FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Josef Kumstýř

Precise and Efficient Incremental Update of Data Lineage Graph

Department of Distributed and Dependable Systems

Supervisor of the master thesis: doc. RNDr. Pavel Parízek, Ph.D.
Study programme: Computer Science
Study branch: Software Systems

Prague 2022

I would like to thank my supervisor Pavel Parízek, my family, and my colleagues in MANTA, especially Lukáš Hermann and Petr Košvanec.

Title: Precise and Efficient Incremental Update of Data Lineage Graph

Author: Josef Kumstýř

Department: Department of Distributed and Dependable Systems

Supervisor: doc. RNDr. Pavel Parízek, Ph.D., department

Abstract: Data lineage forms an essential aspect of today's enterprise environment. MANTA Flow is a data lineage analysis platform that works based on extracting and analyzing customers' source files. However, often the customer wants to update the data lineage graph because of a slight change in provided source files. However, all of the input source files are currently reanalyzed, and most of the time is wasted analyzing unchanged files. In the thesis, we presented how the data lineage analyzer can be improved using incremental updates to analyze only a fraction of all input files while still producing the same correct data lineage.

We changed how the whole analysis is done by changing the granularity of the analysis to much smaller pieces. We also improved the merge algorithm to recognize when an unchanged file could generate a different data lineage using new concepts like source segments, node removal, or node creation. The new MANTA client algorithm now analyzes only changed files and a few unchanged files that could generate a different lineage compared to the last analysis. We also implemented a prototype for the MANTA Oracle scanner that contains these new ideas. It was tested for both the correctness and the performance.

Keywords: data lineage, incremental updates, static analysis, data flow graph, Manta

Contents

1. Introduction

Every organization uses data to stay relevant and competitive while undergoing a constant digital transformation. Nowadays, the amount of data is too huge for many organizations to inspect manually. MANTA Flow [1] is a platform that generates and automatically updates data lineage information, which shows the origin of data and its journey through all the data-processing systems. The result is a data lineage graph that is understandable for company developers and its other stakeholders. The platform eliminates human error and provides accurate lineage information based on hard facts rather than guesses and assumptions.

Tracking data lineage inside client systems is especially important for audits and legal reasons. For example, it is necessary to have complete information about the flow of all client's data throughout the system. Then, whenever the client asks the system to delete all information about themselves, the system should remove all the traces of their existence.

MANTA Flow generates data lineage graphs based on extracting and analyzing source files provided by clients as input. However, in the current version of MANTA Flow, if a client wants to update the data lineage graph because of a slight change in provided source files, all input source files are reanalyzed, which can take many hours. Most of this time is unnecessarily spent analyzing unchanged files that generate the same data lineage graph as the previous analysis run.

This thesis project aims to speed up the data lineage analysis with the incremental update. The main idea of incremental update in MANTA Flow is to reanalyze only a fraction of all the input files that are sufficient to obtain the same data lineage graph as if a client would run a full analysis analyzing all the input files.

1.1 Goals

The first goal of this thesis is to analyze the current version of the MANTA Flow platform and identify all the technical challenges related to the design and implementation of incremental updates, including the effects on the current data lineage analysis. Then, based on this analysis, the second goal of this thesis is to design an efficient and precise algorithm for the incremental update of data lineage graphs. The third goal is to implement this algorithm and create a working prototype of the incremental analysis for the Oracle database technology within the MANTA Flow. Lastly, the fourth goal is to evaluate the created prototype using extensive testing, validation, and performance evaluation.

1.2 Related works

Jan Sýkora has already partly examined this topic in his thesis Incremental update of data lineage storage in a graph database [2]. He introduced the concept of minor revisions, and this thesis will use this concept and further improve its usability for customers. As written in the thesis, the current state is as follows. "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." There is currently only one customer using incremental updates because it has minimal usage, and it is also very complex to use correctly. In case of incorrect usage, generated dataflow is wrong, and the customer does not know about that. We want to reduce these limitations as much as possible by improving underlying algorithms while still maintaining a significant speedup.

1.3 Outline

Chapter 1 introduces the problematics containing a basic overview of the problem and the project goals. Chapter 2 describes both the basics of the MANTA Flow platform and some of its more advanced parts needed to design incremental updates. In chapter 3, we define the solution's key concepts and state the requirements for the solution. Chapter 4 describes the design of incremental updates and how the new merge algorithm was created. Chapter 5 describes the implementation of incremental updates for Oracle database technology within the MANTA Flow application. Chapter 6 evaluates the solution by describing performed tests for both correctness and performance and by discussing solution limitations and possible future extensions. Lastly, chapter 8 concludes the thesis.

2. MANTA Flow platform

This chapter describes parts of the MANTA Flow platform [1], which are essential in the context of this thesis. First, this chapter contains several general topics. Then, starting from Section 2.9, there are a few more specific topics, which can be either read in order or skipped and returned to whenever future chapters reference them.

2.1 Architecture

MANTA is a client-server application. Client and server communicate via a specified interface using TCP/IP protocol.

Client workflow has several steps. First, it extracts all available metadata using connections to the user's systems provided by them. This means extracting source code files, database dictionaries, ETL jobs, and (business intelligence) reports. Then, the client analyzes these metadata and generates parts of the dataflow graph. Usually, each extracted entity (e.g., every source code file or data dictionary) generates one graph, which contains the data lineage of this entity. Finally, the client sends these graphs to the server one by one.

The server has one primary responsibility: taking care of the complete data lineage graph. As the client sends only parts of this graph at the time, the server stores the first part. Each additional part sent by the client is then merged with the stored graph. After the client sends all parts, the server contains a graph with a complete data lineage of the user's environment.

The last remaining thing is to display this resulting graph to the user. The most common approach is to use MANTA Viewer, which shows this graph in the browser. We can see an elementary visualization example in Figure 2.1, which shows on the left the Oracle database `ORCL` with the table `SOURCETABLE` containing the column named `STRING`. Arrow represents a data flow from the Oracle database column to the Power BI data model column `String column` on the right. From the `String column` there, the data flow continues to the aggregation column `Count of String column`, which computes the number of rows in the `String column`.

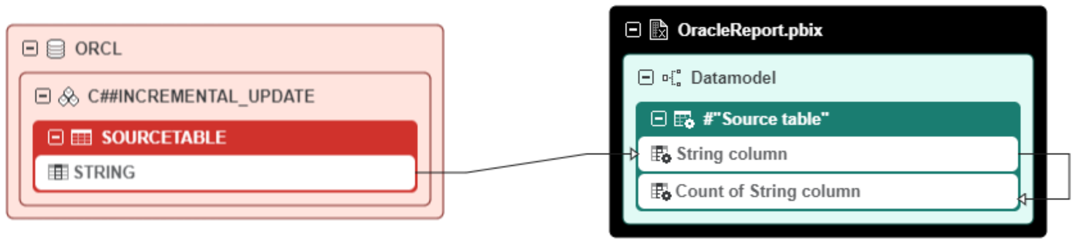

Figure 2.1: Visualization example

2.2 Scenarios

Client work is specific for each technology, divided into several scenarios. Therefore, each technology has a set of scenarios, which are run sequentially. These scenarios are divided into extraction scenarios (which extract and save user's metadata) and analysis scenarios (which create data flow based on extracted metadata).

Let us look at the concrete Oracle scenarios essential for our prototype. Oracle has these scenarios:

- Oracle dictionary mapping master scenario. This scenario connects to each configured Oracle database and stores important server metadata (e.g., global database name).
- Oracle extractor master scenario. This scenario connects to each configured Oracle database and extracts the data dictionary and DDL scripts.
- Oracle dictionary dataflow master scenario. This scenario analyzes metadata from the extracted Oracle data dictionaries and stores the resulting flow graph on the server.
- Oracle DDL dataflow master scenario. This scenario analyzes metadata from the extracted Oracle DDL scripts and stores the resulting flow graph on the server.
- Oracle PL/SQL dataflow master scenario. This scenario analyzes metadata from the provided PL/SQL scripts and stores the resulting flow graph on the server.

2.3 Graph databases

MANTA Flow platform used Titan database [3] for storing, accessing, and transforming the data flow graph. However, for many reasons, it has been replaced by Neo4j database [4]. This process of replacing happened during the implementation of this thesis.

Using the Titan database while creating the prototype for this thesis would not make sense because it would need a considerable rework as soon as the database changed. Therefore our implementation uses the Neo4j database.

2.4 Data model

In Section 2.1, we mentioned that the server contains the data lineage graph created based on the client's metadata. Let us now look at what are vertices and edges of this graph.

2.4.1 Vertices

Each vertex has precisely one vertex type. There are currently nine different vertex types in MANTA. Let us describe them one by one.

The **super root** is the first vertex type. Vertex with this type is the root of the whole data flow graph. There always exists precisely one vertex of this type.

The **asset** node represents an object in the data flow graph. Every asset has a specified node type, which denotes what the asset node represents. Node type can, for example, be a database, procedure, schema, table, column, or even a file.

An **attribute** represents additional information about an asset. An asset representing a database column can for example contain an attribute with information about column's data type or if it is a nullable column.

The **resource** node specifies the technology of its descendant asset nodes. Resource can be, for example, MSSQL, Oracle, Power BI, C#, or Filesystem. Every asset node belongs to a specific resource.

A **layer** represents a logical level of stored metadata. The purpose of the layer is to distinguish between different views of the modeled reality. However, the exact meaning of this node is not important. We only need to know that it exists.

The **revision root** is the root of a separate revision tree graph, which describes all the existing revisions. Revisions serve as a version control system for the generated data flow graph. One revision represents the actual state of the whole data flow graph at one specific time point. There always exists precisely one vertex of this type.

The **revision node** represents one specific revision.

The **source root** is the root of a separate tree graph, which describes all the source code files extracted from the client. There always exists exactly one vertex of this type.

The **source node** represents a path to one specific extracted source code file stored on the server.

2.4.2 Edges

Each edge has exactly one edge type. There are currently nine different edge types in MANTA. Let us describe them one by one.

Direct flow edge represents a direct data flow from its source node to its target node. These data may be transformed (e.g., filtered or sorted) on their way to the target node. This edge type can only connect leaves of the data flow graph.

Filter flow edge represents an indirect data flow from its source node to its target node. Unlike the direct data flow, the source node only indirectly affects what data flows to the target node. For example, filter flow edges are used for nodes representing conditions of IF statements because they decide whether some other data will flow to the target node. This edge type can also only connect leaves of the data flow graph.

InLayer edge connects a resource node with a layer node. This edge specifies for the resource node to which layer belong all its assets.

MapsTo edge represents a relationship between nodes from different layers.

HasParent edge connects a child asset node with its parent asset node.

HasAttribute edge connects an asset node with its attribute(s).

HasResource edge connects a node with its resource node.

HasRevision edge connects a revision node with a revision root node.

HasSource edge connects a source node with a source root node.

2.5 Graph structure

Now that we know all the types of nodes and edges, let us look at how they're connected into graphs. There are three different graphs, and we will look at all of them.

2.5.1 Data flow graph

The most important graph is the data flow graph. Its abstract example is shown in Figure 2.2. We can see that the super root node is the root of the whole graph and has several resource nodes as children, each representing a different technology. Each resource node also belongs to exactly one layer represented by a layer node. Each resource node usually has many asset nodes as its descendants. Every asset node can have one or more attributes.

We can also see in Figure 2.2 that there are DirectFlow, FilterFlow and MapsTo edges only between the asset nodes on the lowest level (i.e., between asset nodes without any child nodes). Dashed arrows visualize these edges. Note that if we remove all these dashed edges, the resulting subgraph is a tree graph. This is a critical property of a data flow graph.

We can see a concrete example of a data flow graph in Figure 2.3. This example shows a data flow graph for an Oracle technology, represented by a green Oracle resource node. This technology contains a database named ORCL with a single dbo schema. There is a table named records in this schema, which contains one id column. We can see that all these four database objects have their blue asset nodes connected with a blue HasParent edges according to their hierarchy in the database. Note that the column also has two yellow attributes nodes connected by HasAttribute edges, which provide additional information about

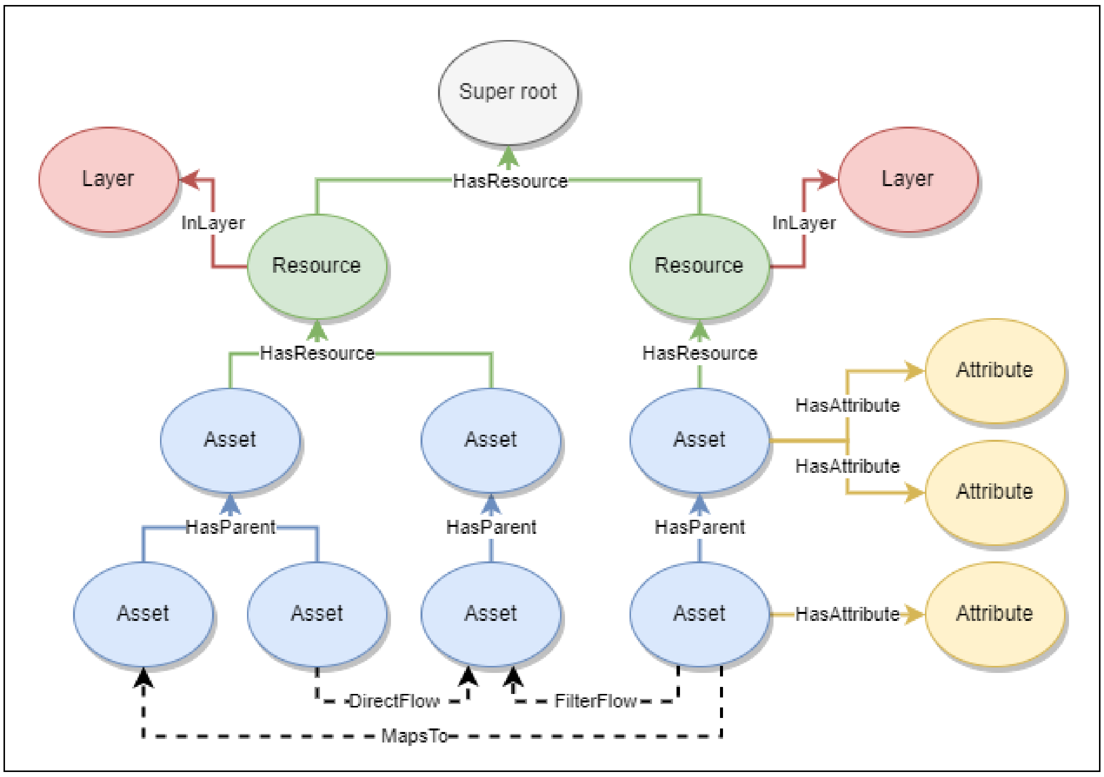

Figure 2.2: Data flow graph example

this column. In reality, there would be many more attributes in this graph, but they are not crucial for this example.

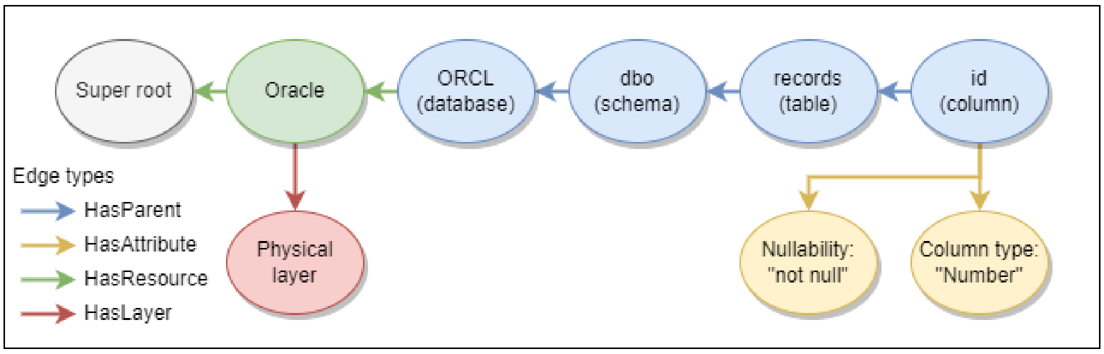

Figure 2.3: Data flow concrete graph example

2.5.2 Source file graph

A source file graph contains pointers to all extracted source code files. Because these files aren't stored in the graph database, source nodes contain references to the physical locations of these files. The source file graph structure is shown in Figure 2.4, where we can see that it contains one source root and multiple source nodes. All source nodes are connected to the source root with a HasSource edge.

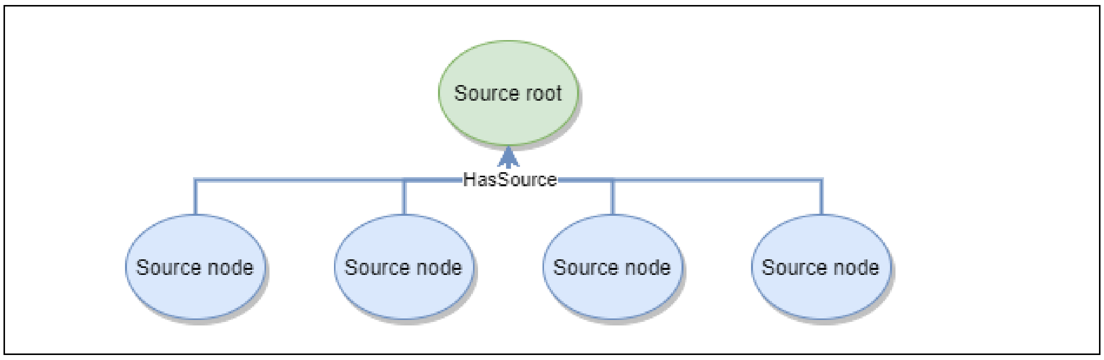

Figure 2.4: Source file graph example

2.5.3 Revision graph

A revision graph contains the list of all revisions. Its structure is shown in Figure 2.5 and is very similar to the source file graph. We can see that this graph contains one revision root, which is connected to revision nodes with HasRevision edge. Each revision node represents one MANTA revision.

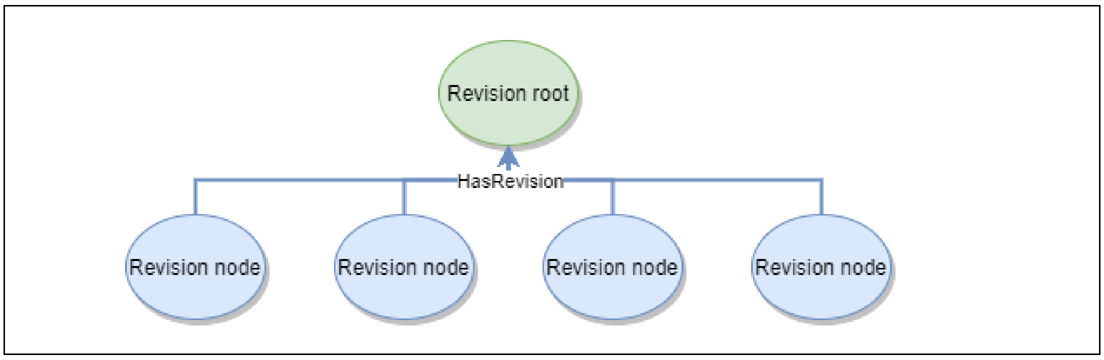

Figure 2.5: Revision graph example

2.6 Revisions

Before describing specific algorithms, we need to dive deeper into revisions. First, we find out how revisions were implemented historically. Then, we will look at the changes invented and implemented in the previous thesis (referenced in Section 1.2), which created the foundation for incremental updates.

2.6.1 Closed revision

First, let us look at how revisions were implemented historically. An integer represented each revision. The first revision had the number 0, the next one had the number 1, and so on. Every edge had two attributes TranStart and TranEnd. TranStart is the number of the revision in which the edge was created, and TranEnd is the number of the latest revision in which the object still existed.

Nodes do not have these attributes. Their validity is derived from the edge to their parent. Note that every node apart from the root node has exactly one parent, the root node is implicitly valid in all revisions. This type of end revision is called a closed end revision because every object has specified its end revision at all times.

Let us look at Figure 2.6, where we can see on the left a data flow graph with current revision 3 and nodes A-F. Each edge has a label with two numbers, the first one is TranStart and the second one is TranEnd. All nodes and edges in the left half have label 1,3, which means that they were created in revision 1, and are still valid in current revision 3. In the next revision 4, the node E was removed and the node 1 was added, which is denoted by their colors.

On the right side of Figure 2.6, we can see what the graph looks like in revision 4 after the previously mentioned changes. Graph objects that need an update in revision 4 because of these changes are denoted by an orange color. It is not only the new node 1 that's updated but (parent edges of) all unchanged nodes B, C, D, F as well. The reason is that their TranEnd must be updated to 4, otherwise, they would not be valid in the new revision 4. On the other hand, deleted nodes do not need any update, as we can see on the example of the node E.

