation only the root node would be created.

Second option would be to simply send to the server a **path to the root node** of the subgraph to be removed and server would only find this node and

then remove it together with all its successors and their edges. For instance, we would send to the server a concrete sequence $Resource \rightarrow Database \rightarrow Schema \rightarrow PLSQLPackage \rightarrow Procedure$, server would find it and remove the whole subgraph starting from the node Procedure.

Sending only a path to a concrete node is definitely faster than creating a special input subgraph. Also, there is no need in merging the input subraph. The input subgraph may include not only the subgraph to be removed and its parent nodes but it may also include other affected nodes and their parent nodes. If the procedure affected other columns and tables, we would have to walk through these as well. It may be significantly faster to traverse only to the root node of the subgraph we want to remove. On top of that, we would also have put an extra indication that the marked root node should not be merged back so we avoided the so called zombie node.

Considering all the advantages of sending only a path to root node of the removed object, we will implement an **extra method of the incremental update** only for the case of object removal.

3.4.3 Overflow

Some users may perform frequent incremental updates and never perform a full update. For example, some users perform every day multiple of small changes that they want to immediately propagate to Manta Flow to see the impact of these changes on data flows. In this case, there is a high risk, the maximum number of minor revisions (one million) may be insufficient and once this maximum is reached, an overflow may happen.

On top of that, some users may be unable to quickly generate input for the full update and they may be limited only by the incremental update. Hence, they cannot perform full update to automatically transition to the new major revision.

To prevent this situation, we have to perform a **manual transition** to a new major revision to start a new series of minor revisions.

The idea is simple. We will increment major end revision of all nodes and edges valid in the latest revision to the new major revision, before we start merging the input subgraph. Let us demonstrate it on an example.

Assume, we have major revision 1, we can have exactly one hundred minor revisions at maximum. We have already performed one hundred incremental updates. That means, we have committed revisions $1.00, 1.01, \ldots, 1.98, 1.99$.

Next, user wants to perform another incremental update but we cannot create revision 1.100. Therefore, we create a new revision 2.00 and we update end revision of all nodes and edges valid in the latest revision (1.99) to 2.99.

Following is the standard merging process. When it comes to a subgraph removal of the changed subgraph, we have to set the end revision of the removed nodes and edges to the latest revision, revision 1.99. When merging the new subgraph back, we also have to search for the nodes and edges valid in

the latest revision 1.99. If the node or edge is still present in the new revision 2.00, then its end revision is set back to 2.99.

Obviously, the logic of incremental update remains the same, although, we have to manipulate not only with minor part but also with major part of end revisions of the affected objects.

Unfortunately, transition to the new major revision is asymptotically comparable with the full update, since we have to visit every node and edge valid in the latest revision and set its end revision to the new major revision. Nevertheless, it is an inevitable operation to keep the database consistent in case the maximum number of minor revisions is reached and it is impossible to perform a full update.

3.5 Summary

The fundamental idea of the newly designed method of incremental update is based on the principle of **major and minor revisions**. This two-level revision system allows effectively perform both full update and incremental update over the same graph data model.

The new concept of two-level revision system led to the series of necessary changes in the current design of version control in Manta Flow. Following are the most important changes:

- Changed data type of revision number
- New merge method for the incremental update
- New input for the incremental update

Properties tranStart and tranEnd have their data type changed from Integer to Double. Double data type allows effective usage of both major and minor revisions. The integer part represents major part of the revision, the decimal part represents the minor part of the revision.

Important difference between the major and minor part is in the usage of **closed and unclosed end revisions**. Major part is using the system of closed end revisions, where the latest valid revision number is always explicitly specified. Minor part is using the unclosed end revisions, where the latest valid revision number can be implicitly specified by the highest possible minor revision number. Thanks to the indirectly specified latest minor revision number, only a part of the graph can be updated without the need of updating the rest of the whole database graph.

The highest possible minor revision number is .999999. This is due to the fact, Titan graph database stores Double data type values only up to six decimal digits. Hence, the maximum number of minor revisions within one major revision is one million.

When the maximum number of minor revisions is reached, we have to perform a **transition to the new major revision** in order to avoid the overflow. Either a full update is performed, which naturally performs transition to the next major revision. Or an incremental update is performed, during which the transition to the next major revision is performed before the merging phase starts. The merge process then operates within the new major revision, preceding the issue of overflow.

The **new merge method** of the incremental update is a modified merge method of the full update. This new merge algorithm merges nodes and edges from the input the same way as the full update does. The only difference is, when a specially marked node is reached, its whole **subgraph is removed**, including all the incoming and outcoming data flows to or from this subgraph. Then, the new version of the subgraph is merged back. Since the marked subgraph represents the change and the old version is removed and the new version is merged back, the changed graph is successfully saved in the graph.

This new merge method is performed from **top to bottom**. Although, applying the merge method in some other systems from bottom to top may bring an advantage, the specificity of the graph data in Manta Flow disallows to perform merging from bottom to top. The input subgraph is always in a topological order, starting from the root node. Moreover, traversing in the graph database is also performed from top to bottom. Therefore, it would be unnatural and **ineffective to perform the merge method from bottom to top**.

Simultaneously with the design of a new merge method, a suitable **input for the incremental update** had to be designed as well. The input for the incremental update is a **subgraph containing the changed data**.

However, **input for the incremental update** cannot be easily created for every change performed. Some changes have a further, currently undetectable, impact. When these changes are propagated via the newly designed incremental update method, an **inconsistency** may be brought to the database. Therefore, users are allowed to incrementally update only some types of changes and some special cases can be dealt with individually.

Currently, the problem of potential inconsistency can be solved only by performing a full update, where the new revision contains certainly no inconsistency. In the future, this problem can be solved by the application of **impact analysis** on the performed change followed by the creation of a proper input for the incremental update.

Nevertheless, some users may prefer the risk of bringing a small inconsistency to the database in an exchange for a fast update. Therefore, it is acceptable to implement this method of incremental update.

Furthermore, when studying the new merge method and input for the incremental update, it was observed that in case of a **whole data structure removal** is more effective to simply perform a **subgraph removal** instead

of performing a merge to the database. Therefore, an extra method of incremental update will be created for the case of a whole data structure removal.

Finally, **querying** to the database should show the similar performance as in the current Manta Flow version. We will use the same index structure, allowing a fast graph traversal. We will use in the index structure the same property keys tranStart and tranEnd. The only difference is in the switching from the Integer data type to the Double data type.

In conclusion, based on the analysis and design from this chapter, the incremental update in Manta Flow should significantly speed up propagation of small changes to the graph database. Furthermore, the update time complexity should be directly proportional to the change size and the querying time complexity in the graph database should show the similar time complexity as in the current implementation.

Chapter 4

Implementation

Based on the design from the previous chapter, a prototype supporting both incremental update and the already existing full updated was implemented. This prototype was implemented as a branch, based on the latest Manta Flow version 1.20.

Technologies used for the prototype implementation are the same technologies used in the current Manta Flow product. That means, the prototype is running on Java platform (version 1.7), using the Spring framework and the graph database used is the well known graph database Titan from the first chapter (version 0.4.4).

In the following text, we will describe changes performed in the prototype branch on both logical parts of Manta Flow, on Manta Flow client and Manta Flow server.

> **Shortened source codes**
>
> Source codes showed in this chapter are shortened and simplified to reduce unnecessary lines of code for description and to save space. For original source codes see directory `src` in the attachment to this thesis.

4.1 Manta Flow server

Graph database (the metadata repository) is located on server. Therefore, main logic of the graph database update is managed on server side. Thus, most of the changes were performed on the Manta Flow server.

The most important changes were made in the modules Core, Connector and Merger. Additionally, some minor changes had to be performed also in other modules such as Exporter, Dump or Viewer. However, we will describe only changes in Core, Connector and Merger.

4.1.1 Core

Module Core defines data model of the graph database in Manta Flow and other core data structures used in Manta Flow server.

According to the new design of revision tree (section 3.1.5), in class `DatabaseStructure` were created **new vertex properties**:

- latestCommittedRevision in revision root
- latestUncommittedRevision in revision root
- previousRevision in revision node
- nextRevision in revision node

Additionally, in class `DataflowObjectsFormat`, which is defining format of vertices and edges in the CSV file sent from client to server, was extended node format to support new optional node property **FLAG**.

Apart from these structural changes no other changes were made in this module.

4.1.2 Connector

Module Connector encapsulates graph database and provides interface for basic operations over this graph database. Hence, a new revision logic and other graph operations were implemented in this module.

The most important changes were made in classes `IndexCreator`, `GraphCreation`, `GraphOperation` and `RevisionRootHandler`.

IndexCreator class creates indices. The only change made in this class is changing index data type of edge properties tranStart and tranEnd from Integer to Double.

Revision number data type has been changed from Integer to Double also in all other classes and modules working with revisions. Nevertheless, it will not be mentioned explicitly any more, if not important, since logic of working with revision numbers is still the same.

GraphCreation class is responsible for creation of vertices and edges in the database. In this class were performed these changes:

- Revision end (tranEnd property) is set to the maximum minor revision
- Set previous and next revision nodes of the newly created revision node and the revision root

According to the design from the previous chapter (section 3.1.4, every created edge or vertex has its end revision newly set to the maximum minor revision in its major revision. The maximum minor revision is revision with revision number "major.999999".

Due to the new design of revision tree (section 3.1.5), every revision node has saved in its properties numbers of its previous and next revision nodes and revision root has saved in its properties numbers of the latest committed and uncommitted revision number. Thus, when creating a new revision node, these properties have to be set as well. See listing 4.1.

```
1  public static Vertex createRevisionNode(Vertex revisionRoot,
       Double revNumber, Vertex prevRevNode, Vertex nextRevNode)
       {
2      ... // some more stuff
3      // set previous revision number
4      if(prevRevNode == null) {
5          newRevisionNode.setProperty("prevRevNumber", -1.0);
6      } else {
7          newRevisionNode.setProperty("prevRevNumber",
               prevRevNode.getProperty("revNumber"));
8          prevRevNode.setProperty("nextRevNumber", revNumber);
9      }
10
11     // set next revision number
12     if(nextRevNode == null) {
13         newRevNode.setProperty("nextRevNumber", -1.0);
14     } else {
15         newRevNode.setProperty("prevRevNumber", nextRevNode.
               getProperty("revNumber"));
16         nextRev.setProperty("prevRevNumber", revNumber);
17     }
18
19     // set revision root
20     if(isCommitted) {
21         revisionRoot.setProperty("latestCommRevNumber",
               revNumber);
22     } else {
23         revisionRoot.setProperty("latestUncommRevNumber",
               revNumber);
24     }
25 }
```

Listing 4.1: Method createRevisionNode

GraphOperation is a class providing graph operations in database. In this class were created three methods:

- `setSubtreeTransactionEnd`
- `deleteSubtree`
- `performTransition`

Method **setSubtreeTransactionEnd** sets transaction end of a subtree. This method is invoked during incremental update when a node has a special flag indicating that its whole subtree has to be "removed" before merging proceeds (see section 3.2.2).

In this method, all vertices in the subtree (and their edges) valid in a specific (walk-through) revision have their end revision (tranEnd) set to a new value (newTranEnd). Vertices and edges absent in the specified (walk-through) revision are ignored. Subtree is specified by its root vertex and the traversal is performed over the `hasParent` edges. See listing 4.2.

```
1  public static void setSubtreeTranEnd(Vertex rootVertex,
       Double walkthroughRev, Double newTranEnd) {
2      Edge controlEdge = getControlEdge(rootVertex);
3      Double tranStart = controlEdge.getProperty("tranStart");
4
5      if (newTranEnd < tranStart) {
6          deleteSubtree(rootVertex);
7          return;
8      }
9
10     // recursively repeat for all direct children nodes
11     Iterable<Vertex> children = getDirectChildren(rootVertex,
           walkthroughRev);
12     for (Vertex childNode : children) {
13         setSubtreeTranEnd(childNode, walkthroughRev,
               newTranEnd);
14     }
15
16     // set tranEnd of all the rootNode's edges
17     Iterable<Edge> edges = getAdjacentEdges(rootVertex,
           Direction.BOTH, walkthroughRev);
18     for (Edge edge : edges) {
19         Double edgeTranStart = edge.getProperty("tranStart");
20         if (newTranEnd < edgeTranStart) {
21             if (edge.getLabel().equals("hasAttribute")) {
22                 // delete node attribute
23                 Vertex nodeAttribute = edge.getVertex(
                       Direction.IN);
24                 nodeAttribute.remove();
25             }
26             edge.remove();
27             continue;
28         }
29         setEdgeTransactionEnd(edge, newTranEnd);
30     }
31 }
```

Listing 4.2: Method setSubtreeTransactionEnd

Notice, when the new end revision (newTranEnd) is less than the start revision (tranStart), the edge or vertex is removed[19]. When performing update, new revision has always the highest revision number and this case never happens (newTranEnd is always greater than any tranStart).