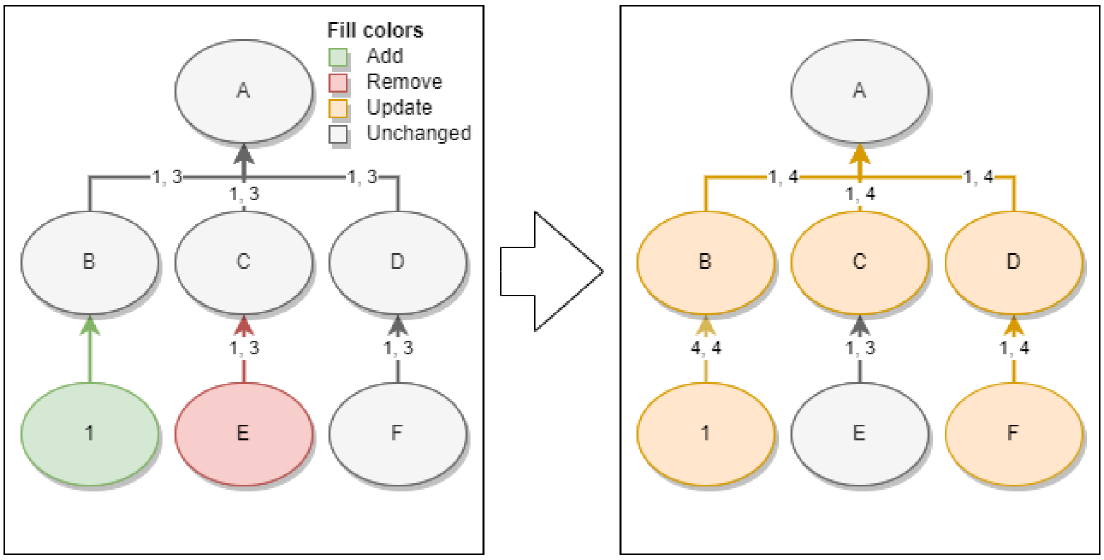

Figure 2.6: Closed revision example

Let us summarize the advantages and disadvantages of the closed revision. The most significant disadvantage is that it is necessary to update revisions of all the nodes and edges which weren't changed in the new revision. On the other hand, the most significant advantage of the closed revision is that revisions of all deleted nodes and edges do not need any updates.

2.6.2 Open revision

Let us look at a different approach called an open revision. In this approach, TranEnd is by default set to ∞. And only after removing a node or an edge from the current revision then their TranEnd is set to the number of the last revision this object was valid in.

Let us look at the example in Figure 2.7, which is the same graph like the last time, but this time it will be changed using an open revision approach. The only change in the left part is that existing nodes have revisions 1,∞ as they were added in revision 1 and they're indefinite.

However, there are many changes on the right side of Figure 2.7. First, TranEnd of the node E must be changed to the revision 3. Otherwise, it would still be valid in the new revision 4. Second, none of the unchanged nodes or edges need any update because their still valid in the new revision thanks to their TranEnd being set to ∞. Lastly, the new node 1 is added with label 4,∞ so it's valid only from revision 4.

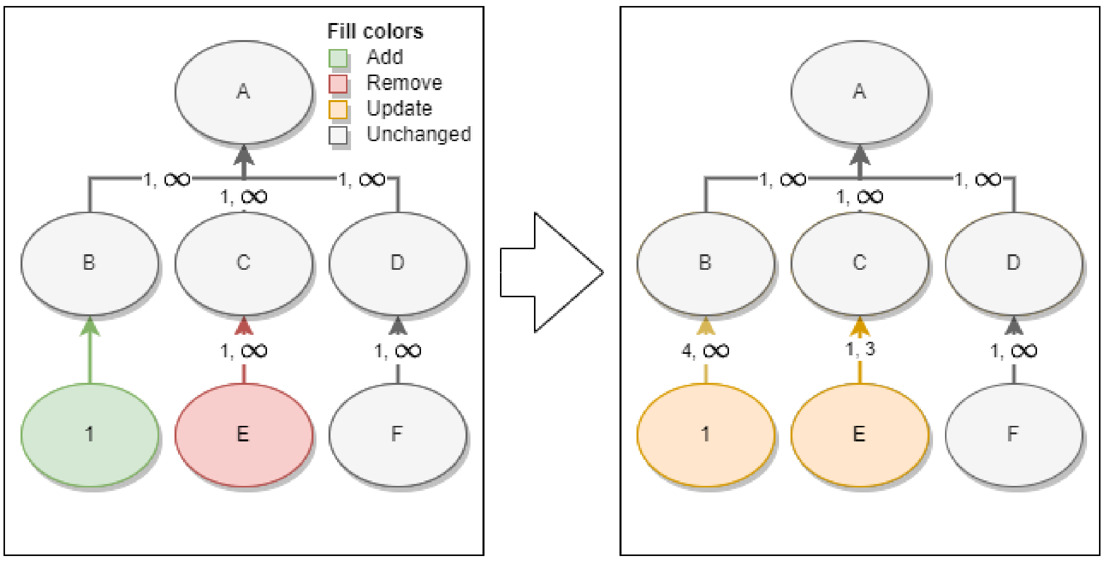

Figure 2.7: Open revision example

Let us again look at the advantages and disadvantages of the open revision. This time, the advantage is that revisions of all unchanged nodes and edges do not need any update. However, revisions of all deleted nodes need an update.

2.6.3 Two-level revision

Let us now compare closed and open revisions. Newly added nodes and edges need an update in both approaches. Deleted nodes and edges need an update only when using the open revision. Furthermore, unchanged nodes and edges need to be updated only when using the closed revision. Therefore both approaches can be useful; it depends on the ratio of deleted and unchanged nodes and edges.

Because both closed and open revision can be useful, the previous thesis [2] decided to combine both of these into a new two-level revision approach. This combined approach benefits from the advantages of both of them.

Instead of only two revision parameters, TranStart and TranEnd, there are now four new revision parameters MajorStart, MajorEnd, MinorStart, and MinorEnd. Parameters MajorStart and MajorEnd represent major revisions, which keep the system of the closed revisions. Parameters MinorStart and MinorEnd represent minor revisions, which use the new open end revision. We interpret major revisions as integers and minor revisions as their decimal part. Therefore, the new two-level revision can be, for example, revisions 1.0, 1.1, 1.2, 2.0, 2.1, etc. For each node, we will call the number MajorEnd.MinorEnd **end revision** of a node and the number MajorStart.MinorStart **start revision** of a node.

Whenever updating the data flow graph, it is possible to choose between full and incremental updates. The **full update** increases the major part of the current revision by one and sets the minor part to zero, for example from a revision 1.5 to a revision 2.0. On the other hand, the **incremental update** is increases the minor revision of the current revision by one and does not change the major revision, for example from a revision 1.5 to a revision 1.6.

2.7 Full update

Now that we have discussed revisions in detail let us look at the full update. Whenever a customer wants to see a data lineage, they run by default the full update. Full update first creates a new major revision and changes the current revision to it, which is done by calling `newRevisionScenario`. Then, it runs extraction and analysis scenarios for every user's technology. These scenarios run on the client and send parts of the resulting data flow graph to the server (as discussed earlier in Section 2.1). The server merges these parts and creates the resulting data flow graph. After all that, the full update ends with marking the current revision as committed so it cannot be changed anymore, which is done by calling `commitRevisionScenario`.

Let us examine how merging works, which is illustrated in Figure 2.8. At the bottom, we can see a merge input that was sent from the client. Whenever the client sends a node to the server for merging, it must send it with all its predecessors. We can see that this condition is satisfied in the example below, as the client is sending not only the two blue asset nodes for merging but also their predecessors, Resource and Super root. There is, in fact, an optimization that omits to send the Super root (as it is always in every merge input), but for simplicity, we will assume in this thesis that it is always sent.

On the left of Figure 2.8 we can see how the data flow graph looked on the server in revision 1.0 before the merging started. This merging is a part of a full update changing the current revision from 1.0 to 2.0. Merging always starts from the super root of the merge input and then continues recursively for all its children. Whenever merging a node this way, if there is the same node in the last revision, then the full update only changes its end revision to newMajorRevision.∞, which is 2.∞ in our example, which makes this node valid in the new revision as well. We can see that this happened in the example for nodes Resource, Layer, and Asset 1. On the other hand, if the currently merged node was not in the data flow graph before, it is added to the graph with start

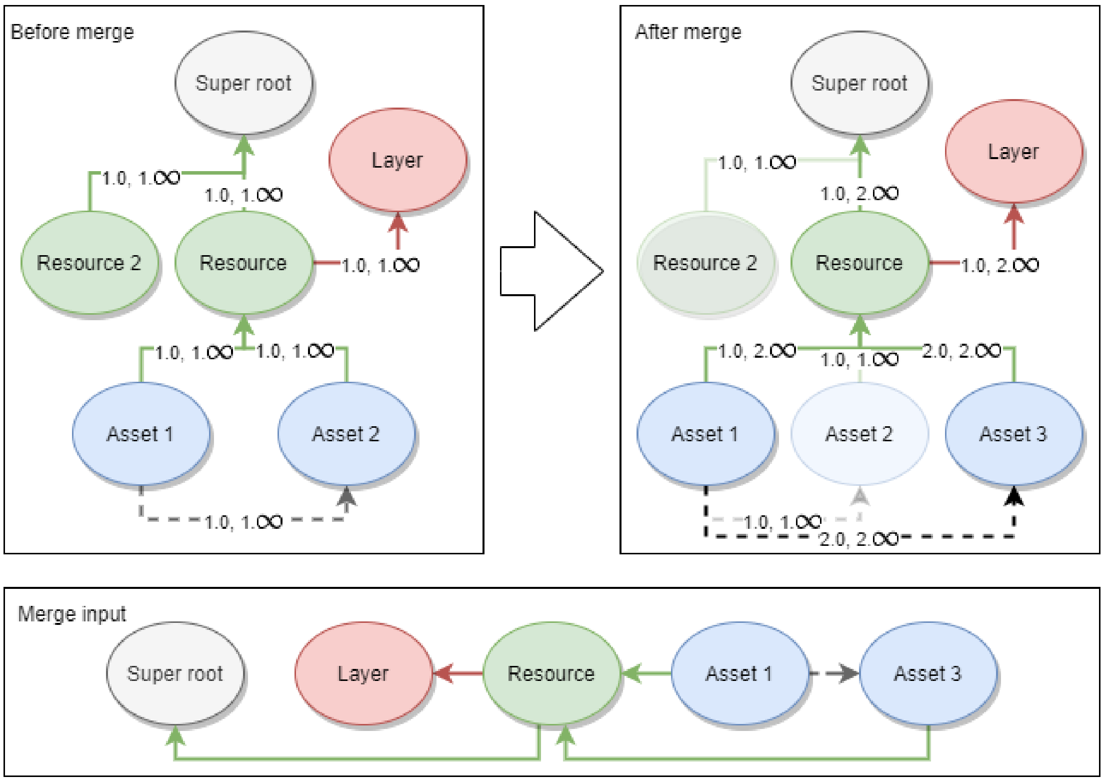

Figure 2.8: Full update example

revision newMajorRevision.0 and end revision newMajorRevision.∞, which is 2.0 and 2.∞ respectively in the example. This happened for the node Asset 3.

One last notable thing in Figure 2.8 is what happened with nodes Asset 2 and Resource 2. Because these nodes were not part of the merge input, their revisions were not updated, and they are no longer part of the new revision 2.0. They were deleted. This is a critical property of the full update. It deletes all the nodes that are not in any of its merge inputs. Therefore client needs to send all the nodes in the data flow graph in every revision again and again, even if they stay unchanged.

2.8 Incremental update

An incremental update is an alternative to the full update. It is different in two ways. First, it creates a new minor revision (e.g., from revision 1.0, it creates revision 1.1), which is an open revision instead of a major close revision as in the full update. This is done by calling `newMinorRevisionScenario`. The second difference is the usage of a modified merge algorithm.

Let us describe the modified merge algorithm. It starts like the full update by merging the merge input from the source root node by node. The difference is that the merge input is modified, so it contains specially marked nodes, which indicate that a change has been performed somewhere in the subgraph starting with this marked node. When such a marked node is reached during merging, the whole subgraph (starting from this marked node) is removed. The removal is performed simply by setting the minor part of the end revision of all nodes and edges to the number of the latest minor revision. For example, when creating revision 1.1, end revisions are set to 1.0. These objects are no longer valid in the new revision 1.1, and they are deleted (in the new revision).

Once the subgraph is removed from the main graph, the merging process continues from this marked node in the same way as the full update would. This means that the new version of this removed subgraph is merged into the main graph. If the removed node or edge is absent in the merge input, it will remain removed from the data flow graph. However, if the removed node or edge is present in the merge input, it is merged back by setting a minor part of its end revision back to the $.\infty$.

Let us look at the example of an incremental update in Figure 2.9, which shows the same situation as the previous example of the full update. However, the incremental update will be performed instead of the full update this time. Let us first discuss the only difference in inputs, which is that the Resource node in the merge input is marked. As already mentioned, it is marked because the client recognized there could be a change in the subgraph of the Resource node. And indeed, there's a change in its subgraph as it now contains the node Asset 3 instead of the node Asset 2.

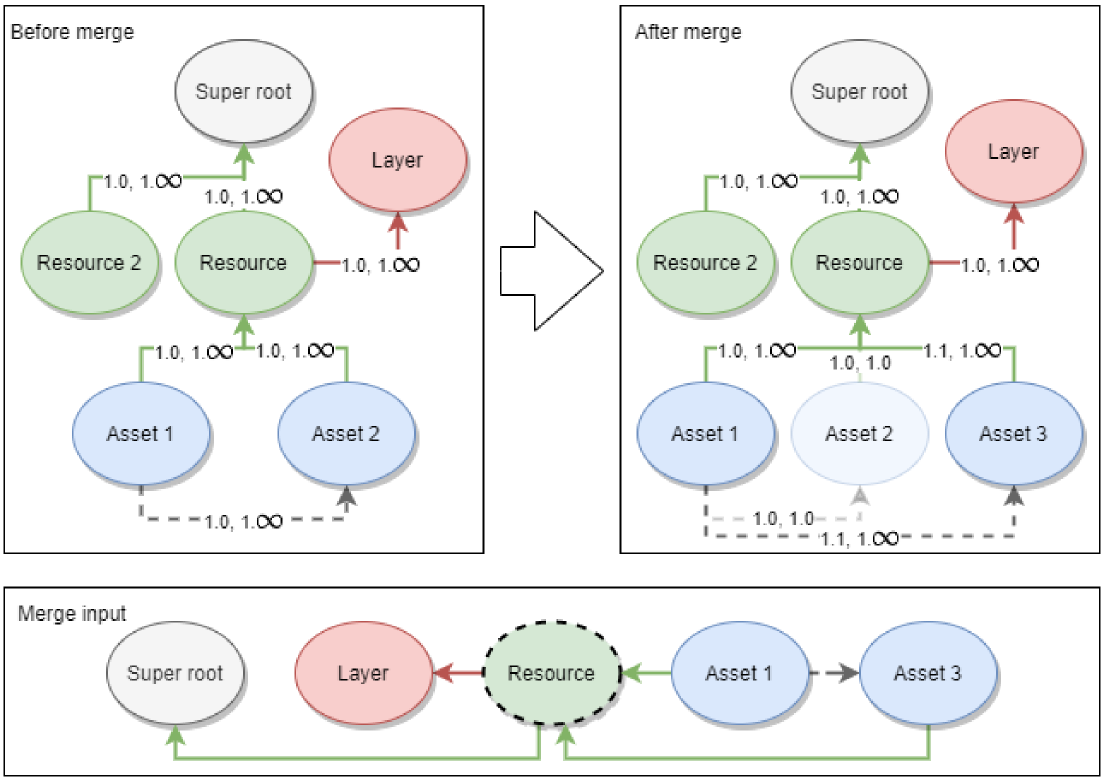

Figure 2.9: Incremental update example

In Figure 2.9, the current revision before the merge was 1.0. Therefore the new revision will be 1.1. Merging the merge input starts with merging the super root, which is the same as in the full update. Then the Resource node should be merged, but it is marked.

This means that its subgraph needs to be removed by setting the max revision of all its descendants to 1.0, which makes them not valid in the new revision 1.1. Then, the merge continues in the same fashion as the full update would by merging Asset 1 with the same node from the previous revision and adding the new Asset 3 node. Note that Asset 2 was deleted because it was not part of the new merge input. Its validity ends in revision 1.0, but the current revision is already 1.1.

On the left of Figure 2.9 is the data flow graph after the merge, which is somewhat similar to the one created by the full update. The vital difference is the node Resource 2, which was deleted by the full update but was not deleted by the incremental update. This is the most significant advantage of the incremental update that other nodes are not implicitly deleted just because they are not part of the merge input. Therefore, it is unnecessary to send every node in every revision again and again, but it is sufficient to send only those nodes that have changed since the last revision.

2.8.1 Merge input

Every merge input contains changed nodes, but it also contains lots of unchanged nodes. The first reason is that whenever merge input contains a changed node, it must contain all its predecessors. It is needed so that the server knows where to merge changed nodes. The second reason is that the merge input often contains data flow edges to other (often unchanged) parts of a data flow graph. For example, when one database procedure calls another one. In such a case, the merge input contains the data flow edge, one or more nodes of the called procedure, and of course, all their predecessors as well.

Consequently, the client needs to mark which nodes from the merge input are changed. And it is currently impossible to mark only changed nodes. The reason for that is that when a line changes in a database script, all the client can do is analyze the whole script again to create a new graph. Nevertheless, this graph does not contain any details about which edges and vertices are new, and the old version is stored only on the server merged into the whole data flow graph. The solution is to mark a root node of a subgraph in the merge input, which definitely contains all changed vertices and edges.

Currently, incremental updates support only changes in database scripts. The user manually selects changed database scripts, which should be part of the incremental update. Then, for each of these chosen scripts, the client analyzes the database script, which generates a data flow graph, and marks the BODY node of that script in the generated graph. Finally, all these graphs are sent to the server one by one, which merges them using the incremental update merge algorithm (described earlier in Section 2.8).

2.8.2 Example

Let us now look at an example of a merge input. We can see in Figure 2.10 a simple procedure with a newly added logging line of a code. A full update previously analyzed this procedure, but the user wants to perform an incremental update after this small change.

```
Procedure P(IN I, OUT O) {
  O = I;
  INSERT INTO LOG(Message) VALUES(I); // Newly added line
}
```

Figure 2.10: Changed procedure

Incremental update analysis for this changed procedure generates the merge input shown in Figure 2.11. We can see that the merge input contains whole procedure P, the smallest unit that can be analyzed independently. In the example, we can also see that for each node, the merge input also contains all its predecessors up to the super root. Another notable thing is that the graph contains procedure P nodes and the LOG table node, and the Message column node, even though these nodes are unchanged. Finally, the Body node is marked, which indicates changes in a subgraph with the Body node as its root node.

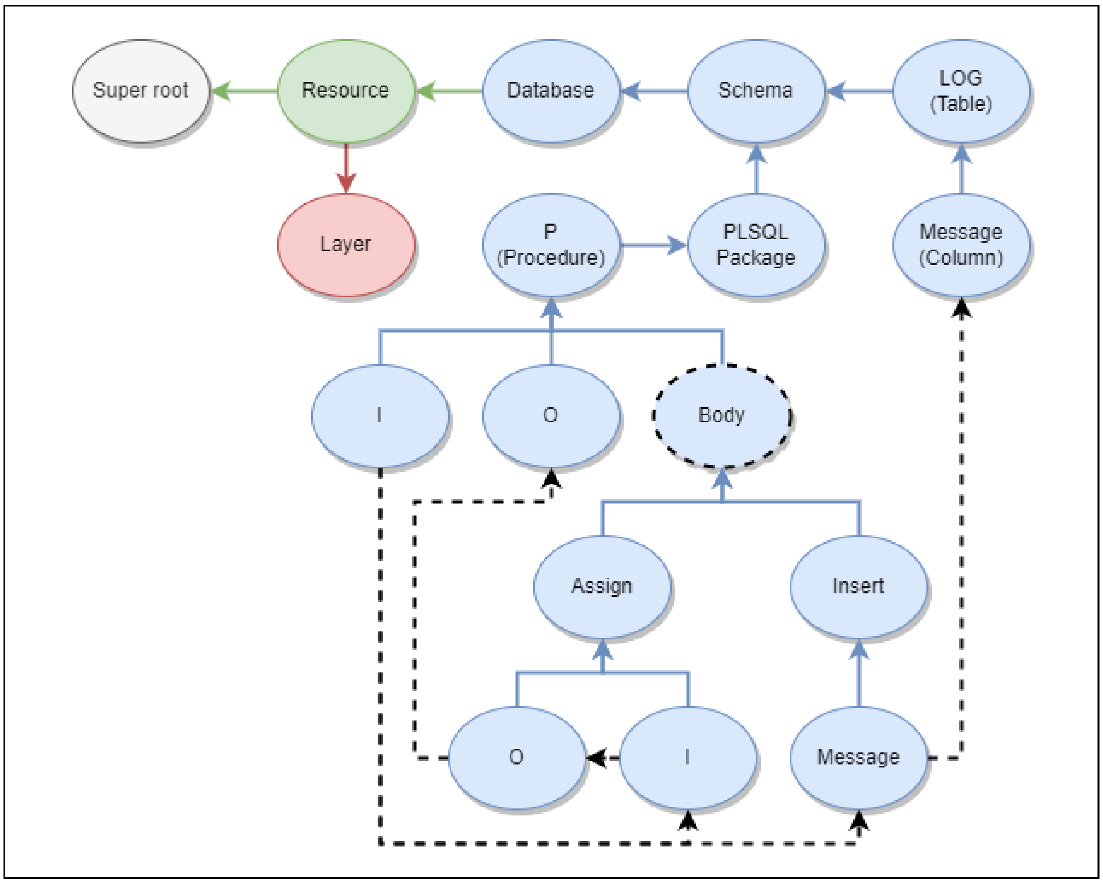

Figure 2.11: Merge input example

You might wonder why the client marked the Body node and not the procedure P node. The reason is that incremental updates currently do not support changes in procedure interfaces. If the client would mark the procedure P node, then the merge algorithm might delete edges created by other procedures. This would mean that the resulting data flow graph would be incorrect. Therefore, it is currently possible only to change a procedure's body using an incremental update.

2.8.3 Summary

The current design and implementation of incremental updates have a few significant limitations. First, it is currently possible to analyze only database procedures and not any other inputs, including changes in database tables, columns, or non-database source code. Second, when analyzing database procedures, there can be changes only in a procedure body, and its interface must remain the same. Third, if any of the previous limitations are not met, the user doesn't know about that and is instead presented with an incorrect data flow graph. Fourth, users must manually input changed database procedures, as they're not automatically detected.