However, when performing update in a system *without* version control (there is one and only one revision `0.0`, no new revision is ever created), then the same method can be used for the incremental update.

Also notice, when a `hasAttribute` edge should be removed, first its target vertex (node attribute) has to be removed first. Otherwise, the target node attribute would be unreachable.

Method **deleteSubtree** deletes all vertices and edges regardless their revision validity. This method performs the same traversal as method `setSubtreeTranEnd`. The only difference is, it simply removes all nodes and vertices without setting saving the history. As we already know, this method is used when performing incremental update in a system without version control.

```
1  public static void deleteSubtree(Vertex rootNode) {
2      Iterable<Vertex> children = getDirectChildren(rootNode, "
           EVERY_REVISION_INTERVAL");
3      for (Vertex childNode : children) {
4          deleteSubtree(childNode);
5      }
6
7      Iterable<Edge> edges = getAdjacentEdges(rootNode,
           Direction.BOTH, "EVERY_REVISION_INTERVAL");
8      for (Edge edge : edges) {
9          if(edge.getLabel().equals("hasAttribute")) {
10             Vertex nodeAttribute = edge.getVertex(Direction.
                   IN);
11             nodeAttribute.remove();
12         }
13         edge.remove();
14     }
15     rootNode.remove();
16 }
```

Listing 4.3: Method deleteSubtree

The third implemented method **performTransition** performs transition from an old revision to a new revision in the whole repository. That means, all nodes and edges in the *main graph* and in the *source code graph* have their end revision (oldRev) set to a new end revision (newRev). See listing 4.4.

[19] In case of vertex its whole subtree is removed, see method `deleteSubtree`, listing 4.3

```
1 public static void performTransition(Double oldRev, Double
      newRev, TitanTransaction transaction) {
2     Vertex superRoot = getRoot(transaction);
3     Vertex sourceRoot = getRoot(transaction);
4     setSubtreeTransactionEnd(superRoot, oldRev, newRev);
5     setSubtreeTransactionEnd(sourceRoot, oldRev, newRev);
6 }
```

Listing 4.4: Method performTransition

RevisionRootHandler is a class responsible for management of revision tree. In this class were created new methods `createMajorRevision` and `createMinorRevision`. These methods replaced the original method `createRevision`, to clearly distinguish what type of next revision will be created.

Both of these methods create a new revision node. The only difference is the way these methods generate next revision number. Method `createMinorRevision` adds `0.000001` to the latest revision and method `createMajorRevision` uses next integer number.

Additionally, method `createMinorRevision` checks for the potential overflow. Thus, when a maximum minor revision is reached and new minor revision creation is requested, a transition to the next major revision context has to be performed first (see listing 4.4. After the transition is performed, incremental update process proceeds unchanged.

Moreover, to ensure correct behavior, class `BigDecimal` is used for operations with floating point. For instance, when adding `0.000001` to another decimal number when generating next minor revision number, `BigDecimal` is used. See an example in help method `getNextMinorRevNumber` (listing 4.5).

```
1 /** Maximum scale up to 6 decimal digits */
2 Integer SCALE = 6;
3 /** Rounding down when cutting rest of digits */
4 RoundingMode ROUNDING_MODE = RoundingMode.DOWN;
5 /** The increment value for a next minor revision */
6 BigDecimal MINOR_REVISION_DELTA = new BigDecimal("0.000001").
      setScale(SCALE, ROUNDING_MODE);
7
8 private Double getNextMinorRevNumber(Double
      latestCommRevNumber, Double latestUncommRevNumber) {
9     ...
10    BigDecimal latestCommRevBD = BigDecimal.valueOf(
          latestCommRevNumber).setScale(SCALE, ROUNDING_MODE);
11    return latestCommRevBD.add(MINOR_REVISION_DELTA).
          doubleValue();
12 }
```

Listing 4.5: Method getNextMinorRevision

4.1.3 Merger

Merger Server Logic[20] provides logic for advanced operations such as contraction, unification, interpolation, merging, edge propagation, view back links creation, prune revisions, rollback revisions and others.

Some of these operations required minor changes due to the change of revision logic. However, there were performed only two important changes in these classes:

- `StandardMergerProcessor` (merging logic)
- `OneFileDeleter` and its private class `ProcessPathTransactionCallback` (delete logic)

StandardMergerProcessor is responsible for merging objects (vertices and edges) to the metadata repository. It simply receives one vertex or edge (together with its **context**, so it knows exactly *where* to merge the object) and merges it to the graph.

According to the analysis and design, the only difference from merging in full update is that some nodes may have additional **flag** indicating a special operation has to be performed (see section 3.2.2). Hence in the method `processNode` were added checks for these flags. See listing 4.6.

```
1  protected ResultType processNode(String[] itemParameters,
       ProcessorContext context) {
2      // get node's flag (if any)
3      flag = itemParameters[NodeFormat.FLAG.getIndex()];
4      // merging revision number
5      Double newRev = context.getRevision();
6      // revision number "major.999999"
7      Double newEndRev = getMaxMinorRevNumber(newRev);
8      // latest committed revision
9      Double latestComRev = getLatestCommRevNumber();
10     ...
11     if (flag.equals("remove_myself")) {
12         // remove node's subtree (including this node)
13         setSubtreeTranEnd(node, latestComRev, latestComRev);
14     } else if(flag.equals("remove") {
15         // remove node's subtree (including this node)
16         setSubtreeTranEnd(node, latestComRev, latestComRev);
17         // merge this node back
18         setVertexTranEnd(node, newEndRev);
```

[20]Merger module is obtained in both client and server parts of Manta Flow. It consists of multiple smaller logical units such as such as Model, Server Logic, Client, Server Service or Server Titan. Changes were performed only in **Merger Client** and **Merger Server Logic**. Merger Client will be explained in the context of Manta Flow Client (section 4.2). Thus, in this section will be examined changes only in Merger Server Logic.

```
19      } else {
20          // no flag, only merge node (set its tranEnd)
21          setVertexTranEnd(node, newEndRev);
22      }
23      ...
24  }
```

Listing 4.6: Method processNode

Notice that thanks to the usage of flags, incremental update is using the same merging processor as full update. No other important changes were performed in Merger module.

Additionally, in Merger Server Logic was created a new logic for **deleting vertices specified by their path**. For this purpose was created a `delete` package. The essential class in this package is class `OneFileDeleter`.

OneFileDeleter is responsible for deleting all vertices specified in an input file. In this file each line represents path to a target to be deleted. Each line is in format:

```
name/name/name/.../targetName[,type/type/type/.../targetType]
```

Thanks to this format it is possible to remove every node specified by its parent nodes. Additionally, to avoid ambiguity, node types can be added as well.

```
Oracle/ORCL/METADATA/DF_NODE
Oracle/ORCL/METADATA/removeSpecChars,*/*/*/Function
Teradata/db/table2/t2c1,Teradata/Database/Table/Column
```

Listing 4.7: Input file for deleting single vertices

For instance, in listing 4.7 are specified three vertices to be deleted. First vertex `DF_NODE` represents a table to be removed. Second vertex `removeSpecChars` represents a function to be removed. Third vertex `t2c1` represents a column to be removed.

Notice, the second and the third vertex are specified also by their type (and all their parent node's as well). Moreover, a special character '*' (asterisk) can be used to represent any type. This way, user can specify only types of selected nodes.

Finally, logic of deleting vertices by their path is very simple. First, the input file is parsed in class `OneFileDeleter`. Second, for each line is called private class `ProcessPathTransactionCallback`. This class traverses from super root up to the target vertex. Once the target vertex is reached, the well-known method `SetSubtreeTransactionEnd` is called which removes this vertex with its whole subtree. See the listing 4.8.

```
1  public OneDeleteResult callMe(TitanTransaction transaction) {
2      Vertex parentVertex = getSuperRoot(transaction);
3      String parentName = "superRoot";
4      Vertex childVertex = null;
5      // walkthrough revision interval <rev, rev>
6      RevisionInterval revInterval = new RevisionInterval(rev,
           rev);
7
8      for (int i = 0; i < names.length; i++) {
9          String childName = names[i];
10         List<Vertex> children = getChildrenWithName(
               parentVertex, childName, revInterval);
11         childVertex = getChildVertex(children, parentName,
               childName, i);
12         parentName = childName;
13         parentVertex = childVertex;
14     }
15     // last child vertex is the target vertex to be deleted
16     Vertex targetVertex = childVertex;
17     // delete vertex by setting it to the previous revision
18     Double newEndRev = getPreviousRevision(rev);
19     setSubtreeTranEnd(targetVertex, rev, newEndRev);
20 }
```

Listing 4.8: ProcessPathTransactionCallback - callMe method

4.2 Manta Flow client

To build a fully functional prototype, input for the incremental update has to be created first on the client side. Hence, also logic of the input creation and sending it to the server had to be changed in Manta Flow client.

These changes were performed in the module **Merger Client**. In this module these three new *tasks* were created:

- `NodeFlagTask`
- `IncrementalDeleteTask`
- `DeleteVerticesTask`

These tasks are invoked during the incremental update on client side, depending on type of update (see section 3.4.2).

4.2.1 NodeFlagTask

`NodeFlagTask` is used during the process of creation of the input subgraph representing **new or modified scripts**. Its purpose is to only add the "*remove*" flag to all nodes specified by their type.

`NodeFlagTask` iterates through all nodes in the generated input subgraph (result of data flow analysis) and if the node is of a specified type, then a "*remove*" flag is appended in form of a node attribute to this node.

Node attribute is created only temporarily for the purposes of correct input creation. Before the input subgraph is sent to server, another class responsible for creating the input CSV file, `MergerWriter`, reads the input subgraph and if a node has node attribute containing flag, then this node attribute is ignored and only this flag is written to the CSV file to the node that is to be flagged.

4.2.2 IncrementalDeleteTask

`IncrementalDeleteTask` is used during the process of creation of the input subgraph representing **removed scripts**.

`IncrementalDeleteTask` creates a new graph containing only direct predecessors of the specified nodes. Only necessary data (path to the node to be removed) is sent. Additionally, it appends a flag "*remove_myself*" to these specified nodes, so these nodes are removed in the new revision, (*including* the flagged node).

This is implemented in three steps:

1. Find all specified nodes and save their path (their list of direct predecessors)
2. Clear the whole graph (remove all nodes and edges)
3. Create a new graph containing only nodes with their direct predecessors
    - Append flag "*remove_myself*" to the specified nodes (as a node attribute)

Thus, the main difference between `IncrementalDeleteTask` and `NodeFlagTask` is that `IncrementalDeleteTask` appends different type of flag and the input subgraph consists only of direct predecessors of a node to be removed (and this node of course).

On the other hand, inputs generated from both of these tasks can be processed by the same merger processor (`StandardMergerProcessor`, see section 4.1.3, listing 4.6).

4.2.3 DeleteVerticesTask

`DeleteVerticesTask` only sends to server a file containing vertices to be deleted. This file is then processed by `OneFileDeleter` in Merger (see section 4.1.3, listing 4.8).

This task does not contain almost any logic. It simply sends a file to server and waits for a response from server. Hence, all additional work such as file format checking or line filtering is also done on server.

4.3 Summary

A prototype based on the design from previous chapter was successfully implemented, including necessary modifications of existing unit tests and creation of new unit tests.