As we can see, the current design has many limitations and is not useful for users. However, it is an excellent foundation with two key ideas of two-level revisions and an updated merge algorithm. Therefore, this thesis will use these ideas, build on them and remove or reduce their limitations.

2.9 Source code upload

Source code upload is an already existing feature that will be both used and modified by incremental updates. The client does not send only parts of the data flow graph to the server, but it also uploads all relevant source code files. Whenever uploading a new version of a source code file, the server creates a new Source node in the Source file graph (described in Section 2.5.2). Source code files are then sent asynchronously and stored on the server.

Each source node contains several properties. First, it contains a path to the source code file. This path is only relevant on the server (as files might be renamed on the server). Second, it contains a global (source code file) id. Path to the file on the client is sent when uploading the file, but it's not currently stored on the server.

This feature is currently used so that every stored procedure node has a reference to a relevant source code file. This reference is then used during visualization to show the procedure's source code. However, source code files are not uploaded to the server for many technologies. For these technologies, nodes in the Source file graph are still created for every source code file.

2.10 Post processing

After the server creates a data flow graph by merging all parts sent from the client, the server runs a post processing. It performs the following several changes in the whole data flow graph:

- For each edge type that should connect only leaves, check if all edges of that type in the data flow graph satisfy that condition. Fix edges connected to an inner node by changing the edge so it is instead connected to a leaf node that is a descendant of that inner node. If there is more than one such leaf node, then create a copy of that edge for each such leaf node.
- Create a reverse edge from the table node to the view node for each existing edge from the table node to the view node. This is not limited to views and not limited to the Oracle database.
- Few customers use some other post processing, but most do not. We will not consider them in this analysis.

2.11 Data dictionary description

A data dictionary is a read-only set of tables that provides metadata about a database. It contains a list of all objects in the database, such as all schemas, tables, columns, and relationships between them.

As already mentioned in Section 2.2, during a full update, data dictionaries are both extracted and analyzed. Each data dictionary is extracted into an H2 [5] database file, where each row represents one object from a data dictionary.

3. Requirements and analysis

The main focus of this chapter is to describe the solution's requirements, based on which we will later design the solution. However, before we get to the requirements, we need to analyze the problem and think about how we want to improve the current full update. Then this will then help us create the requirements and think about them much easier.

3.1 Problem overview

The currently used full update is already described in Section 2.7. It is summarized here in Figure 3.1, which describes the whole process of the full update. On the left, we can see lots of input files (e.g., customer's source code, configuration files). In the middle, we can see several scenarios, each of them processes multiple inputs and generates parts of the result dataflow graph. These parts are then merged into the single resulting data flow graph.

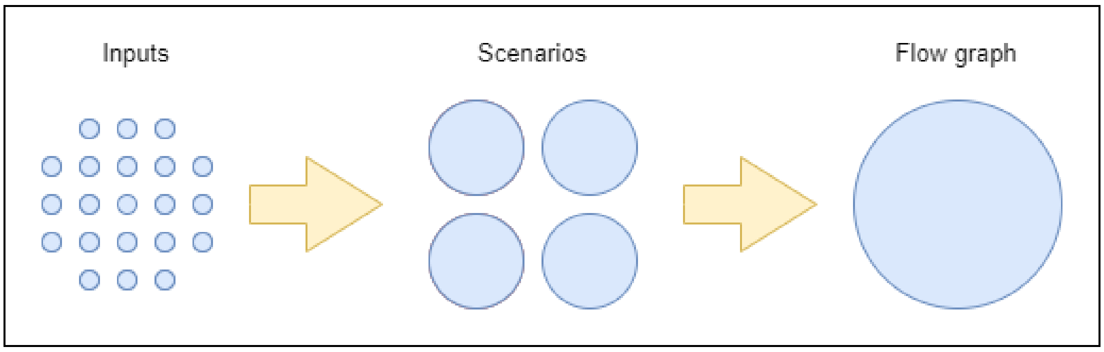

Figure 3.1: Full analysis explanation

After changing even a single input, a full update analyzes all the scenarios to create the updated flow graph. This means that the full update reanalyzes all the inputs, often taking many hours. The main goal of the incremental update is to analyze only a tiny fraction of these inputs while generating the same graph as the full update would.

The first issue is that the scenarios are too big, and there is no way to run only a part of a scenario. Analysis of any technology consists of only 1 to 4 scenarios. For example, a single scenario analyzes all the DDL scripts when analyzing an Oracle database. If we want to analyze a single DDL script, we need to analyze all the DDL scripts in the database.

The second issue is that the full update always deletes the whole flow graph created by the previous full update and then builds the new flow graph from scratch. We need not delete the previous flow graph because we want to reuse the unchanged parts of the flow graph from the last update. However, we still need to delete the obsolete parts of the flow graph locally. This is not currently possible, and we need to create a new way to do this.

3.2 Segments

The incremental update solves the first issue from Section 3.1 by dividing every scenario into multiple much smaller pieces; we call them **segments**. Every segment represents part of the scenario's work that can be analyzed independently. This helps to analyze only what is needed by creating the possibility to analyze small chunks of data. Every segment has to fulfill these two requirements:

- The segment can be analyzed on its own. (Requirement 1)
- We must (easily) recognize if any of the corresponding inputs have changed since the last analysis. (Requirement 2)

Segments allow performing incremental analysis differently from the full analysis. As we can see in Figure 3.2, the incremental analysis starts again with the inputs. First, it finds all the changed inputs. Based on them, it finds all the segments in which inputs are changed, which is possible due to the segment requirement 2. Then incremental analysis analyzes only these segments, which is possible due to the segment requirement 1. Finally, for each analyzed segment, it merges the newly generated flow graph into the resulting graph. However, before merging, the incremental analysis must first delete all the vertices and edges created by this segment in the last update. This deleting is partly covered in Jan Sýkora's thesis [2], but it still needs many changes to work correctly in our case.

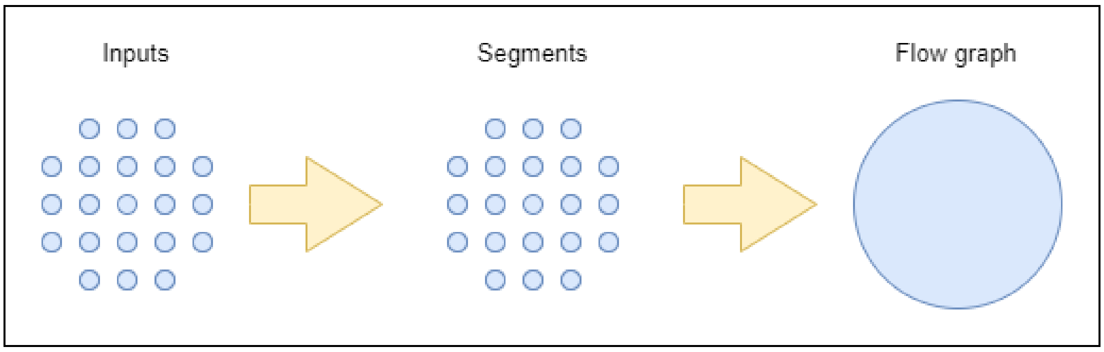

Figure 3.2: Incremental analysis explanation

Let us briefly mention what can be a segment and what cannot. The most helpful example of a segment is analyzing a single source code file. We can analyze a single source code file independently, and we can easily recognize if it has changed since the last analysis. This is our default choice for segments going forward. Another helpful example of a segment is analyzing a few source code files together. The last example we mention is a segment containing the analysis of all the inputs together. In this case, our incremental analysis would become the same as the current full update analysis. One negative example is one line of a source code file. It cannot be a segment because we cannot analyze it without the rest of the file.

3.2.1 Dependencies between segments

However, reanalyzing only the changed segments is insufficient because this simple approach would lead to many dataflow inconsistencies. One such example is a segment containing a database procedure with a “Select * from X” statement. Although this segment is unchanged, its analysis generates output different from the analysis in the last update. That’s because of a change in another segment, which contains the definition of the table X, which has a newly added column in the new version. Therefore we need to find out all the unchanged segments, which would generate different dataflow based on the changes in changed segments. Let us call this process **impact analysis**. Then we need also to reanalyze all these segments found by the impact analysis.

The goal of the impact analysis is to find transitive dependencies between segments. There are two types of dependencies:

1. Dependencies between whole segments. Dependency between whole segments X and Y means that if segment X changed, we also need to analyze segment Y because segment Y might yield a different result flow graph than in the last update. As soon as we know which segments have changed in the current update, we can use these dependencies to find all the segments which must be reanalyzed. Let us call these **L1 dependencies** (L1 stands for level 1). Note that this relationship has many-to-many cardinality.

2. Dependencies between segment parts - sometimes dependencies between whole segments are insufficient and need to be more specific. If we have L1 dependency between segments X and Y, only some changes of X mean that Y needs to be really reanalyzed. Only if a specific part of segment X was changed, then we need to reanalyze segment Y. This is precisely the meaning of **L2 dependencies**. L2 dependencies are between a part of segment X and the whole segment Y.

   Let us specify what part of a segment X means in this context. For example, it could be a line number in a specified source file, but it would be needed to parse every source file and track how these lines moved and changed whenever the file changes. Instead, let us utilize what the MANTA is best at and use a node in the generated flow graph as a source for L2 dependency. For example, we can have L2 dependency between a parameter of one database procedure x and some other procedure y. Then, procedure y will be reanalyzed whenever this parameter is changed or deleted. As we can see, this approach is much more accurate compared to L1 dependencies. Note that only knowing which segments changed is insufficient to use L2 dependency. We also need to know how the segments changed.

Note that we need to have all the dependencies precomputed from the last update. The reason is that if we have an unchanged segment, then we need to know all its dependencies without actually analyzing it. Otherwise, we would need to reanalyze all the segments during every incremental update. We can have dependencies between segments of the same technology or between segments of different technologies.

3.3 Use cases

There are two different use cases for incremental updates:

- One-technology incremental update - guarantees the correctness of all the nodes and edges in one specified technology, while it does not guarantee anything about other technologies. This is expected to be used mainly while setting up MANTA. We want to minimize analysis time because users often run analysis multiple times during a short time period to set up things properly.
- All-technologies incremental update - guarantees the correctness of all the nodes and edges in the graph. This is expected to be used in production.

3.4 General requirements

This section summarizes all requirements that incremental updates should meet:

1. Performing incremental update should leave CLI and Server in the same state as after running a full update.
   (a) This also means that the incremental update should generate the same data flow as a full update.
   (b) It is possible and expected that we would need to make exceptions to this rule in some cases.
   (c) The incremental update should also generate the same mapsTo and perspective edges as the full update.
2. This project should define theoretical conditions that are necessary (and also sufficient) to perform an incremental update according to the first requirement.
3. The system should recognize what changes have been made since the last update. This includes input files and all other inputs of full analysis.
   (a) This includes, for example, extracted data, manually provided data, manual dictionary mappings, scanner-specific mappings (Informatica parameters, Talend custom variables, StreamSets runtime values, Cognos search path mapping), etc.
4. The incremental update should analyze only changed segments and segments found by impact analysis.
5. Incremental update can be executed by running workflow run-(technology)-incremental for every supported technology, e.g., run-oracle-incremental.
   (a) These workflows will contain all the scenarios from the extraction and analysis phase of their technologies.

   (b) These scenarios will newly contain a switch to choose between full and incremental update.
   (c) Each scenario can run an update for one connection. Process manager allows users to choose multiple connections and run a workflow for all of them at once.

6. This project will not deal with incremental data extraction.
   (a) Extraction phase of the full update will be used instead.
   (b) Incremental update feature will have no impact on the extraction phase.

3.5 Impact analysis requirements

The impact analysis should meet these requirements:

1. Incremental update should execute impact analysis to find transitive dependencies.
   (a) The impact analysis should find all unchanged segments generating data flow different from the last analysis.
   (b) Impact analysis is allowed to find some false positives (i.e., unchanged segment in which data flow is the same in the new update).
2. Reporting technologies should use L1 dependencies.
3. Database technologies should use L2 dependencies.
4. L1 dependencies will be stored on the client, most likely in a relational database.
5. L2 dependencies will be represented by graph edges and stored on the server.
   (a) They will be stored either in the dataflow graph or in a separate graph.
6. We need to ensure 1:1 mapping of segments between client and server.

3.6 Prototype requirements

This project should implement a prototype following these requirements:

1. Prototype should support incremental updates for Oracle database technology.
   (a) This also means creating workflow run-oracle-incremental.

2. Prototype should be able to handle both changes in implementation and interfaces.

3. Prototype should handle all 3 Oracle analysis scenarios (mentioned in Section 2.2).

4. Prototype should run in a development environment.

5. Prototype should only support the one-technology use case.

6. Prototype will not implement parallelization of its merge algorithm.

3.7 Non-functional requirements

This project should also meet the following non-functional requirements:

1. Both design and implementation should focus on the performance of incremental updates (i.e., time of execution).

    (a) Analysis should come up with a suite of tests to measure the performance.

2. Focus on performance also means focus on the possibility of parallelization of the analysis. This includes:

    (a) interscenario parallelization - multiple incremental scenarios running in parallel for different connections or technologies,

    (b) intrascenario parallelization - multiple inputs analyzed at once from the same connection,

    (c) merge algorithm parallelization - design merge algorithm in synchronization with Tomáš Polačok's thesis [6] for parallelization of full update's merge algorithm.

3. The project should include both technical and user documentation.

4. The project should use minor updates (and other fitting ideas) from Jan Sýkora's thesis [2].

4. Design

This chapter describes the design of the new solution. First, we explore the flaws of the previous merge algorithm and design an improvement for all of them one by one. Then, we analyze other aspects of the new merge algorithm and design a solution for them. After that follows a summarized description of the new merge algorithm with a detailed example. The chapter ends with a description of a client algorithm that calls and orchestrates a previously designed merge algorithm.

4.1 Simplified client algorithm

Before designing the new merge algorithm, which runs mainly on the MANTA server, we need to describe how the MANTA client orchestrates the whole incremental update. It is done by performing a client algorithm, and for now, we will describe only its simplified version. The reason for that is that although we need this client algorithm to understand the description of the merge algorithm better, the full version of the client algorithm described later in Section 4.8 needs knowledge of the new merge algorithm, which has not been described yet.

The client algorithm starts with a user using MANTA graphical interface. Users can choose which technologies should be analyzed and if they should be analyzed using a full or an incremental update. Let us for now assume only the One-technology use-case mentioned in Section 3.3, which means that the user has changes only in one technology. In case the user chooses the incremental update, the client will newly perform these steps, comments are in grey:

1. Run extraction phase scenarios for the changed technology. This step is the same in full update, as incremental updates should use the same extraction.
2. Start an incremental update by calling newMinorRevisionScenario. This is an already existing scenario that correctly initializes graph database structures for an incremental update.
3. Find all changed segments and add them to a queue Q. Changed segments are segments with a changed input (described in Section 3.2).
4. Use L1 dependencies to find unchanged segments that generate different output because of changed segments and add them to the queue Q.
5. While the queue Q is not empty:
    (a) Pop segment S from the queue Q, analyze the segment, and send created data flow graph to the server to merge it.
    (b) Receive from server a list L of segments that need to be reanalyzed because of changes in segment S and push them to the queue Q. Server uses L2 dependencies to find segments in the list L.

   (c) Use L1 dependencies to find all unchanged segments that generate different output because of changed segments in the list L and add them to the queue Q.

6. Commit the incremental update by calling commitRevisionScenario. This is an already existing scenario that commits all the changes.

Note that we can use L2 dependencies only while merging the result data flow graph. That is the only moment when we reliably know which parts of merged segments have changed since the last update, which is the information needed to know which L2 dependencies should be used.

4.2 Merge algorithm improvements

This section describes changes and improvements for the merge algorithm used by incremental updates. First, we recapitulate the previous version of this merge algorithm. Then we go one by one through all of its issues. We introduce each issue with an example code snippet and a figure depicting an incorrect merge. After that, we present a solution to this issue and a figure showing a correct merge.

4.2.1 Previous algorithm

The previous merge algorithm for an incremental update is described in Section 2.8. Let us summarize it into individual steps, so we can easily refer to them:

1. Client - analyze all changed source code files and mark a body node of every changed procedure. For each analyzed procedure, send its generated data flow graph separately to the server to merge.

2. Server - for each graph sent from the client, merge it node by node with the data flow graph stored on the server. Additionally, whenever merging a marked node:

   (a) Remove the marked node's previous subtree from the data flow graph (all the vertices and adjacent edges).

   (b) Merge vertices from its new subtree with the data flow graph.

   (c) Merge edges from its new subtree with the data flow graph.

4.2.2 New merge algorithm

The main idea of the new merge algorithm is to improve the previous one by using as much already **existing information** in the data flow graph as we can.

We want this to save both time and space of running an incremental update and also to save implementation time. Otherwise, there would be too much to code for every technology that would like to use incremental updates.

This means that we use information from dataflow edges to capture dependencies between segments. Whenever analysis of segment X created a dataflow edge adjacent to a vertex created by another segment Y, this edge tells us that segment X is dependent on Y. Technically speaking, we can treat such a dataflow edge as an L2 dependency edge from its source node in the segment Y to the segment X.

Sometimes information from dataflow edges will not be sufficient. We will create new L2 dependency edges describing these dependencies in such cases.

Aside from dataflow edges, we need one more thing from the analysis. We need to know for every vertex and edge which segment created them; let us call it a **source segment**. For now, let us assume that every vertex and edge has a method `getSourceSegment`, which returns the source segment. We'll discuss its implementation later in Section 4.4.

The merge algorithm is merging a new reanalyzed version of a segment. We want to create as many edges correctly as possible based on data from the currently merged segment. However, there will often be some edges about which we cannot decide based only on data from the currently merged segment (e.g., if an edge created by another segment should be in the new version). For these edges, we will return their source segments to the client, where they are added to the reanalyzation queue. This causes that these segments will be later reanalyzed as well. That is when we will have enough information to decide about those edges correctly. We've already seen this in step 5b of simplified client algorithm in Section 4.1.

4.2.3 Segment root node

Analysis of every segment creates a graph that is sent to the server and merged into the data flow graph. This graph is called a merge input, and we have already discussed it in Section 2.8.1. Merge input contains both nodes and vertices with a source segment equal to the currently analyzed segment and some other nodes and vertices (e.g., predecessors of that nodes). The following invariant is true for every merge input created by analyzing a segment S, “If a node has a source segment S, then all its children have the same source segment S.”

Let us now look at the example of the same merge input as in Figure 2.11. Figure 4.1 depicts a merge input of a segment P containing a procedure P. This merge input contains blue vertices and edges with a source segment P. It also contains grey vertices and edges without a source segment. We can see in the example that the invariant about blue vertices holds.

As a consequence of the previous invariant, all blue nodes in Figure 4.1 create together one blue subtree (i.e., a subgraph that is a tree). The subtree has a blue root node, which we will call a **segment root node**, denoted by the red line in

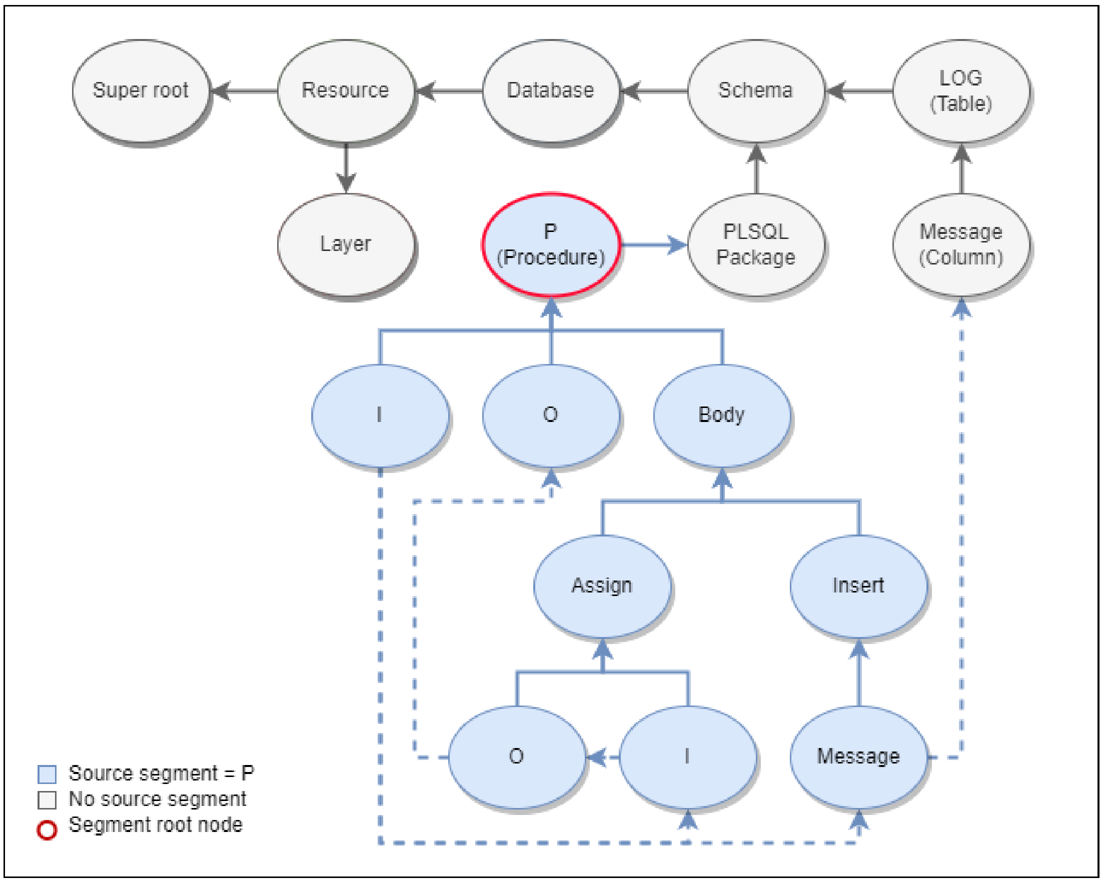

Figure 4.1: Source segment example

the example. In general, every merge input has 0-n blue subtrees and the same number of root nodes, most typically just one.