Manta Flow newly allows update of changed scripts in the metadata repository. New and modified scripts are analyzed and merged in the form subgraphs to the metadata repository. Removed scripts can be either merged to the repository in the form of subgraphs or removed directly by specifying their path in the repository.

The whole incremental update process can be summed up into the following steps:

1. Extraction phase (optional)
2. Create a new minor revision
3. Analytic phase - applied only on the selected (changed) scripts. Depending on type of change, following tasks are invoked during the input creation:
   - New or modified script - `NodeFlagTask`
   - Removed script - `IncrementalDeleteTask`
4. Post-processing phase (optional)
5. Commit the new minor revision

Incremental update is currently limited by updating only the changed scripts. Current implementation does not support update of database dictionaries. However, result of the update may vary, depending on the type of change in script. For instance, when a new table or a view is created[21], a new table or a view is created in the repository and correct data flows are created as well. On the other hand, when a column is removed from a table [22], the removed column will be still present in the repository, although, it will not be connected to the lineage (data flow to the table definition will no longer exist).

We could identify multiple of such inconsistencies and these may also differ by technologies used or other processes included during the update (e.g. whether a new database dictionary was extracted). To update these changes properly, data flow analyzer (responsible for creation of the input subgraphs)

[21] Command `"CREATE OR REPLACE TABLE/VIEW ..."` is added to a DDL script and this script is updated

[22] Command `"ALTER TABLE table_name DROP column_name"` is added into a DDL script and this script is updated

has to be customized for such changes. Therefore, the current implementation of incremental update may produce specific changes in the metadata repository and user has to be aware of such behavior.

Chapter 5

Performance Testing

Performance testing was performed, to test the the implemented incremental update in Manta Flow and compare its expected effectiveness with the original full update effectiveness. On top of that, we have also tested effectiveness of script removal during incremental update using either a direct path to the script root node or merging input subgraph representing the removed script.

Testing was performed on a following configuration:

- **Operation system** - Windows 10 Pro (64 bit)
- **Processor** - Intel(R) Core(TM) i7-6820HQ CPU @ 2.70GHz 2.70GHz
- **RAM** - 8 GB
- **Manta Flow version** - product version 1.20 / prototype (originating from version 1.20)

5.1 Test cases

All of these test cases follow the same scenario:

1. Empty database (revision 0)
2. Create new PL/SQL scripts
3. Add these scripts to the database using full update (revision 1)
4. Change some scripts (at maximum 10 % of all scripts)
5. Perform full/incremental update (revision 2)

The only difference between these test cases is in the update from the first revision to the second revision. Some cases perform full update and some cases perform incremental update:

Case 1 Full update integer

- Revision 1: Full update – add all scripts
- Revision 2: Full update – update all scripts

Case 2 Full update double

- Revision 1: Full update – add all scripts
- Revision 2: Full update – update all scripts

Case 3 Incremental update all

- Revision 1: Full update – add all scripts
- Revision 2: Incremental update - update all scripts

Case 4 Incremental update only changed

- Revision 1: Full update – add all scripts
- Revision 2: Incremental update – update only changed scripts

Case 5 Remove by graph

- Revision 1: Full update – add all scripts
- Revision 2: Incremental update – update removed scripts using merging (`IncrementalDeleteTask`)

Case 6 Remove by path

- Revision 1: Full update – add all scripts
- Revision 2: Incremental update – update removed scripts using path (`DeleteVerticesTask`)

In all test cases is always performed the same full update in the first revision. Second update is different for each test case but the result is always the same. Scripts that were created after first revision are present only in the second revision and scripts removed after the first revision are absent in the second revision. The only difference is the way the update is performed.

Test case 1 is testing full update on the Manta Flow version using integers for revision numbers (the old version without incremental update). Test cases 2 – 6 are all tested on the new version using double data type for revision numbers.

Test case 2 is testing the same full update as test case 1 but using the new version (with doubles).

Test case 3 performs incremental update in the second revision but the incremental update is applied on all scripts, including the unchanged scripts.

Test case 4 performs incremental update but on the input are only the changed scripts.

Test cases 5 and 6 also perform only incremental update but all changed scripts are removed. The only difference between test cases 5 and 6 is, what method is used for script removal.

5.2 Test data

For performance testing were used only Oracle PL/SQL scripts, provided by supervisor. Test cases 1 – 4 were run on three different sizes of data:

Scenario name	#Scripts	#Nodes + edges	#Changed scripts
small	400	124 000	20
big	3 100	220 000	145
large	3 700	405 000	29

Table 5.1: Input data for test cases 1, 2, 3 and 4

Column **Scripts** stands for the number of scripts stored in database after the first update. **Nodes + edges** column represents a total number of objects in database after the first update. Notice, every script can be represented by a different number of nodes and edges, depending on script's complexity.

Changed scripts column stands for a number of scripts that were either newly added or removed in the second revision. The number of changed scripts was always chosen to be significantly smaller than the overall number of scripts stored in the database. Also, changed scripts were chosen to be of a similar complexity (so differences of update times of each script were similar).

Test cases 5 and 6 were run for a different size of input (2 000 scripts, 96 000 nodes + edges) and there were performed incremental updates to the second revision removing three different numbers of scripts each time:

Scenario name	#Scripts	#Nodes + edges	#Removed scripts
smallRemove	2 000	96 000	5
mediumRemove	2 000	96 000	25
largeRemove	2 000	96 000	125

Table 5.2: Input data for test cases 5 and 6

5.3 Measurement

All test cases from above were measured only for the **time spent on updating the graph database** on server (test cases 1-4 *merge time*, test cases 5 and 6 *merge/delete time*). In these tests were not included times necessary for

extraction, data flow analysis, input creation, sending input to the server and other processes connected with the whole update (edge propagation, backlinks creation, pruning revisions, exporting etc.).