Figure 4.1 also contains grey nodes without a source segment. These nodes will be discussed later in Section 4.3. Until then, all examples will omit these nodes without a source segment for a simplification.

Now let us incrementally update the previous merge algorithm described in Section 4.2.1. Next sections (from Section 4.2.4 to Section 4.2.9) describe issues of the previous merge algorithm one by one. Each such section first contains an issue description with an example. Then, it contains a solution to this issue, again with an example. Lastly, there is also sometimes a discussion about the implementation of the solution.

4.2.4 Changing the marked node

Issue: The previous merge algorithm (mentioned in Section 4.2.1) in step 1 marks body node of a database procedure when that procedure is changed. As previously discussed, this means that incremental updates work incorrectly whenever there is a change in a procedure interface. Let us examine this behavior on an example in Figure 4.2. This example contains a database procedure P, which had a single argument I in the old revision, but it was changed, so it currently has no arguments in the new revision. And user wants to propagate this change to the data flow graph using an incremental update.

```
// Old revision
Procedure P(IN I) { ... }
// New revision
Procedure P() { ... }
```

Figure 4.2: Changing the marked node code example

On the left side of Figure 4.3 we can see how the data flow graph looked in the old revision. It contained the procedure P node with two children nodes Param I and Body. At the bottom of the example is a merge input created by the client, which contains a marked Body node. It does not contain the Param I node as it is deleted in the new revision. We can see the data flow graph after the merge on the right side, which incorrectly contains the Param I node. That's because the previous merge algorithm marks the Body node, which leads to the server deleting the Body node and its whole subgraph during the merge. However, the Param I node is never deleted as it is not a part of the deleted Body node subgraph.

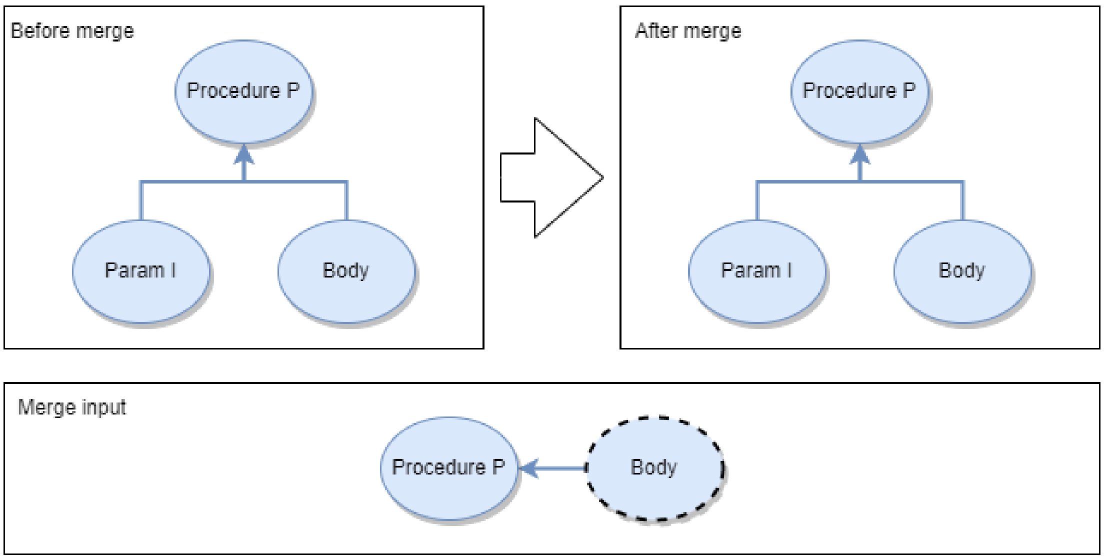

Figure 4.3: Changing the marked node example issue

Solution: Solution to this issue is to change which node client marks. Newly, when merging a segment S, the client marks all segment root nodes of segment S (defined in Section 4.2.3). And because every node with a source segment S is either a segment root node of segment S or its descendant, the merge algorithm will surely delete all nodes with a source segment S.

Figure 4.4 contains the same example as previously shown in Figure 4.3, but with a fixed merge input. This time, the Procedure P node is marked instead of the Body node. Note that the Procedure P node is a segment root node. Therefore, the Procedure P node and its descendants are removed during the merge. And after merging the merge input back, the Param I node is not in the resulting data flow graph because it is not in the merge input. So this change successfully fixed the issue when the Param I node stayed incorrectly in the graph.

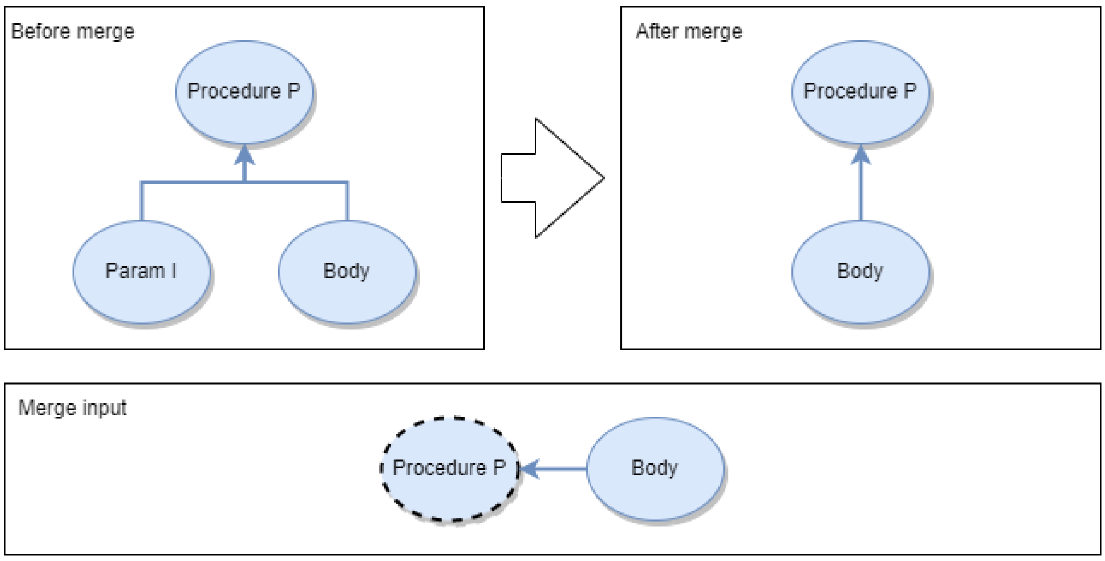

Figure 4.4: Changing the marked node example solution

4.2.5 Edge restoration

Issue: Imagine merging some segment P. Then all the edges with source segment different from P that are adjacent to any vertex V with source segment P are removed in the step 2a of the previous merge algorithm. However, for every such removed edge, if its source segment is unchanged and vertex V was not removed, then all these removed edges should be in the new revision. Therefore their deletion creates a data inconsistency.

Let's describe this on an example from Figure 4.5. We can see a stored procedure P and some other procedure S, which calls the procedure P. Procedure S is unchanged in the new revision and procedure P has some changes in its body. For our example, it is not important what exactly these changes are.

```
// Old revision
Procedure P(IN I, OUT O) {}
Procedure S() { ... P(I, O) ... }
// New revision
Procedure P(IN I, OUT O) { ... }  // Changed
Procedure S() { ... P(I, O) ... } // Unchanged
```

Figure 4.5: Edge restoration code example

To continue with the example from Figure 4.5, we can see on the left of Figure 4.6 the important part of the flow graph stored on the server before merging the new revision. We can see that the analysis of procedure S created two dataflow edges to nodes in the P subtree. Note that these edges are red in the example because their source segment is procedure S, as they were created during an analysis of the segment S. On the bottom of Figure 4.5 is a merge input, which contains a new version of segment P because only this segment changed in the new revision. The Procedure P node is marked. For simplicity, this example omits a body of the Procedure P, which would be changed in the merge input.

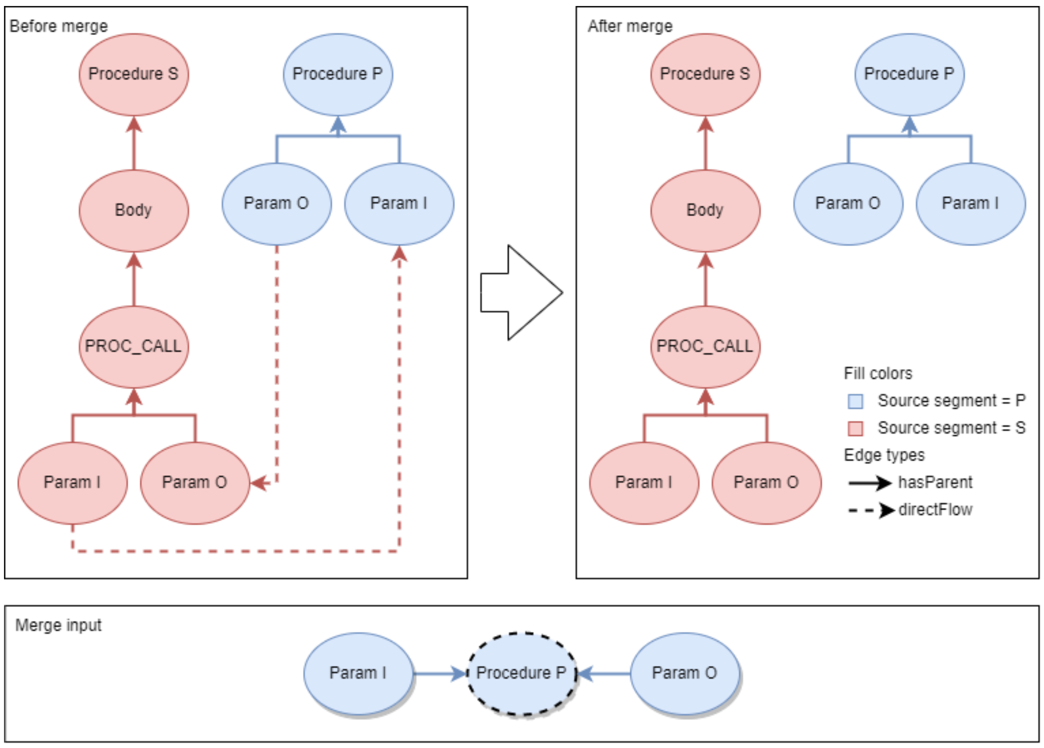

Figure 4.6: Edge restoration example issue

On the right of Figure 4.6 we can see (the important part of) the flow graph stored on the server after merging the new revision. When merging the merge input using the previous merge algorithm, it removes all nodes and edges in the subtree of P (in step 2a). This includes both red direct flow edges in the example created by the procedure S, which are removed. Because S is unchanged, S is not reanalyzed, and these edges are not created again. Therefore they are not in the new data flow graph, although they should be.

Solution: After merging any vertex (in the step 2b of the previous merge algorithm), the new merge algorithm will restore all edges adjacent to this vertex. Restoring an edge means changing its state back to valid in the current revision by changing its end revision. However, the algorithm will restore only those edges that were removed in step item 2a of the previous merge algorithm. There is an exception for edges with a source segment equal to the one currently being merged, which is discussed later in Section 4.2.6.

Let us get back to our example, which has the same graph before the merge and the same merge input as the previous example in Figure 4.6. A difference is in the used merge algorithm, which now uses edge restoration. We can see in Figure 4.7 on the right side how the flow graph on the server will look like using edge restoration. When merging the new version of the procedure P, during merging nodes `Param O` and `Param I`, both red direct flow edges created by the procedure S are restored. This change fixes the issue presented earlier.

Implementation: To implement this feature correctly, we need to remember all the edges created by other segments that were removed in step 2a. For any removed edge e between nodes u and v while merging a segment X, we add these

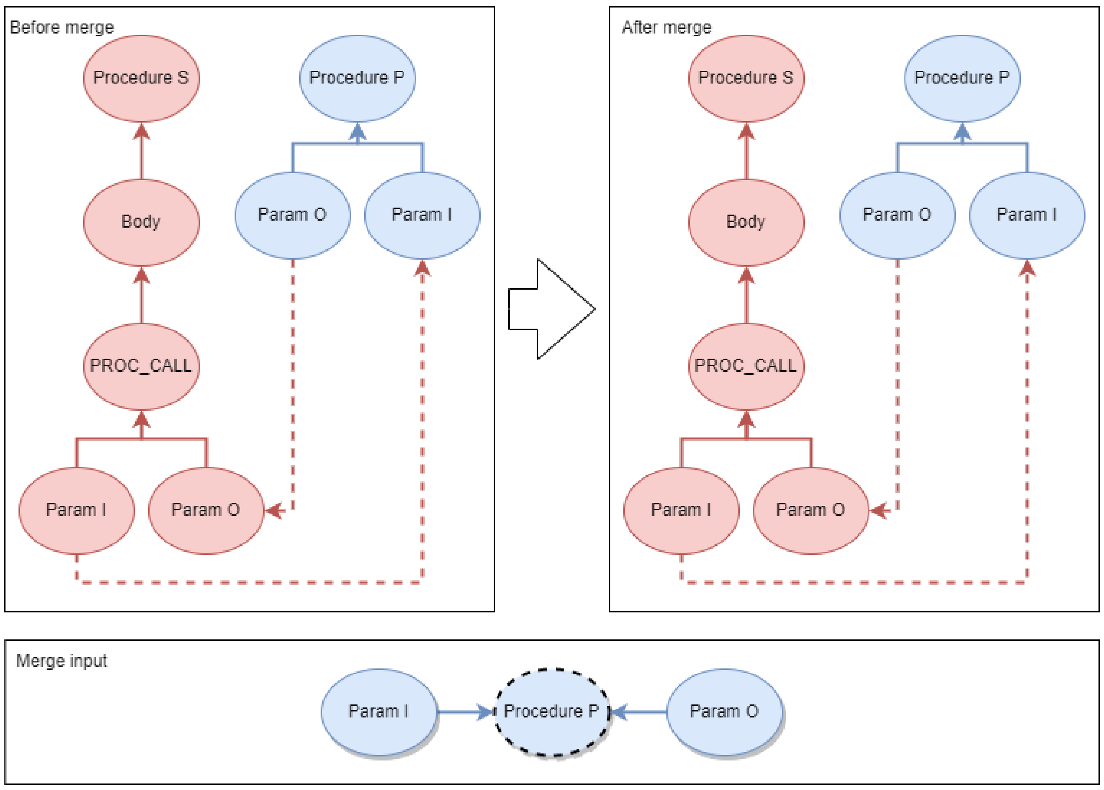

Figure 4.7: Edge restoration example solution

edges into a hash map H. The hash key will be node u or v, whichever has source segment X, and the hashed value will be the edge e. This map can have multiple values with the same key. We need this hash map because we could not otherwise tell the difference between an edge removed from the data flow graph because of our current merging and an edge removed from the graph by some other changes in the last revision.

Whenever the algorithm merges any vertex, it uses this vertex as a key into the map H and finds all edges adjacent to this vertex. If there are any such edges, the new merge algorithm restores all of them. This is also why this data structure is a map, to find all these adjacent edges quickly.

4.2.6 Edge restoration 2

Issue: Now we know that when merging a node, the new merge algorithm needs to restore removed edges. However, it should not restore all edges. If it would, it would create a data inconsistency. Let us examine it on an example in Figure 4.8, which contains two procedures, P and Q, that are the same in both old and new revisions. The example also contains the procedure S, that called the procedure P in the last revision, but now it calls the procedure Q in the new revision.

For this example, we can see how the data flow graph looked in the old revision on the left side of Figure 4.9. Most importantly, it contained a data flow edge

```
// Old revision
Procedure P(IN I) { ... }
Procedure Q(IN I) { ... }
Procedure S() { P(I) }
// New revision
Procedure P(IN I) { ... } // Unchanged
Procedure Q(IN I) { ... } // Unchanged
Procedure S() { Q(I) } // Changed
```

Figure 4.8: Edge deletion code example

between parameters of procedures S and P because the procedure S called the procedure P. At the bottom of the example, we can also see the new merge input. It newly contains a data flow edge between parameters of procedures S and Q. If the merge algorithm would restore all created edges, then the edge between procedures S and P would be restored. We can see that on the right side of Figure 4.9. This is, of course incorrect, because the procedure S does not call the procedure P anymore.

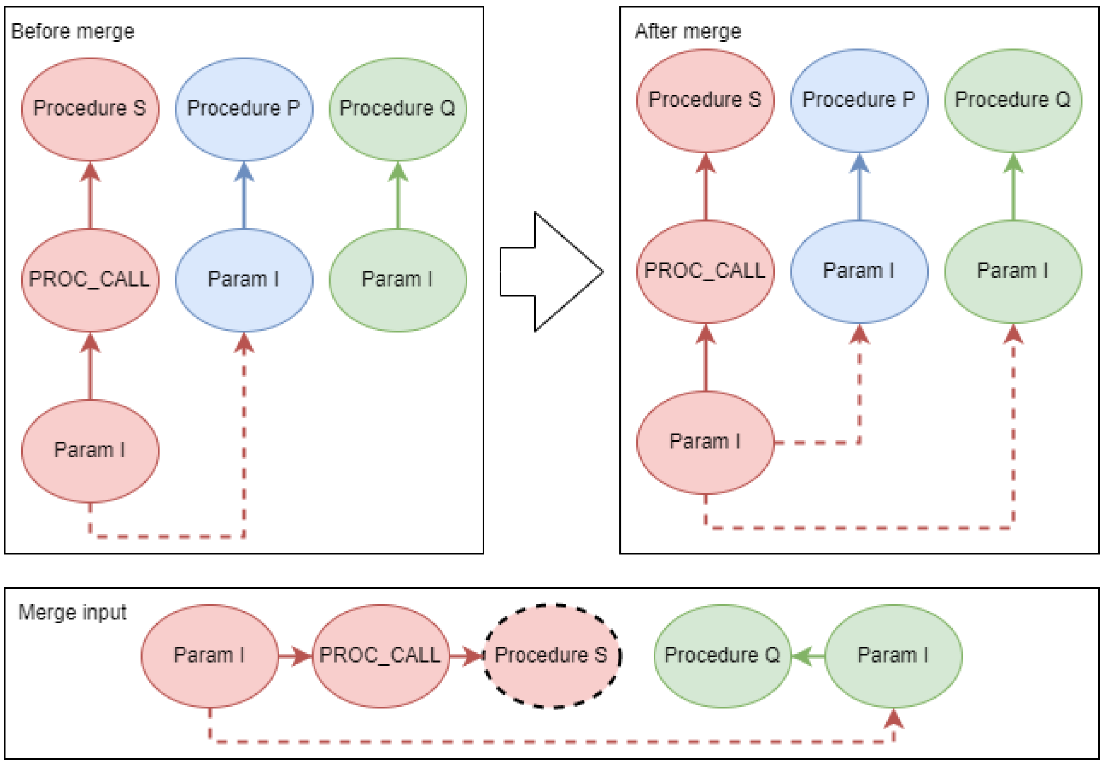

Figure 4.9: Edge restoration 2 example issue

Solution: Whenever merging a merge input created by an analysis of a segment X, we should restore only edges created by other segments. Edges created by the segment X do not need to be restored because if we delete an edge created by the segment X which should be in the new revision, it will be merged back anyways in the step item 2c of the previous merge algorithm. That is because edges with source segment X will always be part of a merge input created by analyzing the segment X.

In our example, we can see the solution in Figure 4.10. It contains the same data flow graph before the merge and the same input as the previous example in Figure 4.9. However, after not restoring edges created by the merged segment S, we can see on the right side that the resulting data flow does not contain the data flow edge between procedures S and P anymore. This fixes the issue, and the resulting data flow graph is correct.

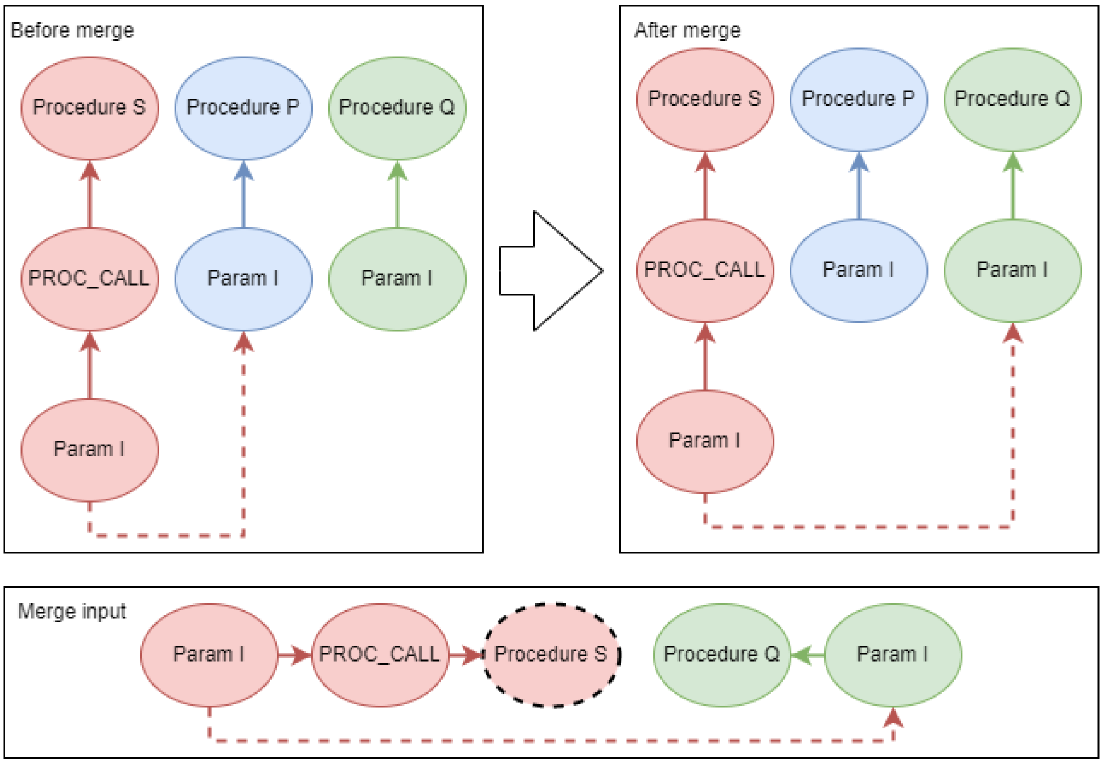

Figure 4.10: Edge restoration 2 example solution

4.2.7 Node removal

Issue: During merging of any segment X, there is only one situation that leads to deleting an edge created by some other segment. This happens when we delete a node adjacent to some edge e created by a segment other than X. By deleting the node, we mean that it existed in the last revision and it does not exist in the current revision (i.e., it was removed in the step 2a of the previous merge algorithm and not added back in the step 2b). The reason to delete the edge e is that we deleted one of its adjacent nodes, and we do not know if we should replace this node with another node or delete the edge. We do not know it because we do not know anything about the segment that created that edge at the moment. The only way to know would be to reanalyze the source segment of the removed edge e.

However, if we would delete the edge e and would not do anything else, it could create data inconsistencies in the graph. This is exactly what the previous merge algorithm does. We can see an example in Figure 4.11, which has a procedure P with a single parameter. The only change in the new revision is the renaming of the parameter from I to J. Then the example has a procedure S that calls the procedure P.

```
// Old revision
Procedure P(IN I) { ... }
Procedure S() { ... P(K) }
// New revision
Procedure P(IN J) { ... }  // Changed
Procedure S() { ... P(K) } // Unchanged
```

Figure 4.11: Node removal code example

On the left side of Figure 4.12 we can see that before the merge graph contained a data flow edge between parameters of procedures S and P created by analysis of the procedure S. Because only the procedure P changed in the new revision, the merge input on the bottom contains a graph of the procedure P with the new Param J node. However, the Param I node was deleted during the merge, and the data flow edge was deleted with it. And because the merge input does not know anything about the procedure S, that data flow edge is not merged back, so it stays deleted. We can see this on the right side of the example, which is incorrect.

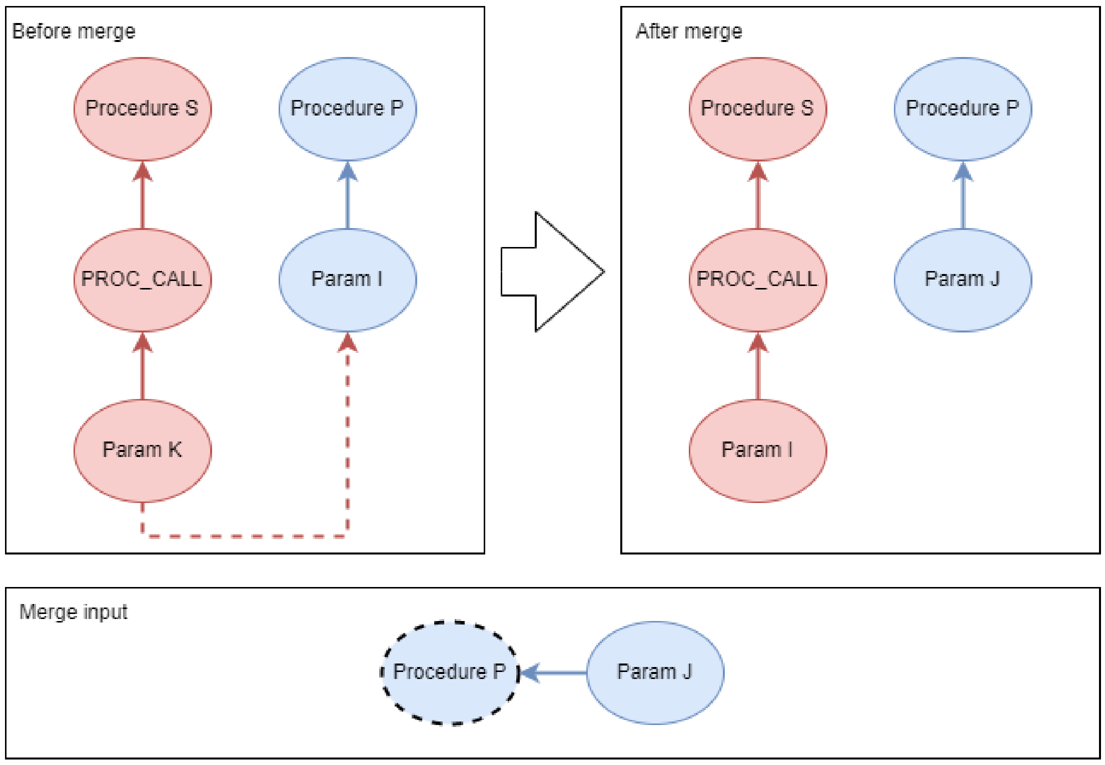

Figure 4.12: Node removal example issue

Solution: To fix this issue, whenever the new merge algorithm deletes an edge created by some other segment Y during a merge, then it sends this segment to the client after the merge is completed. Afterward, the client adds all these segments to the reanalyzation queue (in step 5b of the simplified merge algorithm). Thus later in the same revision, segment Y will also be reanalyzed, which will recreate deleted edges if needed.

This means that after the merge of the segment P in the previous example (in Figure 4.12), the server sends back to the client the segment S. The client then reanalyzes the segment S in the same revision and sends its merge input to the server. We can see this merge input on the bottom of Figure 4.13. We can see that it contains the marked Procedure S node as well as the data flow edge. This edge is missing in the data flow graph before the merge, as we can see on the left of the example. However, merging this merge input will recreate this edge between the two procedures, as we can see on the right side of the example. This fixes the issue in the same revision, so the user will not notice that the edge was missing for some time.

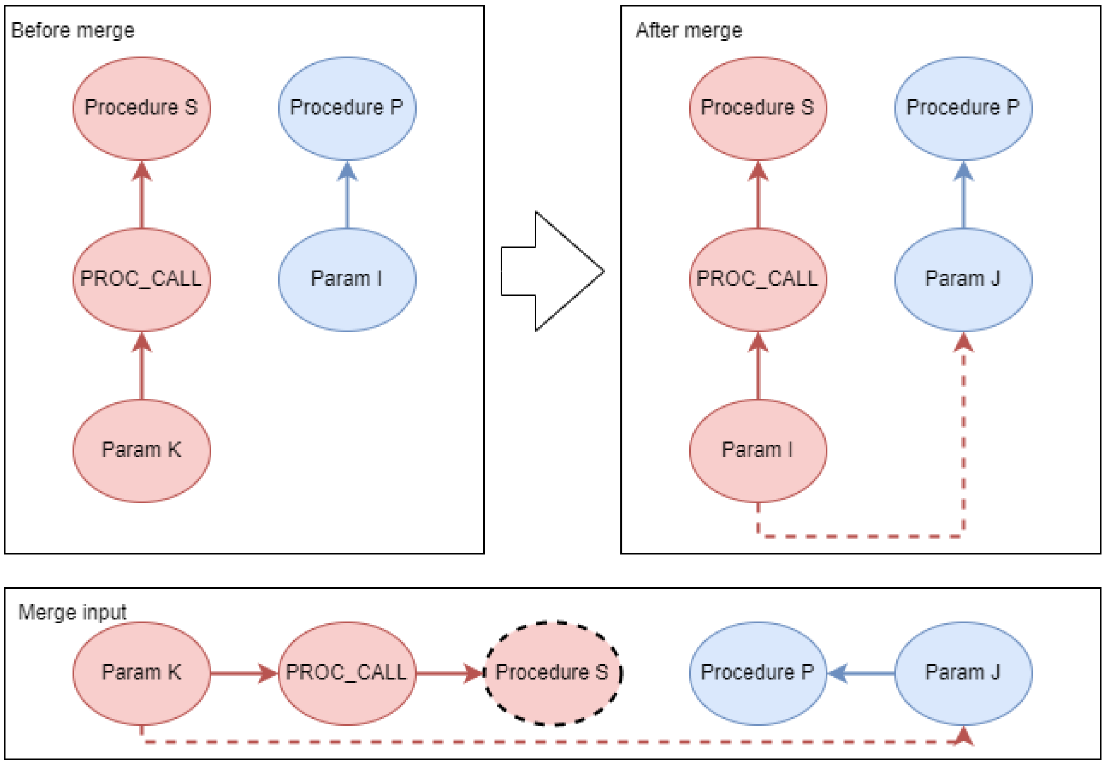

Figure 4.13: Node removal example solution

This approach will generate some false positives, reanalyzations of some segments that do not need to be reanalyzed. That is because these segments will create the same graph as there already is. The good thing is that one false positive cannot generate another one because it will not change anything in the graph.

Implementation: The last question is how to implement this feature. We need to find all the edges that were removed in this revision. The easiest way is to use the hash map H, which was introduced earlier in Section 4.2.5. This map contains all edges created by other segments that were removed by step 2a of the previous algorithm. During the edge restoration, many of these edges are restored (i.e., added back into the graph). Whenever we restore an edge, we also delete this edge from the map H. This does not create any issue for the edge restoration algorithm because that algorithm restores every edge at most once anyways.

After merging a segment X ends, what is left in the map H is exactly what we need. That is all the edges with source segment different from the X removed by this revision. The last step is to use these edges to find their source segments and send them back to the client for reanalyzation.

4.2.8 Node creation

Issue: The next question is what to do after the merge algorithm creates a new node. There could be some other segment X that would create an edge to this node in this revision. However, this edge does not exist because that node did not exist when the segment X was analyzed. And if the segment X did not change in the current revision, this edge will be incorrectly missing in the data flow graph.

One such example is in Figure 4.14, where we have an unchanged procedure P that selects all columns from a table Persons. The table Persons is changed in the new revision, and it newly has an additional column Y. However, the procedure P is unchanged in the new revision.

```
// Old revision
CREATE TABLE Persons (X int); // Old revision
Procedure P() { // Unchanged
  DECLARE V Persons%ROWTYPE;
  Select * Into V FROM Persons;
}
// New revision
CREATE TABLE Persons (X int, Y int); // Changed
Procedure P() { // Unchanged
  DECLARE V Persons%ROWTYPE;
  Select * Into V FROM Persons;
}
```

Figure 4.14: Node creation code example

We can see how a data flow graph looks for this example before the merge on the left of Figure 4.15. It contains one table column X, and the variable V also has one child X. The merge input on the bottom of that example contains a new version of the table Persons with two columns.

The previous merge algorithm does not perform any particular action after a node is created, which creates a data inconsistency, as we can see on the right side of Figure 4.15. In the new revision, table Persons changed, and therefore it was reanalyzed and has a new column Y. However, procedure P is not reanalyzed in the revision (because it is not changed), and therefore its variable V has only one column, which is incorrect.

Solution: There is no easy solution to fix this problem. This is the situation where it is tough to find out dependencies between segments only from dataflow

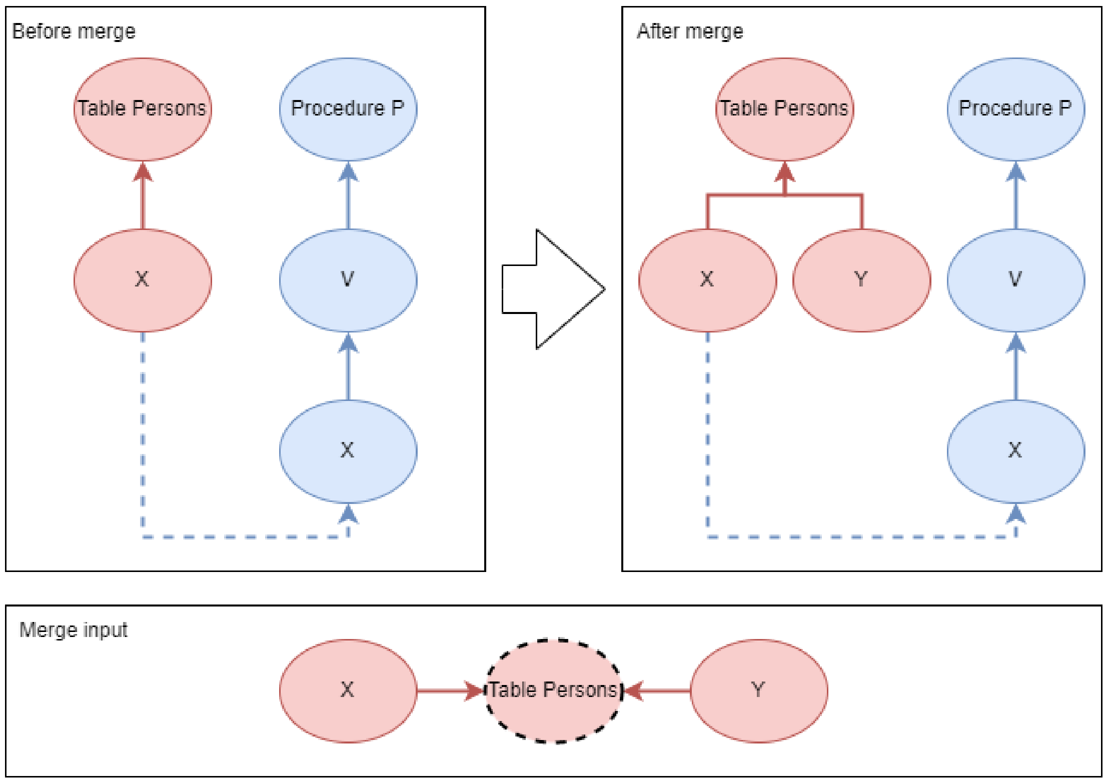

Figure 4.15: Node creation example issue

edges. It is generally impossible to know which segment would create an edge connected to a newly created node, if any.

We cannot find out all dependencies only from dataflow edges. Therefore we create a new type of edge called **dependency edge**. This edge leads from a segment X into a node Y. There are multiple subtypes of the dependency edge, each with a slightly different meaning:

- Subtype ReanalyzeOnChildCreate - reanalyze the segment X whenever a child of the node Y is created.
- There could be more types based on later discovered issues.

The issue that was mentioned earlier in Figure 4.15 can be solved by adding a dependency edge of type ReanalyzeOnChildCreate. All we need to do is whenever we have a “Select * From T” statement in a procedure P is to create this edge between the segment of the procedure P and a node representing the table T.

This is shown in Figure 4.16 below, where the dotted edge represents the new dependency edge. In this example, first will be reanalyzed segment of table Persons because this table changed. During that, when the node Y is merged, the new merge algorithm will check its parent for the presence of the ReanalyzeOnChildCreate edge. After it’s discovered, the source segment of this edge will be sent to the client as a segment to reanalyze. This means that a segment with the procedure P will be reanalyzed later during the same revision. Furthermore, during its analysis, the node Y will be correctly created, as we can see on the right side of Figure 4.16.

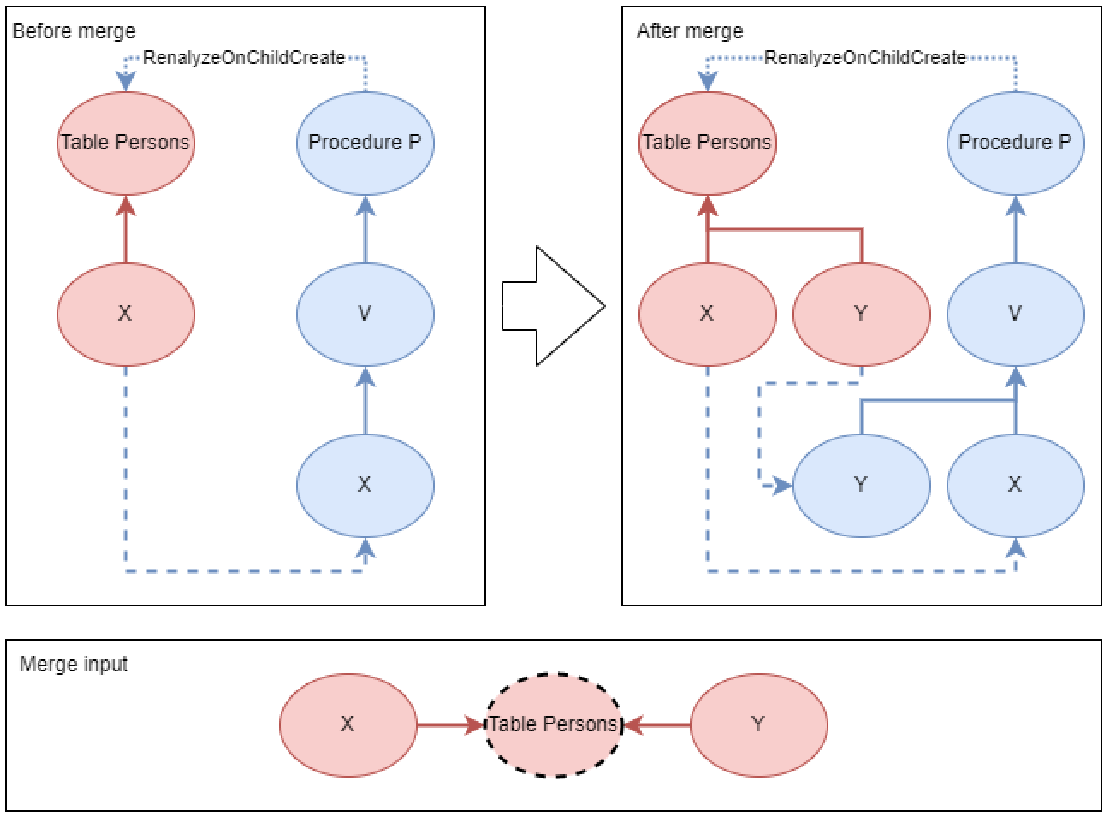

Figure 4.16: Node creation example solution

4.2.9 Shared segments

Issue: We've so far assumed that there is only one source segment for every edge and node. This is true for edges, but it is not always true for nodes. One such example is in Figure 4.17, which contains two files. The code in the first file createMyType creates a type with a single attribute. The code in the second file alterMyType adds a second attribute to that type.

```
// File createMyType
CREATE OR REPLACE TYPE my_type IS OBJECT (
    field1 int,
);
// File alterMyType
ALTER TYPE my_type ADD ATTRIBUTE field2 int;
```

Figure 4.17: Shared segment code example

In a new revision user changed the file alterMyType, we can see its new content in Figure 4.18. The user renamed the attribute field2 to field3.

```
// File alterMyType
ALTER TYPE my_type ADD ATTRIBUTE field3 int; // Changed
```

Figure 4.18: Shared segment code example - new revision

Figure 4.19 shows on the left the data flow graph in the old revision before changing the file. We can see that it correctly contains both attributes, field1, and field2. It also contains the my_type node, which was created by both analyzed files, and so it has two source segments. On the bottom is a merge input created based on an analysis of the changed file alterMyType. It correctly contains the renamed field field2.

Figure 4.19 shows on the right the data flow graph in the new revision after the merge. It correctly contains the attribute field3 from the merge input, but it incorrectly misses the field1 attribute. It is missing because the previous merge algorithm in step 2a deleted the whole subtree under the my_type node. However, if the server would not delete that subtree, the new data flow graph would incorrectly contain the field2 attribute.

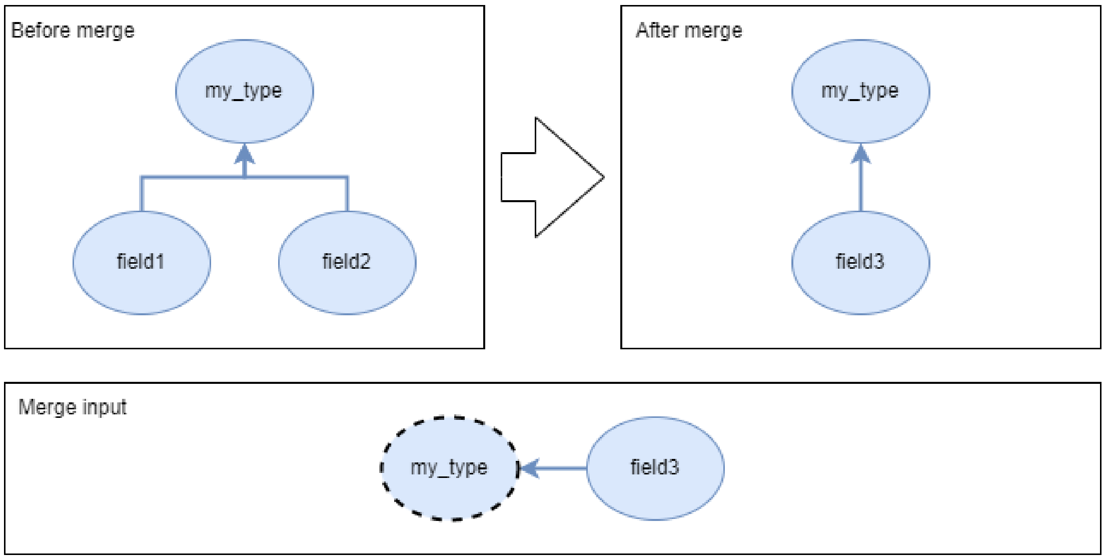

Figure 4.19: Shared segments example issue

Solution: If a node has multiple source segments, then also all its predecessors up to a segment root node have all these source segments. Therefore, if a segment root node has only a single source segment, all its successors also have a single source segment. And because the new merge algorithm always marks only segment root nodes (as described in Section 4.2.4), we can use this observation.