For each data size (small, big and large) were performed in each test case 5 runs. Hence, test cases 1-4 were run 5 times for the small input, 5 times for the big and 5 times for the large input. Following table summarizes average merge times of second update (second revision) for test cases 1-4:

Scenario name	**Case1 [ms]**	**Case2 [ms]**	**Case3 [ms]**	**Case4 [ms]**
small	78 930	89 870	100 095	1 918
big	197 771	218 181	244 790	13 453
large	373 503	379 400	399 525	2 790

Table 5.3: Average merge time of test cases 1, 2, 3 and 4

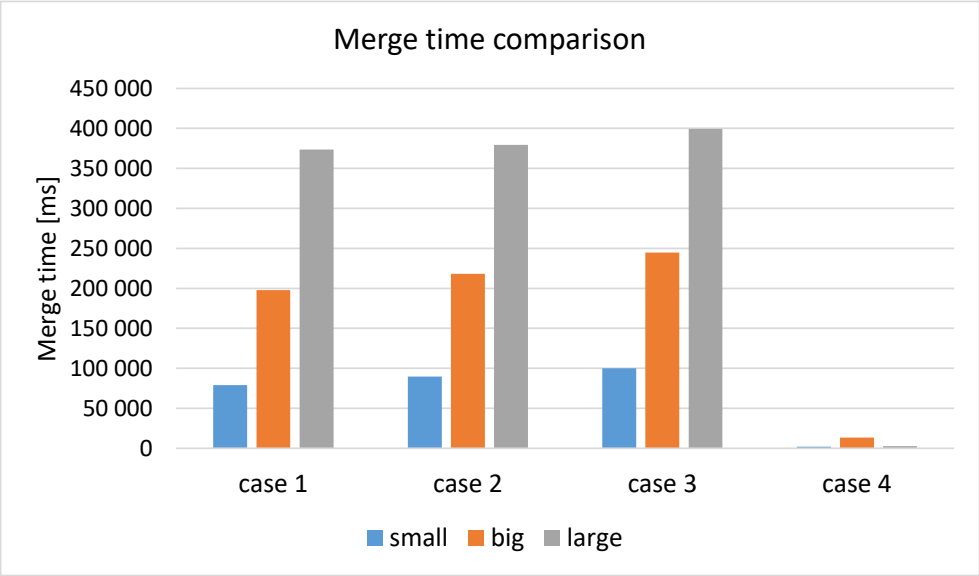


Figure 5.1: Merge time comparison of test cases 1, 2, 3 and 4

From table 5.3 and its visualization in figure 5.1 (and 5.2) are obvious the following points:

1. Full update using Double (case 2) instead of Integer (case 1) is only slightly slower. What more, the bigger the input, the smaller the impact on merge time (small input is slower by 13 %, big input is slower by 10 %, large input is slower by 1%)

2. Performing incremental update as full update (case 3) is also only slightly slower than the regular full update (case 2). Although, this "dummy" incremental update would be expected to be significantly slower than

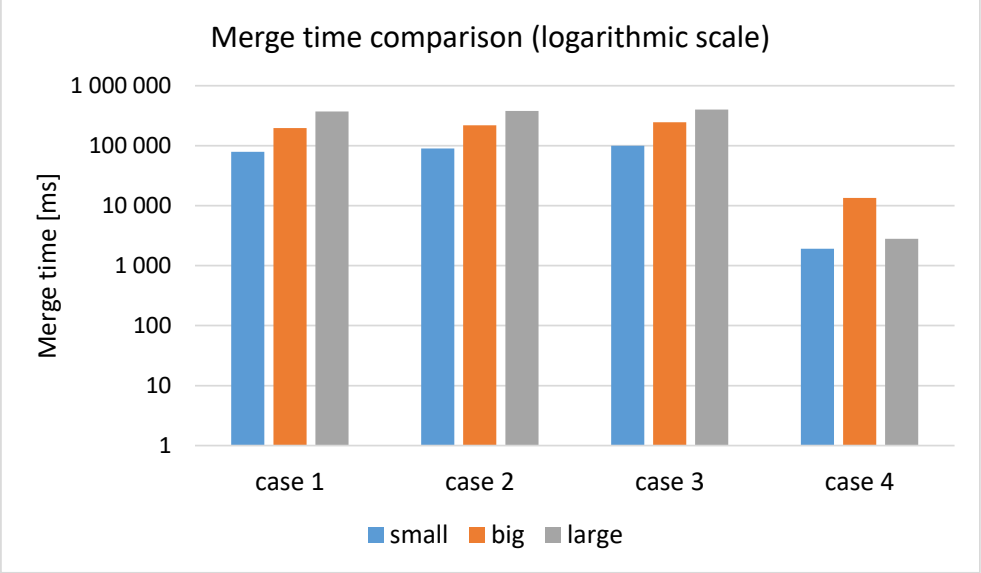


Figure 5.2: Merge time comparison of test cases 1, 2, 3 and 4 in a logarithmic scale

the same full update (since it is incrementally updating all scripts and not only the changed ones) it is not (small input is slower by 11 %, big input is slower by 12 %, large input is slower by 5 %). The reason is, the additional operation invoked at every script in the database during incremental update (`setSubtreeTranEnd`) does not actually delete vertices and edges representing the script. It only sets values of tranEnd properties and this operation is not significantly time-consuming.

3. Incremental update applied only on the changed scripts (case 4) is working exactly as expected. Its merge time is *directly proportional* to the size of the changed scripts:

$$\frac{1918ms}{13453ms} \doteq \frac{20scripts}{145scripts} \qquad \frac{1918ms}{2790ms} = \frac{20scripts}{29scripts}$$

This is the most desired ability of incremental update, since it is independent on the total volume of data present in the new revision.