The new merge algorithm deletes its whole subtree when it merges a marked segment root node. Newly, if this segment root node has more than one source segment, the algorithm sends all these source segments (except the currently merged one) to the client for the reanalyzation. The client reanalyzes all these segments and sends their merge inputs back to the server. When merging these merge inputs, the server does not again delete the subtree of their marked nodes, and it does not again send other source segments back for the renalayzation.

In our example from Figure 4.19, everything happens in the same way. The only difference is that the server additionally sends back to the client request for reanalyzing the createMyType file. The client then reanalyzes this file and sends back to the server the merge input depicted on the bottom of Figure 4.20 which, of course, contains the field1 attribute. On the left of Figure 4.20 we can see that the data flow graph looks before this merge the same as how it looked after the first merge in Figure 4.19.

Finally, the server merges this new merge input to the data flow graph, but it does not delete anything this time. After that, as we can see on the right side of Figure 4.20, the data flow graph contains correctly both attributes field1 and field3.

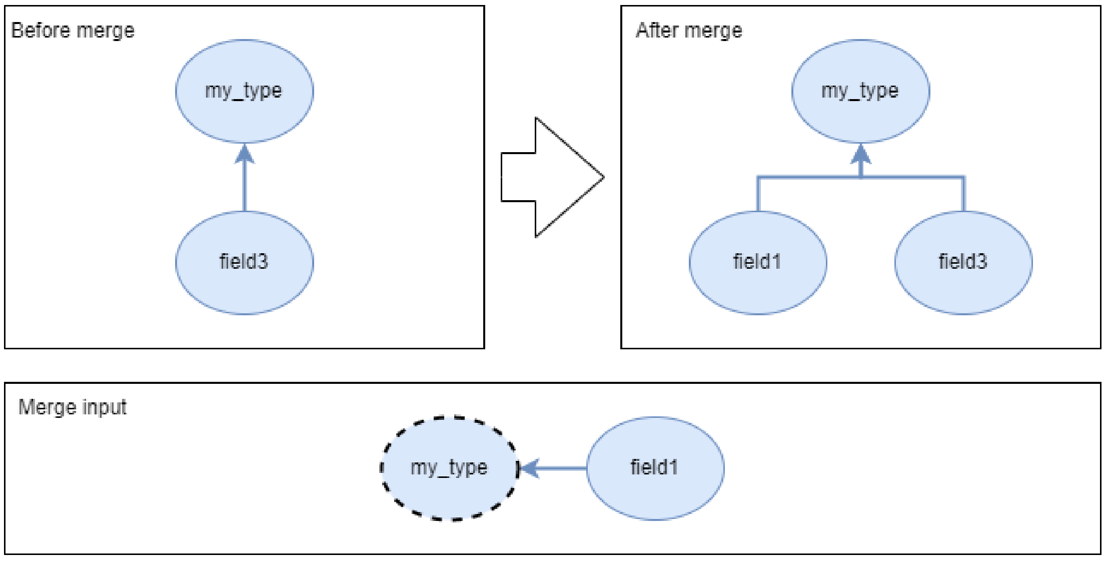

Figure 4.20: Shared segments example solution

To summarize, whenever multiple segments overlap in nodes they create, they are practically joined together. This means that if one of them changes, all nodes created by all these segments in previous revisions are deleted together at once. Later during the same revision all of these segments are reanalyzed (even if they didn't change) and all their nodes are merged back to the graph.

4.3 Data dictionary

A data dictionary is a read-only set of tables that provides metadata about a database (described in Section 2.11). It is analyzed in its own scenario, which means it can be analyzed before all other source files, after them, or not at all (depending on the user's settings). Analysis of the data dictionary creates a graph on the client, which is sent to the server to merge. The merge algorithm should merge it similarly to other merge inputs from the client. However, there are some significant differences in this case, which request changes in the merge algorithm.

4.3.1 Differences

Let us start with thinking about the differences between merge inputs created by analyzing a data dictionary and a merge input created by an analysis of a regular source code file.

The first difference is that any merge input created by analyzing a data dictionary is not sent to the server at once, unlike the regular source files that we have met so far. For performance reasons, graph G created by an analysis of a data dictionary is divided by the client into smaller subgraphs sent separately. Every node from G is sent at least once. However, because again with every sent node must be also sent all its predecessors, any node from G can also be sent to the client multiple times.

The second difference is that data dictionary nodes do not fulfill the condition from Section 4.2.3 that "If any node has a source segment, then all its children have this same source segment". We can see an example of that in Figure 4.21, which contains a whole data flow graph. In the example, only yellow nodes were created by analyzing the data dictionary, so they have their source segment data dictionary. We can see that some yellow nodes have non-yellow children.

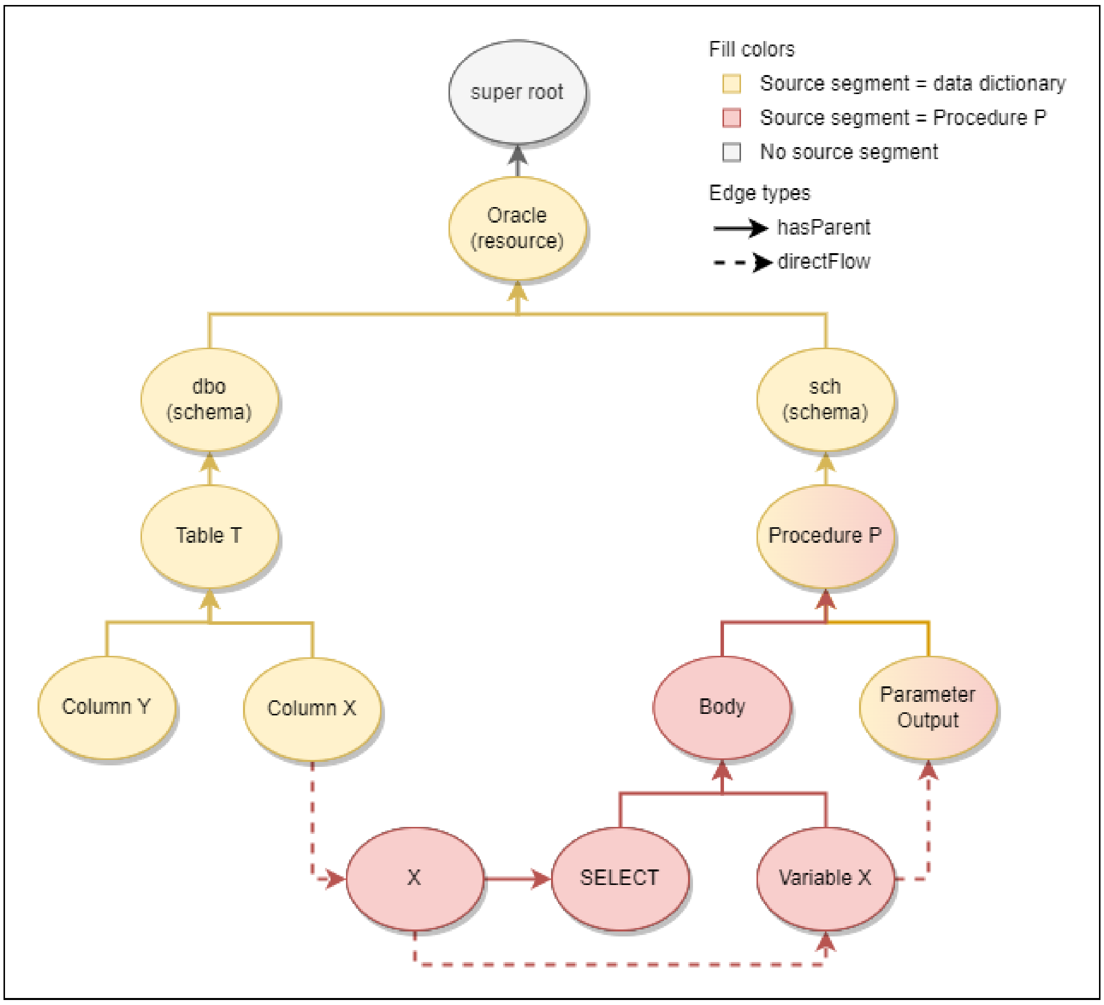

Figure 4.21: Data dictionary difference example

The third difference is that some nodes are created both by a data dictionary analysis and a source code file analysis. We can see examples in Figure 4.21, where nodes Procedure P and Parameter Output are both created by a data dictionary analysis and a procedure P source code file analysis.

The fourth and final difference is that the data dictionary merge input contains no actual edges. It can contain only hasParent and hasAttribute edges, but they are both represented as properties of nodes and not as edges. In some cases, this can help us because we do not need to worry about edges in the merge input. We can see an example of this also in Figure 4.21, where all data flow edges are in red because they were created by an analysis of the procedure P. An analysis of a data dictionary creates zero yellow data flow edges.

4.3.2 Possible solutions

Based on the previously mentioned differences, here is what needs to be done when the server merges a merge input from a data dictionary:

1. Server needs to delete all nodes created by data dictionary analyses in previous revisions that are not in the new revision.
2. Server should not delete nodes not created by a data dictionary analysis, even if that node is a child of a node created by a data dictionary analysis.
   (a) If the server deletes them, they would be reanalyzed and recreated. However, because almost every node in a graph is a descendant of a data dictionary node, the server would analyze almost every node in a graph, which is precisely what incremental updates try to avoid.

There are two different categories of possible solutions:

1. First solution is the full update of data dictionaries. The client will send all the extracted data to the server. We will change the merge algorithm only slightly so it can properly delete old nodes. The advantage of this solution is that it is easier to implement. The disadvantage is that incremental updates will always process all nodes in the data dictionary, slowing them down.
2. Second solution is a proper incremental update of data dictionaries. The client will analyze what changed in the input data dictionary and send only changed parts to the server. The advantage of this solution is its better time complexity. The disadvantage is that client needs to know what parts of the input data dictionary were changed.

The second solution would be much easier to implement using an incremental extraction. However, the incremental extraction is not a part of this thesis, and it might be analyzed and implemented later. Implementing the second solution without an incremental extraction would cost us a lot of implementation time. And later, after implementing the incremental extraction, all this work would be discarded. Therefore we have decided to implement the first solution, even though it's slower.

4.3.3 Rejected solutions

Because we have decided to use a full update for data dictionaries, the first obvious solution is to consider using the same data dictionary merge algorithm currently used by the full update. However, that solution is of no use for us because it expects the data flow graph to be deleted between every two revisions. Because of that, it does not support deleting nodes, which is what we need (as discussed in Section 4.3.2).

The second obvious solution would be to use the same data dictionary merge algorithm, which is used by incremental updates for a regular source code file and was created in Section 4.2.2. This also does not work because it marks a node and deletes all its descendants. If it would mark a data dictionary node and delete all its descendants the same way, it would delete many nodes created by many other segments. And as mentioned in Section 4.3.2, this would lead to unacceptable time complexity.

There is not a simple way of fixing any of these two algorithms. Therefore, we need to create a different algorithm for merging merge inputs created by a data dictionary analysis.

4.3.4 Accepted solution

Before discussing the data dictionary merge algorithm, let us mention two things. First, we will assume for now that a data dictionary is analyzed in every revision. What happens if that is not the case will be mentioned later in Section 4.3.6. Second, all nodes created by a data dictionary analysis have the data dictionary as their source segment.

The idea of the data dictionary merge algorithm is to perform deleting and merging together. Consequently, whenever the server merges a child of a node N, then the server needs to have complete information about all children of N at that time. Otherwise, it would not know which children of N created by the previous revision should be deleted. Therefore we need to change how the client divides the whole data dictionary graph into the separate merge inputs, independently sent to the server. We need every merge input sent from the client to meet the following condition:

- Whenever a merge input contains a (direct) child of a node N for the first time in a revision, the merge input also contains all other children of the node N.

The server then uses this information to delete all children of the node N that are no longer in the new revision and add all the new children that previously were not in the data flow graph. Without this condition, the server could not delete a child of N in the data flow graph and not in the merge input because it could still be a part of the following data dictionary merge input for the same revision. The client marks all such nodes because it is hard for the server to recognize which nodes satisfy the condition in a merge input.

We can see example of the new data dictionary merge input in Figure 4.22. The whole data dictionary graph was divided into two merge inputs. The first merge input is on the left, and the second one is on the right. All the red nodes were sent for the first time in the first merge input, and all the blue ones were sent for the first time in the second merge input. We can see that the Oracle node and the dbo node are marked in the first merge input because they are sent with all their children for the first time. Other nodes are marked in the second merge input. Note that every node is marked exactly once. Also, because this is a small example, the second merge input contains all nodes from the first one, which is not always true for bigger graphs.

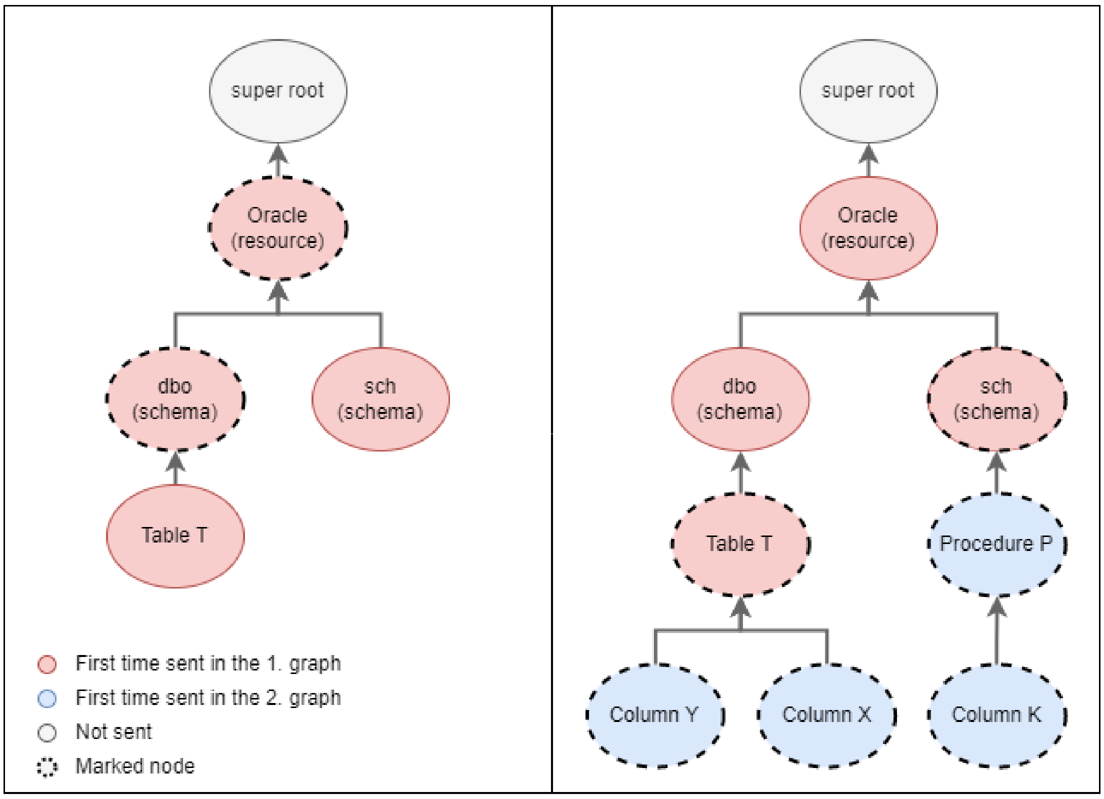

Figure 4.22: Data dictionary merge inputs example

Implementation

Now, let us discuss how the client divides the data dictionary graph into individual merge inputs. The client uses a DFS to traverse the data dictionary graph and adds all visited nodes into a merge input. Whenever there are precisely 4096 nodes in the merge input, the client sends the merge input to the server, deletes it, and continues with the DFS. We need to change this algorithm.

The first needed change is changing the algorithm from a DFS to BFS because DFS simply does not work with our condition. The second needed change is when the client sends the merge input. The new condition is that the merge input has at least 4096 nodes, and our condition is met. This slightly increases the volume of the graphs, but it should not increase it by much.

Shared nodes

Our data dictionary merge algorithm already covers the first two differences mentioned in Section 4.3.1. However, there is still one difference remaining. The third difference is that some nodes are created both by a data dictionary analysis and a source code file analysis. We want to prioritize information from the source code file when this happens. Therefore, the server does not merge this node when a data dictionary merge input contains a node that is already in the data flow graph with a different source segment. This means that the node will stay with a single source segment, which is not a data dictionary.

4.3.5 Algorithm

Let us summarize all this into an algorithm. This algorithm also uses some merge algorithm ideas that were previously discussed in Section 4.2. All these parts have a reference to the corresponding idea.

Input: Client analyzed a data dictionary and created a data dictionary graph G'', which was divided into multiple merge inputs. Now client sends one of the merge inputs G' to the server, which merges it with the flow graph G. The merge input G' meets the following condition “Whenever G' contains a direct child of a node N for the first time in a revision, the merge input also contains all the other children of the node N''. Every node satisfying this condition is marked.

Steps of the algorithm:

1. Go through the merge input G' and for each marked node N:
   (a) If N exists in the data flow graph G and has a source segment different from a data dictionary, do not do anything with N and skip all the following steps for it.
   (b) Remove from G all children of N that aren't in G', together with all their descendants and any adjacent edges or attributes.
      i. Perform node deletion from Section 4.2.7. This means that whenever this step deletes an edge with a source segment S that's not a data dictionary, add S into a list L of segments to reanalyzation.
   (c) Merge all the children of N from the merge input G' into the data flow graph G. Set their source segments to the data dictionary.
      i. Perform node creation from Section 4.2.8. Whenever the server adds a new child of N, check if N has any OnChildCreate edges. If yes, add source segments of all these edges into the list L of segments for reanalyzation.

Output: List L of segments which should be reanalyzed later in this revision, server sends this list back to the client.

4.3.6 Limitations

We have so far assumed that a data dictionary is analyzed in every revision. However, the user can decide for every revision if it should analyze a data dictionary or not. So, what should incremental updates do when a data dictionary is not analyzed?

An easy situation was running an incremental update without a data dictionary when the data dictionary was not analyzed in the previous revision. In such a case, whenever the server deletes a segment root node, it also deletes its parent if it has no other child, and the server does this recursively for a parent of this parent and so on. This way, the data flow graph contains the same nodes as it would after a full update.

The complicated situation was running an incremental update without a data dictionary when the previous revision analyzed the data dictionary. There are data dictionary nodes in the data flow graph, and the server does not know which should stay in the graph and which should be deleted because the data dictionary was not analyzed. Incremental updates will not support this situation.

It might look like the user needs to choose once and for all if a data dictionary analysis will be performed or not, which seems very limiting. However, there is an easy way to change this decision by running a single full update. And because a full update recreates the whole data flow graph, it also resets this decision. Therefore, the limitation is that after running a full update with a data dictionary analysis, all the following incremental updates must also analyze the data dictionary. And vice versa, after running a full update without a data dictionary analysis, all the following incremental updates should not analyze the data dictionary as well. This way, the limitations are minor, and it is not worth creating a complicated algorithm to solve them.

4.4 Segment mapping

We need to represent each segment in the flow graph somehow, and we also need an id for each segment. However, we cannot use segment root nodes for this purpose as there can be more that one for a single segment.

Because most segments correspond to a user source code file, we decided to represent each segment with a `source` node of the corresponding source code file (introduced in Section 2.9). Let's call this `source` node the **segment id node**. For every segment without a source code file, a "fake" empty file is created, so every segment has exactly one `source` node which represents that segment. Because every source node has a globally unique id, we can use this id to reference that segment. Let us call it the **segment id**.

We also need to slightly change the source code upload feature (mentioned in Section 2.9). For every source code file, we newly need to save its client file path (as an attribute of the segment id node). We need this in the step 5b of the simplified client algorithm in order to find the corresponding file on the client.

Implementation of getSourceSegment method

When we started improving the merge algorithm, we assumed in Section 4.2.2 that every node and edge has a `getSourceSegment` method. This method returns the source segment for that node or edge. Let us now discuss how this method functions for both edges and nodes.

Edge's source segment can differ from the source segments of both its source node and its target. So we need to save the edge's source segment inside every edge. It is newly stored as an edge attribute, the value of the attribute is the segment id of its source segment.

Nodes source segment could be saved for every node, but it is unnecessary. Instead, let us use the rule from Section 4.2.3 which says that "If a node has a source segment S, then all its children have the same source segment S." Using this rule, it is enough to save the source segment only for segment root nodes. Every other node has implicitly the same source segment as its parent node. A dependency edge newly connects every segment root node to a segment id node of its source segment.

We introduced the concept of data dictionaries in Section 4.3. Their analysis creates nodes that should have a data dictionary as their source segment. The easiest way to implement this is to say that every node that does not have any source segment using the previous methods has the data dictionary as its source segment. This approach might not be detailed enough in the future, but it is sufficient for now.

Example

Let us look at how source segments look in the flow graph with these changes in Figure 4.23. On the right, we can see the source file graph described in Section 2.5.2. Apart from the `Source root` node, it contains `File with procedure P code` node representing the source code file with the procedure P. This source node has a generated segment id 1. We can also see that all direct flow edges have this id as their attribute, which denotes that their source segment is the procedure P.

In the Figure 4.23, we can also see there that the analysis of the procedure P created a segment root node `Procedure P` that is connected to the segment id node `File with procedure P code` with a dependency edge. This means that the `Procedure P` and all its descendants have the same source segment P. All other nodes are not connected to any segment id node, so their source segment is implicitly the data dictionary. The only exception is the `source root` which is always present in the graph and has no sources segment.

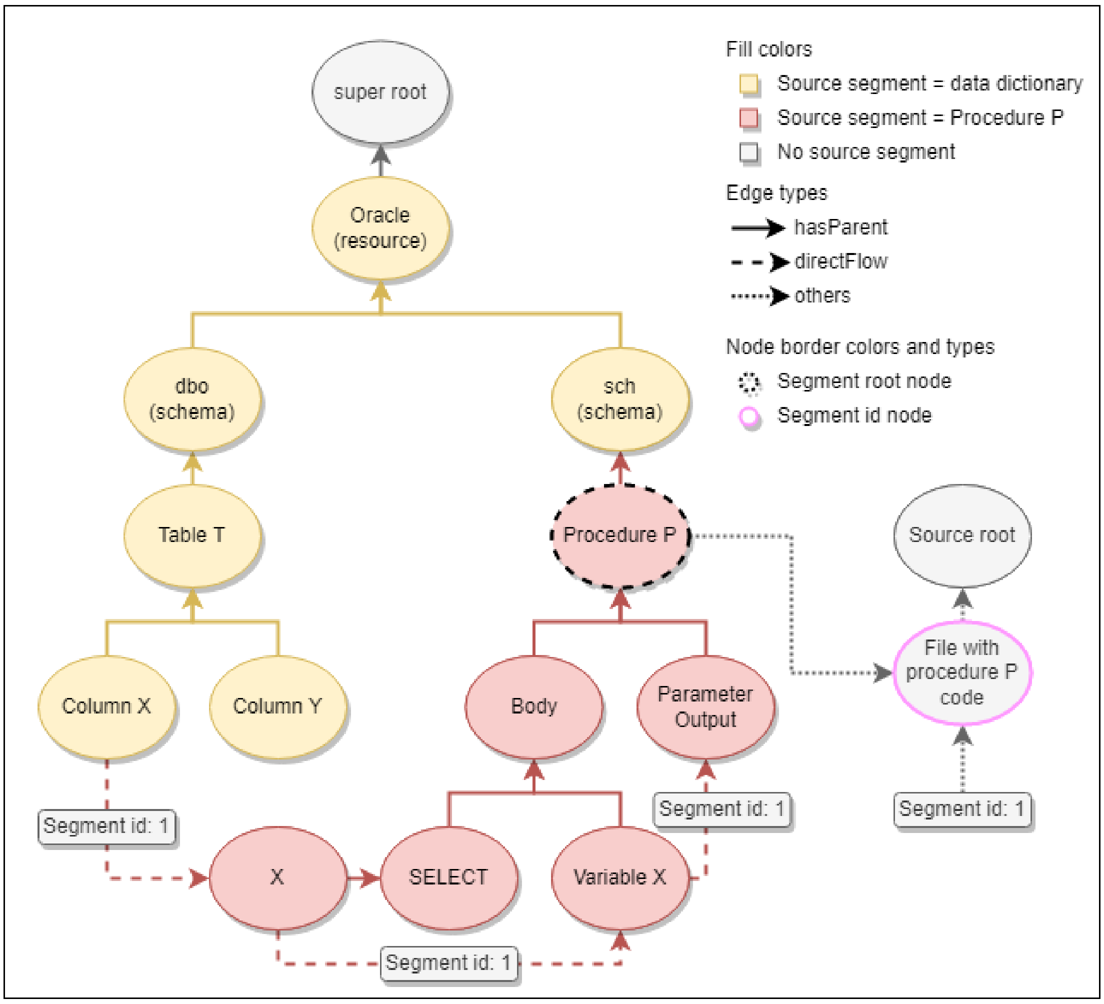

Figure 4.23: Segment mapping example

4.5 Post processing

We have discussed the purpose and previous implementation of post processing in Section 2.10. Post processing visited the whole graph, which is unacceptable for incremental updates because the whole point of incremental updates is only to visit parts of the graph affected by the current changes. Because of this, we need to change how post processing works.

In order to not visit the whole graph, we need to find all the segments where something either changed or could change. Then post processing can visit only parts of the flow graph that corresponds to these segments instead of going through the whole graph. This way post processing makes the same changes but does not visit the whole flow graph. However, how to find these segments?

Changing (inserting, updating, or deleting) a node with a source segment X can happen only during the analysis of the segment X. Thus whenever analyzing a segment X, it is enough to mark segment X as changed and later run post processing only for its subgraph (i.e., for the segment root node of the segment X and all its children).

However, it is different for edges. Analysis of a segment X can create an edge between nodes with different source segments Y and Z. Therefore marking only segment X as a changed segment and then later running post processing only on its subgraph is not enough. The reason is that the created edge is between nodes not in X's subgraph, so such a search would not find it. We need to mark either segment Y or Z (or both) as changed segments to find this new edge during post processing.

How to mark either of these segments? Let us realize that whenever we create or update (not delete) an edge adjacent to a vertex V with a source segment different from the one that's currently being merged, let us say segment Y, there will definitely be a segment root node of segment Y in merge graph sent to the server. The reason is that vertex V is either a segment root node or has a predecessor that's a segment root node. And merge graph contains with every node all its predecessors (as explained in Section 4.2.3), so it must include a segment root node of the segment Y as well. We can use this segment root node to mark segment Y as potentially changed, ensuring that post processing will find all the changed edges. Note that there can as well be no change in the source segment Y itself, so it might not be even reanalyzed in the current revision, which is why we need to do this in the first place.

And how to mark a segment as changed? It is best to use the segment id node (defined in Section 4.4) because there is exactly one for every segment. For every segment id node, there will be a new attribute containing the last updated revision of the corresponding segment.

Algorithm

The merge algorithm finds all the segment root nodes in the subgraph that's being merged. For every such segment root node, merge algorithm visits corresponding segment id node and sets its last updated revision attribute to the current revision.

Post processing then visits all the segments id nodes. For each one that was changed in the current revision, post processing visits its segment root node and all its descendants performing the same post processing algorithm it would for the full update.

Note that we cannot only mark the node as changed, but we need to save the current revision into its attribute containing the last updated revision. That is because post processing has multiple phases which can run in parallel. Therefore, there is no good place to set segments back into an unchanged state. This way (with the last updated revision attribute), there is no need to set anything back.

Source segment of created edges

Post processing creates new edges, so there is a question of what should be their source segment. Let us realize by looking at both supported post processing use cases in Section 2.10 that whenever post processing creates an edge, there is another edge that caused it. So the best solution is to copy the source segment X from the edge that caused the creation of the new edge.

This way, it's guaranteed that whenever the segment X is reanalyzed, all these edges created by the post processing because of some edge from the segment X will be deleted (because of edge deletion from Section 4.2.6). But also, because the segment X was changed, these edges will be recreated during the post processing if they're still needed.

This also means that whenever such edge is deleted by some other segment, segment X will be reanalyzed (because of node removal from Section 4.2.7). This will trigger post processing for the segment X, which will recreate the deleted edge if needed.

Issues

There is an issue with deleting edges not triggering post processing. Let's assume an analysis of a segment X, which contained an edge to another segment Y in the last revision. After deleting this edge in the new version, if there is no node from segment Y in the merged subgraph, the segment Y will not be marked as changed.

Consequently, the segment Y will not be visited by the post processing, although it should be as it has changed. However, this behavior does not break anything because post processing currently does something only when new edges or nodes are created, and it is not affected by the deleting.

Example

Let us look at an example of post processing. In Figure 4.24 below, we can see an unchanged procedure P with a single parameter and a second procedure S that calls the procedure P. The procedure S contains some insignificant change. In our example, the variable J was renamed to K.

```
Procedure P(IN I) { ... }  // Unchanged in the new version
Procedure S() { ... P(J) } // Old version
Procedure S() { ... P(K) } // New version
```

Figure 4.24: Post processing code example

We can see the corresponding data flow graph in Figure 4.25. On the left is the flow graph before the merge. It contains two segment id nodes (denoted by violet circles), both were changed in the last revision, and so they have the `Last updated revision` attribute set to 1.0.

On the bottom of Figure 4.25 is the merge input for the revision 1.1 created based on the previously described changes of the procedure S. The merge input contains marked node `Procedure S`.

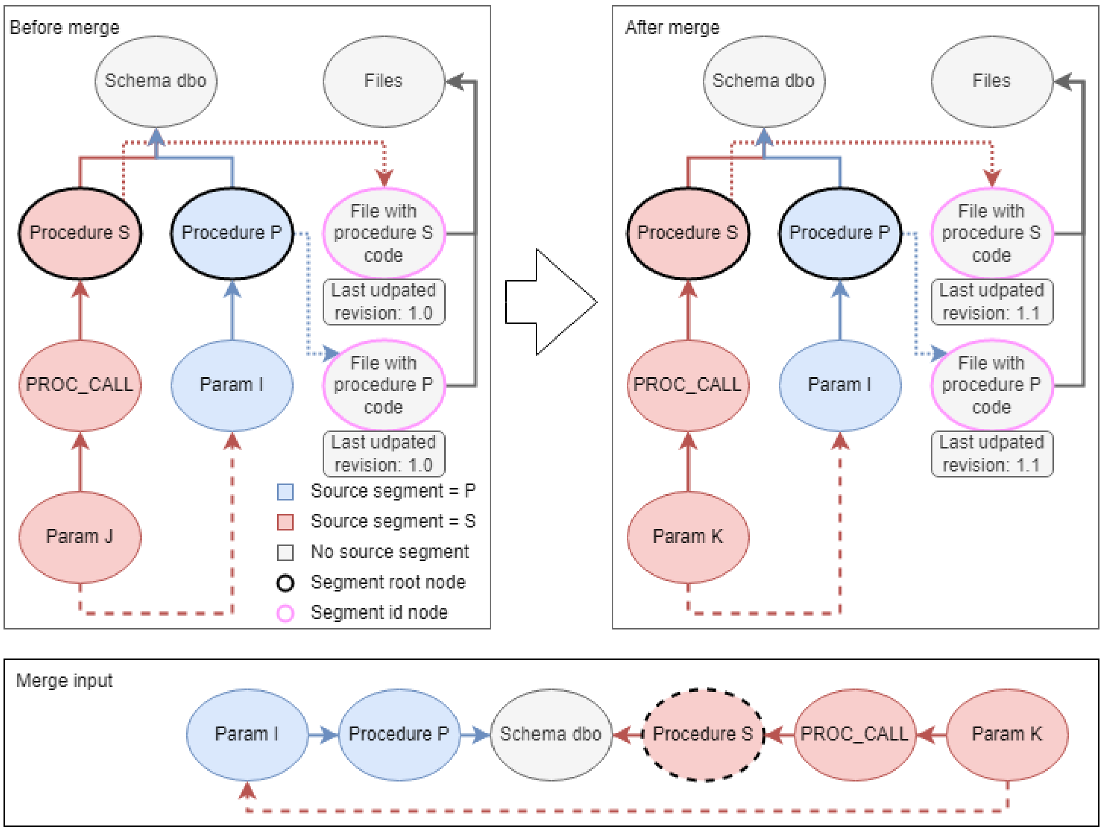

Figure 4.25: Post processing example solution

We can see the flow graph after the merge on the right of Figure 4.25. Segment id nodes for both procedure S and P have the `Last updated revision` set to 1.1. The reason is that the merge input contained segment root nodes of both these segments. Consequently, post processing will in the revision 1.1 visit subtress of both segments S and P.

4.6 New merge algorithm description

After analyzing issues of the previous algorithm and its new improvements one by one, let us put them together and see what the resulting algorithm looks like. Before we look at the algorithm itself, let us examine its data structures and their purpose.

The first data structure is the list L. This list starts empty, and whenever the merge algorithm founds a segment that could generate a different graph compared to the last revision, it adds it to the L. This list is sent back to the client at the end of the merge algorithm. The client reanalyzes every segment from this list later during the same revision if this segment was not already analyzed in this revision.

The second data structure is the hash map H. During the merge of a segment S, this map contains all edges removed by the merge algorithm with the source segment different from S. Edges from this map are first used for edge restoration (see Section 4.2.5), and they are deleted after restoring. At the end of the merge algorithm, all remaining edges are used for node removal (see Section 4.2.7).

Each edge e between some nodes u and v is in the hash map H twice, once with the node u as its hash key and once with the node v as its hash key. The hashed value in both cases is the edge e. This map can have multiple values with the same key. $H[N]$ represents (contains) all edges removed by the merge algorithm adjacent to the node N, which is exactly what the merge algorithm needs to know.

Algorithm

Now that we know the main data structures let us look at the new merge algorithm. Steps of the algorithm are written in black color, comments in grey.

Input: The whole data flow graph G stored on the server and the graph G' which was created by an analysis of a segment S and should be merged with G. Graph G' has marked by an attribute all the segment root nodes in the graph which have source segment equal to S. This indicates that a change has been performed somewhere in the subgraph, starting with this marked node.

Steps of the algorithm:

1. Create the empty list L and the empty hash map H.
2. Delete all edges created by the merged segment S in previous revisions.
   (a) Skip this step if S is a shared segment and its segment root node was already changed in the current revision. This step is explained in the Shared segments analysis in Section 4.2.9.
   (b) Technical details of this steps are described later in Section 5.4.
3. Merge graphs G and G' node by node until a specially marked segment root node in G' is reached.
   (a) Whenever the segment root node is reached, find the corresponding segment id node and mark it as changed by setting its last updated revision attribute to the current revision. This step is explained in the post processing analysis in Section 4.5.
4. For every marked node in G':
   (a) Remove its subtree from G by setting the minor end revision property to the latest revision. This means removing the marked node, all its descendants, all adjacent edges, and all adjacent attributes.
      i. Skip this step if S is a shared segment and its segment root node was already changed in the current revision. This step is explained in the Shared segments analysis in Section 4.2.9.

   ii. If S is a shared segment, find all other shared segments connected to the same segment root node and put them into the list L. This step is explained in the Shared segments analysis in Section 4.2.9.
   iii. Save all edges with source segment different from S removed this way into the hash map H.

(b) Continue merging descendants of the marked node. For every node N from G' merged this way:
   i. If the same node N was found in G in the latest revision:
      A. Add the node N back to graph G by setting its minor end revision back to maximum.
      B. Add every edge from $H[N]$ back to G (in the same way as the node N). Remove all these edges from H. This step is explained in the Edge restoration analysis in Section 4.2.5.
   ii. If the same node N was not found in G in the latest revision:
      A. Add the node N to G as a new node.
      B. If there is a dependent edge of type `OnChildCreate` from any node X to the parent of the node N, then add the source segment of the node X to the list L of segments to reanalyzation. This step is explained in the Node creation analysis in Section 4.2.8.

(c) Merge all edges from the subtree of the marked node. Note that all these edges have S as their source segment. For every edge E from G' merged this way:
   i. If the same edge E was found in G in the latest revision, then add the edge E back to G by setting its minor version back to the maximum.
   ii. If the same edge E was not found in G in the latest revision, then add the edge E to G as a new edge.

(d) For every edge that remained in the hash map H add its source segment to the list L of segments for reanalyzation. This step is explained in the Node removal analysis in Section 4.2.7.

Output (sent to the client): List L containing segments that need to be reanalyzed as well.

This algorithm works not only for data flow edges, but also for `mapsTo` and `perspective` edges. We need to set source segments correctly for these edges to reanalyzation work properly.

4.6.1 Algorithm for segment deletion

We have so far assumed that segments are only updated. The previously mentioned algorithm works correctly when updating existing segments and when creating new segments. However, the algorithm does not work when a segment is deleted. In such a case, there is no merge input for the deleted segment, so the deleted segment stays unchanged in the flow graph and is not deleted. Consequently, whenever deleting a segment, the client needs to tell the server that the segment was deleted.

Let us say that a segment is deleted in the new revision. The client first finds which segments were deleted, then finds all their segment root nodes and sends them to the server. After that, the server finds all these nodes in the flow graph and deletes them and all their successors. The server also deletes all the edges created by these deleted segments in previous revisions.

Algorithm

Inputs: List of segment root nodes sent by the client. The data flow graph G stored on the server.

Steps of the algorithm:

1. Create the empty list L. Same as the list L in the main merge algorithm, contains segments that are sent to the client and will be reanalyzed in the same revision.
2. For every node N in the list of deleted segment root nodes:
   (a) Find the node N in the graph G and delete it and all its successors.
      i. Also, delete all adjacent edges and attributes.
      ii. Whenever an edge is deleted this way, add its source segment to the list L if the source segment is different from S. This step is explained in the Node removal analysis in Section 4.2.7.
      iii. Find and delete all edges created by the segment S. This step is same as the step 2 in the main merge algorithm.

Ouput (sent to the client): List L containing segments that need to be reanalyzed in the same revision.

4.7 New merge algorithm example

Let us examine the new merge algorithm step by step in an example with four procedures that we can see in Figure 4.26. Most important for us is the procedure P with a single argument. In the old version, this procedure stored its input value into the variable X and called another procedure Q. However, in the new revision, someone found out that the variable was useless, removed it, and changed the called procedure to another procedure R. Then there is also the unchanged procedure A, which calls the procedure P.

First, let us look at the flow graph G on the server before the incremental update started, which is in the first revision 1.0 in Figure 4.27. We can see that

```
Procedure dbo.Q(IN I) { ... }  // Unchanged in the new version
Procedure dbo.R(IN I) { ... }  // Unchanged in the new version
Procedure dbo.P(IN J) { VAR X = J; Q(X) } // Old version
Procedure dbo.P(IN J) { R(J) } // New version
Procedure sch.A() { ... P(K) } // Unchanged in the new version
```

Figure 4.26: Algorithm example code

each of the four procedures creates its segment, and each is shown in a different color. Each segment also has its segment root node with a black border color connected to a segment id node with a violet border color.

The graph also contains data flow edges. We can see data lineage that starts in the `Param K` node of the procedure A, goes through nodes of the procedure P, and ends in the `Param I` node of the procedure Q.

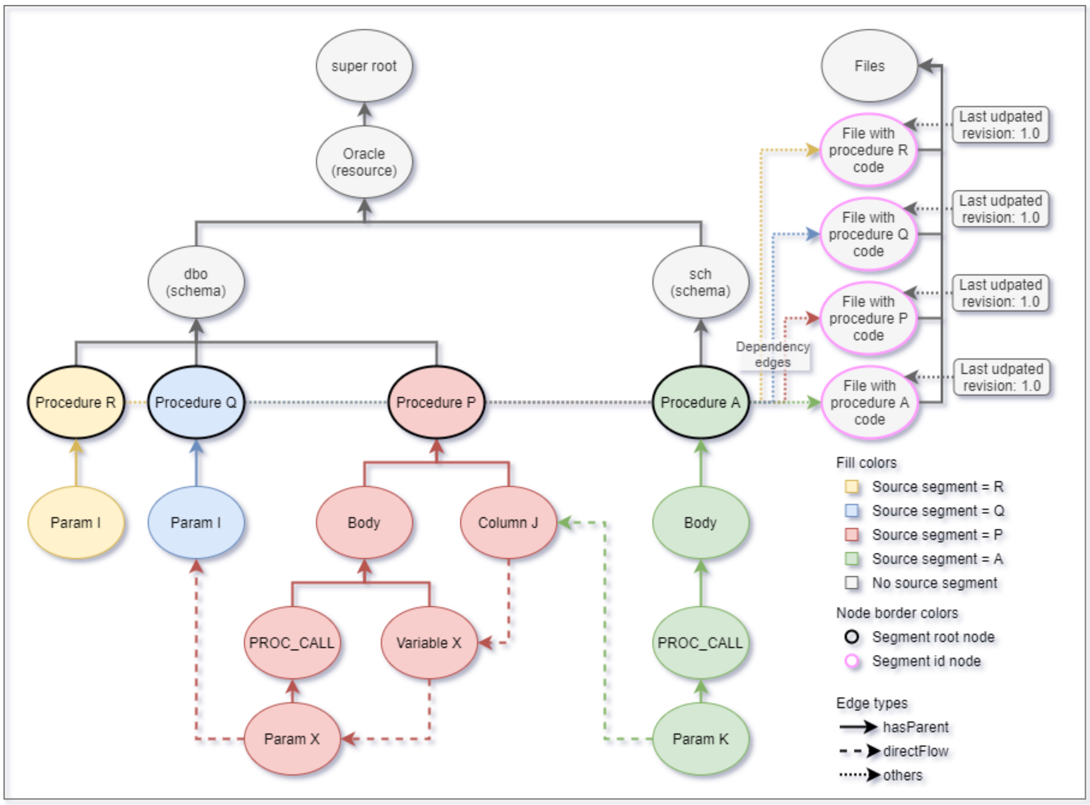

Figure 4.27: Algorithm example - graph before update

Input

The incremental analysis starts on the client and follows the client algorithm described in Section 4.1. In the first step, the client extracts source code files of all four SQL procedures introduced earlier in Figure 4.26. The client starts the incremental update in the second step by creating a new minor revision 1.1. In the third step, the server adds the segment P into the queue Q as it is the only one with the changed input (as the procedure P is the only input that changed).

The fourth step is skipped because $L1$ dependencies are not used for database technologies.

The fifth step analyzes segments from the queue Q while it is not empty. It currently contains only the segment P, which the client reanalyzes, creates a merge input graph G', and sends this graph to the server. We can see this merge input in Figure 4.28. It reflects the changes made in the new version of the procedure P as it no longer contains the `Variable X` node. We can also see two segment root nodes in the merge input. One of them is marked because its source segment matches the currently merged segment.

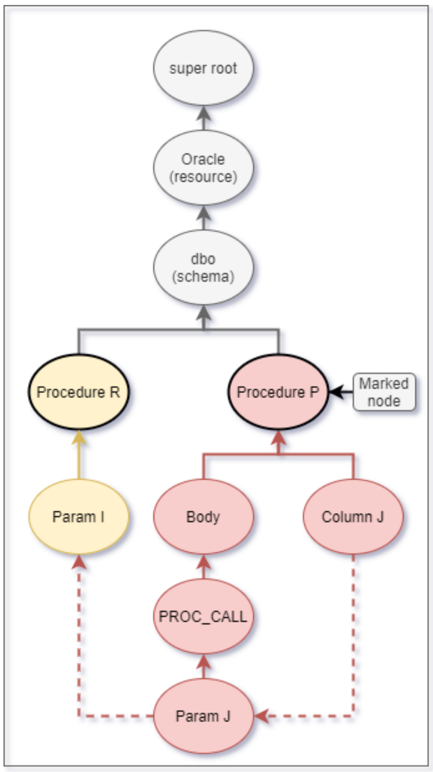

Figure 4.28: Algorithm example - graph to merge

Let us now describe how the server merges this input into the whole data flow graph described earlier. The server uses the new merge algorithm described in Section 4.6. We will examine it step by step.