The next table 5.4 captures time spent on deleting scripts from database either by merging (case 5) or by specifying their path in database (case 6):

From table 5.4 and its visualization in figure 5.3 is obvious that:

1. Removing a small number of scripts (in this case less than 25) is faster when using their direct paths (case 6)
2. Removing a higher number of scripts (in this case more than 25) is faster when using merging mechanism (case 5)

Scenario name	Case 5 [ms]	Case 6 [ms]
smallRemove	817	252
mediumRemove	1 463	1 325
largeRemove	4 255	6 522

Table 5.4: Average delete time of test cases 5 and 6

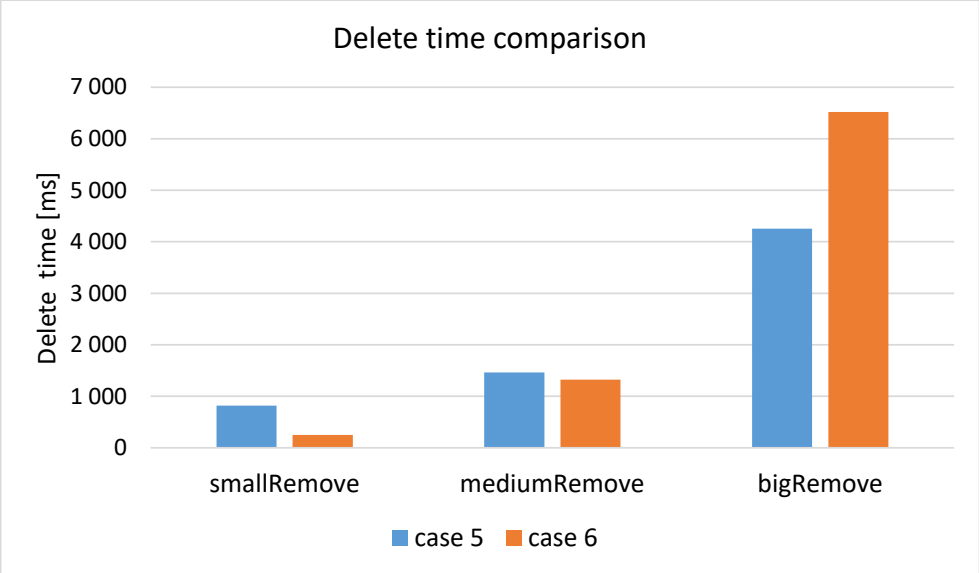


Figure 5.3: Delete time comparison of test cases 5 and 6

This is a natural behavior. Removing one script just by searching for its root node (remove by path) is definitely faster than also applying the merging process before reaching this root node (see analysis of Object removal in section 3.4.2).

However, when applying removal by path on 125 scripts, this process has to be repeated 125 times. Hence, the scripts' root nodes have to be located 125 times, each time starting traversal from super root. If all 125 scripts are located in the same subtree in database, then a traversal up to their common direct parent node is performed **each times**.

On the other hand, when all these 125 scripts are to be deleted by merge and all are located in the same subtree in database, then the traversal to their common direct parent node is performed **only once**[23]. Moreover, thanks to the preparation on client's side (`IncrementalDeleteTask`), only the direct path to the script's root node is created. Thus, no redundant merges are performed during the merge-removal.

[23]The traversal is performed only once because the input subgraph created by dataflow analyzer is a one complete subgraph containing all scripts' common predecessors only once.

To sum it up, due to the redundant traversals, the removal by path is slower than the removal by merge when a higher number of scripts from the same location is to be removed. Nevertheless, this testing *does not include* the time necessary for preparation of the input subgraph for the merge process, and thus, the overall removal time may significantly vary in all three cases.

5.4 Summary

Performance testing proved expectations of the implemented incremental update in Manta Flow. Incremental update merge time is directly proportional to the size of the input (change size). Moreover, change of revision number data type from Integer to Double causes only a slight slowdown. Hence, the implementation of incremental update does not exclude simultaneous usage of full updates with incremental updates. Therefore, the implementation of incremental update in Manta Flow can be considered as successful and can be deployed to the product version.

Conclusion

The goal of this thesis was to design and implement incremental updates of data lineage storage in a graph database. New incremental update methods have been successfully designed, implemented and tested. Performance testing proved assumptions of the analysis. When using incremental update instead of full update, the update time is directly proportional to the size of the change. Thus, the update time is significantly decreased in case of updating minor changes.

Moreover, performance testing also proved only a slight slowdown when the same amount of data is updated by incremental update instead of full update. This applies also for a large amount of data. Therefore, the implemented incremental update can be deployed to the product version of Manta Flow, and thus, significantly reducing time necessary for updating changes in the data lineage storage.

Due to the nature of data processed by Manta Flow, the new incremental update methods cannot be currently applied on all possible changes. Moreover, inconsistencies may be brought to the data lineage storage, if incremental update methods are not used correctly. Both of these issues can be solved by performing an impact analysis on the changed data, followed by creation of the correct input for the incremental update. The downside of the impact analysis is that it may cause an equal, or even higher, time complexity of the whole incremental update process rather than using only a full update method. Therefore, follow-up to this work may include an analysis of a more suitable incremental update input creation process that would eliminate the potential inconsistencies in a reasonable time.

Bibliography

[1] Holeček, P. *Temporální data v grafové databázi v projektu Manta.* Master's thesis, Czech Technical University in Prague, Faculty of Information Technology, 2015.

[2] Company – MANTA. Oct 2017, [Cited 2017-11-08]. Available from: https://getmanta.com/company/

[3] Manta Flow. 2017, [Cited 2017-11-08]. Available from: https://marketplace.informatica.com/solutions/manta_flow

[4] Hermann, L. How To Inspect Raw Data Lineage With Manta Flow. Jan 2017, [Cited 2017-11-08]. Available from: https://getmanta.com/how-to-inspect-raw-data-lineage-with-manta-flow/

[5] Manta Flow. [Cited 2017-11-08]. Available from: https://www.capterra.com/p/145178/Manta-Flow/

[6] Home. [Cited 2017-11-15]. Available from: http://titan.thinkaurelius.com/wikidoc/0.4.4/Home.html

[7] Transaction Handling. [Cited 2017-11-15]. Available from: http://titan.thinkaurelius.com/wikidoc/0.4.4/Transaction-Handling.html

[8] Architectural Overview. [Cited 2017-11-15]. Available from: http://s3.thinkaurelius.com/docs/titan/1.0.0/arch-overview.html

[9] LaRocque, D. Using Persistit. Apr 2014, [Cited 2017-11-15]. Available from: http://titan.thinkaurelius.com/wikidoc/0.4.4/Using-Persistit.html

[10] LaRocque, D. TinkerPop Graph Stack. Apr 2014, [Cited 2017-11-15]. Available from: http://titan.thinkaurelius.com/wikidoc/0.4.4/TinkerPop-Graph-Stack.html

[11] Blueprints interface. Apr 2014, [Cited 2017-11-15]. Available from: http://titan.thinkaurelius.com/wikidoc/0.4.4/Blueprints-Interface.html

[12] LaRocque, D. Titan Data Model. Apr 2014, [Cited 2017-11-15]. Available from: http://titan.thinkaurelius.com/wikidoc/0.4.4/Titan-Data-Model.html

[13] Chang, F.; Dean, J.; Ghemawat, S.; et al. Bigtable: A Distributed Storage System for Structured Data. 2006, [Cited 2017-11-15]. Available from: https://static.googleusercontent.com/media/research.google.com/en//archive/bigtable-osdi06.pdf

[14] Musso, J.-B. Defining a Property Graph. Jul 2016, [Cited 2017-11-15]. Available from: https://github.com/tinkerpop/gremlin/wiki/Defining-a-Property-Graph

[15] Kratochvíl, E. Vrstvy a interpolace hran. Apr 2017, [Cited 2017-11-20]. Available from: https://mantatools.atlassian.net/wiki/spaces/MTT/pages/62652429/Vrstvy+a+interpolace+hran

[16] LaRocque, D. Indexing Backend Overview. Apr 2014, [Cited 2017-11-15]. Available from: http://titan.thinkaurelius.com/wikidoc/0.4.4/Indexing-Backend-Overview.html

[17] LaRocque, D. Type Definition Overview. Apr 2014, [Cited 2017-11-15]. Available from: http://titan.thinkaurelius.com/wikidoc/0.4.4/Type-Definition-Overview.html

[18] LaRocque, D. Using Lucene. Apr 2014, [Cited 2017-11-15]. Available from: http://titan.thinkaurelius.com/wikidoc/0.4.4/Using-Lucene.htmls

[19] LaRocque, D. Vertex Centric Indices. Apr 2014, [Cited 2017-11-15]. Available from: http://titan.thinkaurelius.com/wikidoc/0.4.4/Vertex-Centric-Indices.html

[20] Collins-Sussman, B.; Fitzpatrick, B. W.; Pilato, C. M. Version Control Basics. 2011, [Cited 2017-12-07]. Available from: http://svnbook.red-bean.com/en/1.7/svn.basic.version-control-basics.html

[21] Collins-Sussman, B.; Fitzpatrick, B. W.; Pilato, C. M. Version Control the Subversion Way. 2011, [Cited 2017-12-07]. Available from: http://svnbook.red-bean.com/en/1.7/svn.basic.in-action.html

[22] Collins-Sussman, B.; Fitzpatrick, B. W.; Pilato, C. M. Basic Work Cycle. 2011, [Cited 2017-12-07]. Available from: http://svnbook.red-bean.com/en/1.7/svn.tour.cycle.html

[23] Subversion Design. 2002, [Cited 2017-12-07]. Available from: https://svn.apache.org/repos/asf/subversion/trunk/notes/subversion-design.html

[24] WC-NG Nodes. 2002, [Cited 2017-12-07]. Available from: https://svn.apache.org/repos/asf/subversion/trunk/notes/wc-ng/nodes

[25] WC-NG Design. 2002, [Cited 2017-12-07]. Available from: https://svn.apache.org/repos/asf/subversion/trunk/notes/wc-ng/design

[26] What is Subversion's WC-NG? April 2010, [Cited 2017-12-07]. Available from: http://prng.blogspot.cz/2010/04/what-is-subversions-wc-ng.html

[27] Collins-Sussman, B.; Fitzpatrick, B. W.; Pilato, C. M. Strategies for Repository Deployment. 2011, [Cited 2017-12-07]. Available from: http://svnbook.red-bean.com/en/1.7/svn.reposadmin.planning.html

[28] Repository. March 2013, [Cited 2017-12-14]. Available from: https://www.mercurial-scm.org/wiki/Repository

[29] Buehlmann, A. CVS Concepts. November 2012, [Cited 2017-12-14]. Available from: https://www.mercurial-scm.org/wiki/CvsConcepts

[30] O'Sullivan, B. Behind the scenes. [Cited 2017-12-14]. Available from: http://hgbook.red-bean.com/read/behind-the-scenes.html

[31] Bruna, W. Revlog. February 2012, [Cited 2017-12-14]. Available from: https://www.mercurial-scm.org/wiki/Revlog

[32] Bullock, K. RevlogNG. March 2015, [Cited 2017-12-14]. Available from: https://www.mercurial-scm.org/wiki/RevlogNG

[33] Lippincott, K. Nodeids. December 2015, [Cited 2017-12-14]. Available from: https://www.mercurial-scm.org/wiki/Nodeid

[34] Jagula, M. Design. October 2013, [Cited 2017-12-14]. Available from: https://www.mercurial-scm.org/wiki/Design

[35] Changeset. January 2017, [Cited 2017-12-14]. Available from: https://www.mercurial-scm.org/wiki/ChangeSet

[36] Incremental backup and restore. [Cited 2017-12-14]. Available from: https://developer.couchbase.com/documentation/server/3.x/admin/Tasks/tasks-backup-restore-incremental.html

[37] Fechtner, T.; Toušek, J. Manta Flow Client Architecture. Dec 2017, [Cited 2017-11-20]. Available from: https://mantatools.atlassian.net/wiki/spaces/MTKB/pages/8650854/Manta+Flow+Client+Architecture

[38] LaRocque, D. Titan Limitations. Apr 2014, [Cited 2017-11-20]. Available from: http://titan.thinkaurelius.com/wikidoc/0.4.4/Titan-Limitations.html

Appendix A

Acronyms

JVM Java virtual machine

SQL Structured Query Language

CSV Comma-separated values

PLSQL Procedural Language/Structured Query Language

SVN Apache Subversion

VCS Version control system

CVCS Centralized version control system

DVCS Distributed version control system

Appendix B

Contents of enclosed CD

- `src` ..... implementation source codes
- `thesis` ..... LaTeX source codes of the thesis
  - `graphs` ..... XML source codes of graphs
  - `images` ..... images used in thesis
- `testing` ..... performance testing directory
- `DP_Sykora_Jan_2017.pdf` ..... thesis text in PDF format
- `readme.txt` ..... file with CD contents description