Steps 1 and 2

Step 1 only creates empty data structures (the list L and the hash map H).

Step 2a deletes all edges created by the currently merged segment P from the flow graph G. The updated graph G is in Figure 4.29. We can see that all red data flow edges are half-transparent, representing that they are not valid in the current revision. They are still part of the data flow graph, but only as parts of the previous revisions.

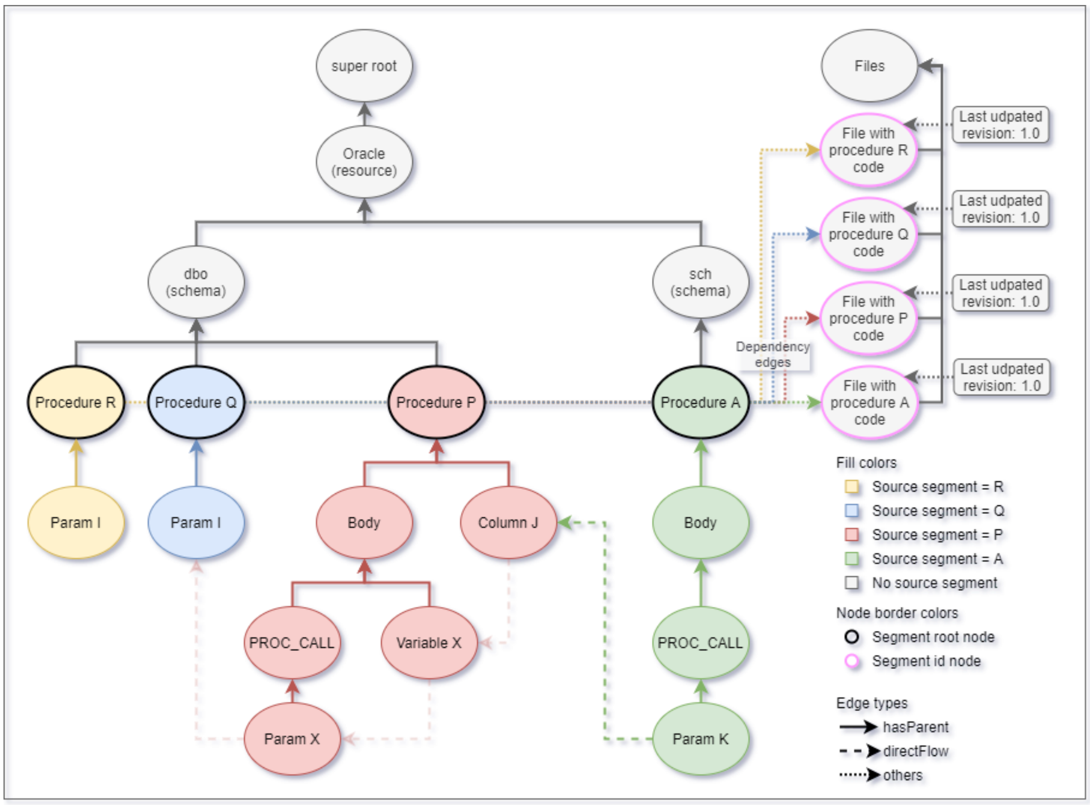

Figure 4.29: Algorithm example - graph after steps 1 and 2

Step 3

Step 3 starts in the super root and merges the merge input graph G' node by node into the flow graph G. It merges `super root`, `Oracle` and `dbo` nodes first, not creating any changes, because these nodes were already present in the graph G. Then the merge algorithm can arbitrarily decide between nodes `Procedure R` and `Procedure P`. The order does not matter, for our example, let us say that the algorithm chose the `Procedure R` as the first one. This means that nodes `Procedure R` and `Param I` are merged into G as well, not creating any changes. Then algorithm gets to the node `Procedure P`, which is the marked node. This means that the step 3 ends here.

Step 3*a* during this merging also checks if every merged node was a segment root node. This was true for the nodes `Procedure R` and `Procedure P`. Therefore, their segment id node was found using dependency edges, and its last updated revision was set to 1.1, which is the number of the current revision.

We can see these changes in Figure 4.30. Nodes from the merge input G' that has been already merged are marked with a dashed border. We can also see that the last updated revisions for the segment id nodes for segments R and P have changed to 1.1.

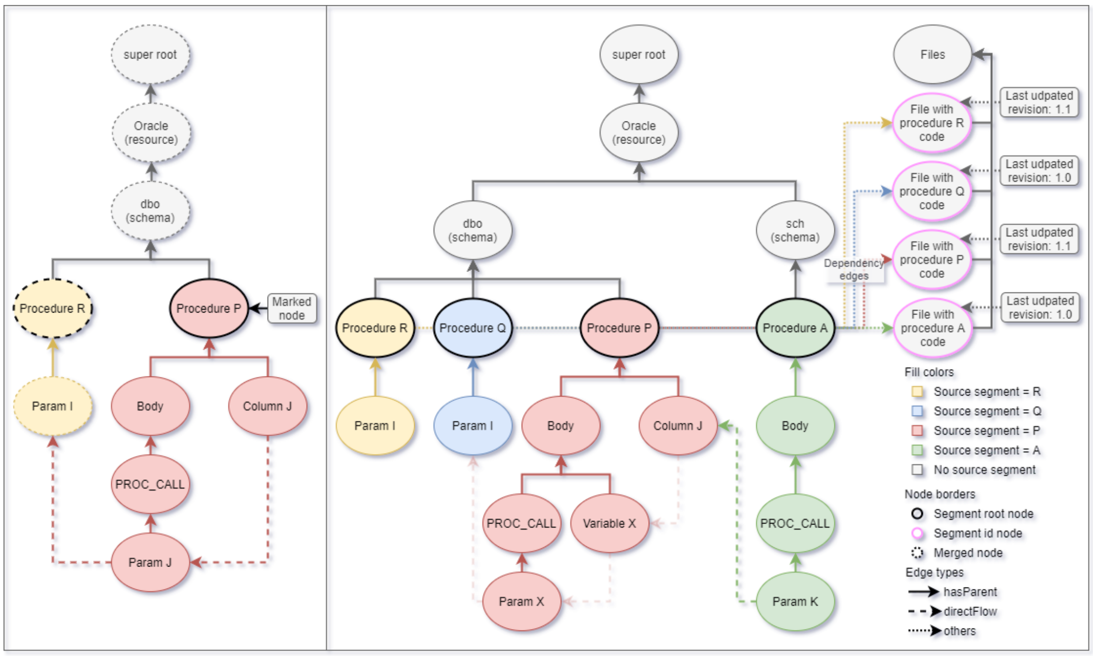

Figure 4.30: Algorithm example - graphs after step 3

Step 4a

Step 4*a* finds and removes from the current revision the whole subtree of the marked node `Procedure P`, we can see the results in the Figure 4.31. The edge from the `Param K` node to the `Column K` node is also removed because this step also removes all the adjacent edges and attributes. This removed edge is in step 4*a*-*iii* saved into the hash map H.

Step 4b

Step 4*b* continues merging the merge input G' starting from the `Procedure P` node and continuing with all its descendants. All nodes that are merged this way and were previously removed in the step 4*a* are added back to the current revision in the step 4*b*-*i*-*A*. This includes `Procedure P`, `Column J`, `Body` and `PROC_CALL` nodes. After merging each of these nodes, in the step 4*b*-*i*-*B*, the hash map H is checked for any adjacent edges to this node. One is found when merging the `Column J` node, and that edge from the `Param K` node to the `Column J` is added back to the graph G as well and deleted from H.

The only remaining node `Param J` is merged differently because it did not exist in the last revision. It is created as a new node in the step 4*b*-*ii*-*B*. We can see the resulting flow graph in Figure 4.32. Note that nodes `Variable X` and `Param X` were not added back to the current revision in G because they are not part of the merge input G'. So these nodes are deleted in the new revision.

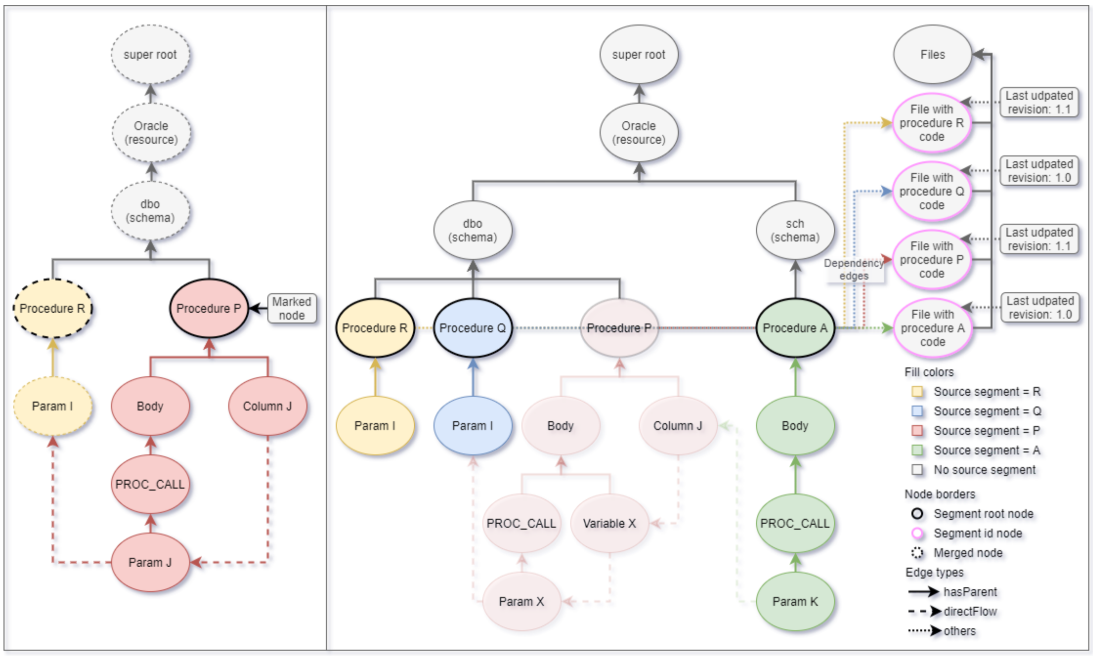

Figure 4.31: Algorithm example - graphs after step 4a

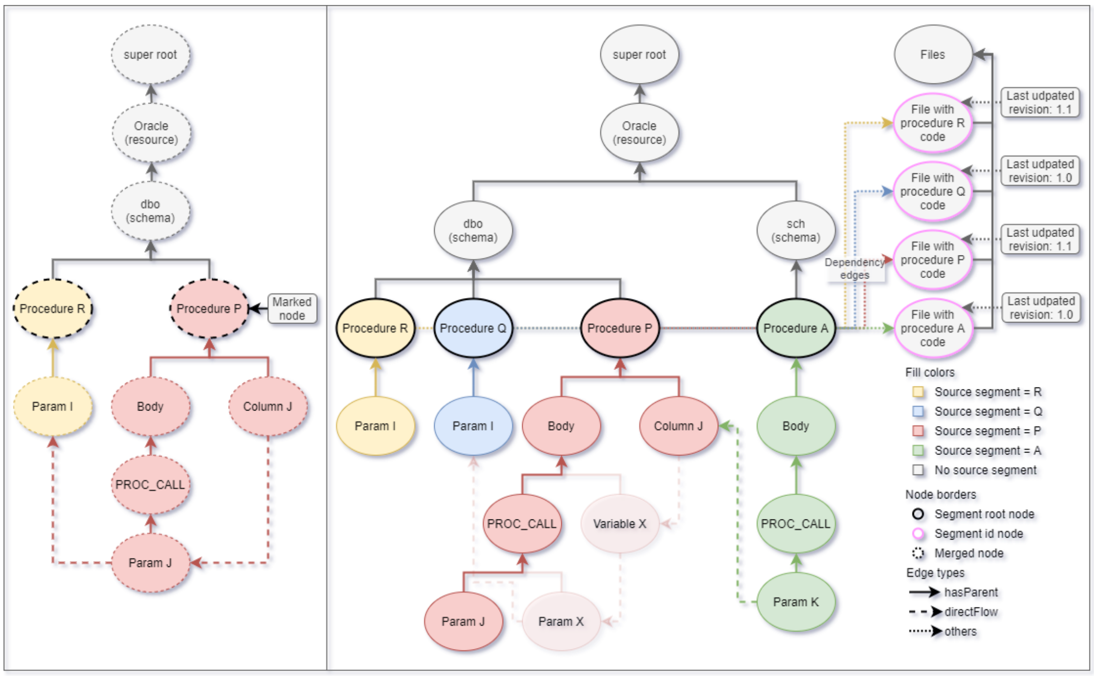

Figure 4.32: Algorithm example - graphs after step 4b

Steps 4c and 4d

Step 4*c* merges all the edges from the graph G' into the graph G. In the last revision, these merged edges did not exist, meaning they are newly created in the flow graph G in the step 4*c*-*ii*. We can see the new flow graph in Figure 4.33. Note that all other red edges created by the segment P are removed in the revision.

Step 4*d* should add all the segments from the map H into the list L for later reanalyzation by the client. But in our example, the hash map H is empty. An example of this situation was described earlier in Section 4.2.7.

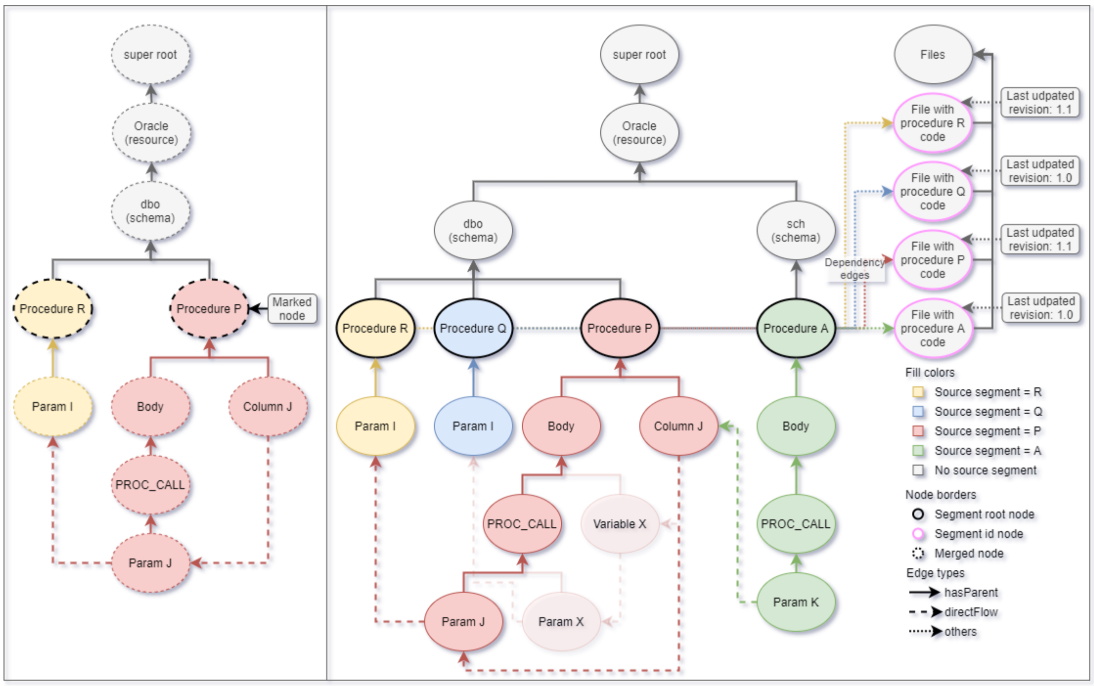

Figure 4.33: Algorithm example - graphs after step 4d

Output

In this example, the list L containing segments that need to be reanalyzed is empty. This means that this empty list is returned to the client. And because the queue Q of segments that need to be analyzed is empty, the client algorithm ends.

Let us look one last time at the graph from this example in Figure 4.34. We can see the new flow graph without the deleted objects from the last revision.

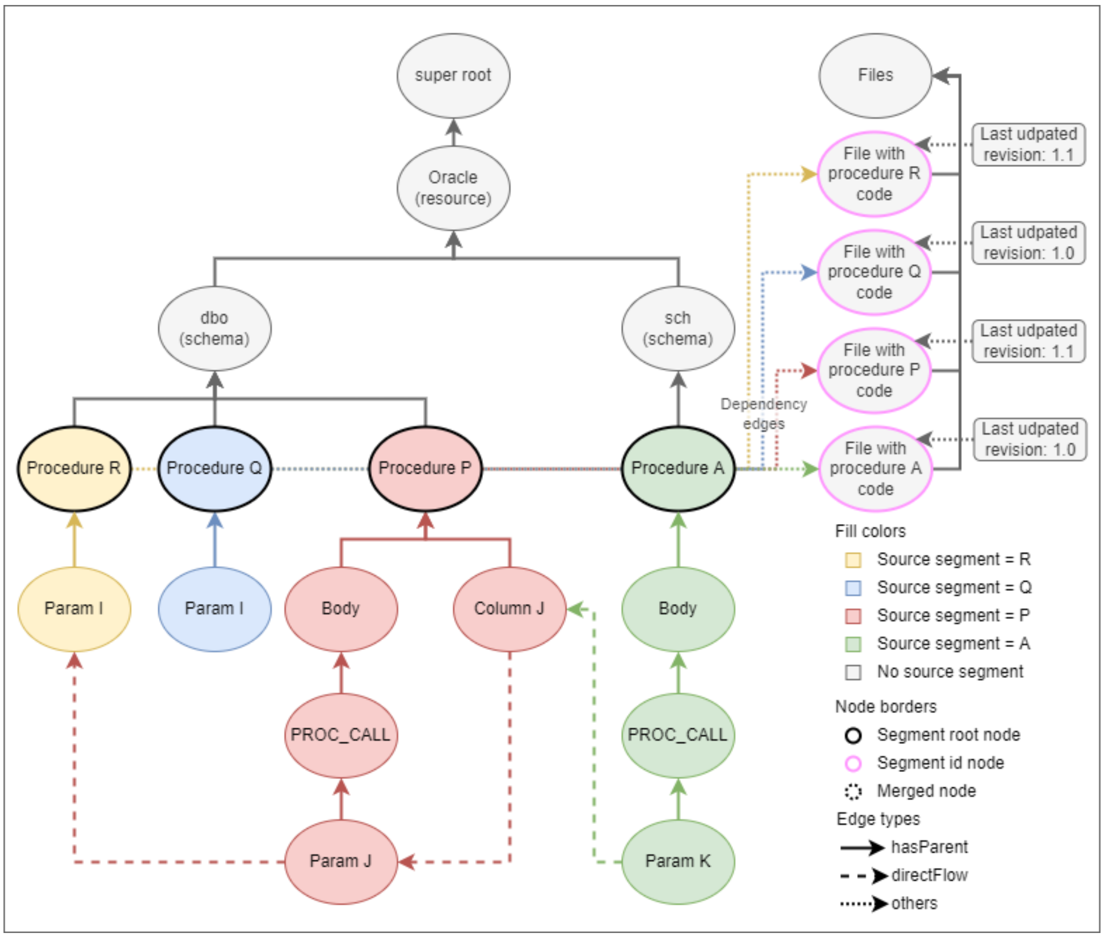

Figure 4.34: Algorithm example - resulting graph

4.8 Full client algorithm

The client algorithm calls the merge algorithm and orchestrates the whole process. We have already seen its simplified version in Section 4.1. Now that we are familiar with all the details of the new merge algorithm, we can finally introduce the full version of the client algorithm. It contains a few new steps compared to the previous simplified version. Each of these steps contains a comment stating this fact.

The situation is that a user made some changes in their files and wants to update their data flow graph. At least one full update has already been performed (note that there is no reason for the first update not to be a full update), which created the data flow graph G on the server. User chooses which technology (e.g., Oracle) should be updated. The client algorithm then does these steps to update the chosen technology incrementally:

1. Run extraction phase scenarios for the changed technology. This step is the same in full update, as incremental updates should use the same extraction.
2. Start an incremental update by calling newMinorRevisionScenario. This is an already existing scenario that correctly initializes graph database structures for an incremental update.

3. Find all changed segments and add them to a queue Q. Changed segments are segments with a changed input (described in Section 3.2).
4. Use L1 dependencies to find unchanged segments that generate different output because of changed segments and add them to the queue Q.
5. Find all deleted segments. Analyze them, find all their segment root nodes, and send these segment root nodes to the server. This is a new step needed for the segment deletion algorithm introduced in Section 4.6.1.
   (a) Server returns the list of segments L that need to be reanalyzed. Add these segments into the queue Q.
6. While the queue Q is not empty:
   (a) Pop segment S from the queue Q. If S has already been analyzed in this revision, discard S and go to the beginning of step 6.
   (b) Analyze the segment S and send created data flow graph to the server to merge it.
      i. In the created data flow graph, mark all segment root nodes and set correct source segments for all edges. This new step contains requirements discovered during our merge algorithm analysis.
   (c) Receive from server a list L of segments that need to be reanalyzed because of changes in segment S and push them to the queue Q. Server uses L2 dependencies to find segments in the list L.
   (d) Use L1 dependencies to find all unchanged segments that generate different output because of changed segments in the list L and add them to the queue Q.
7. Perform post processing calling updated repositoryPostprocessingScenario. This is a new step explained in Section 4.5.
8. Commit the incremental update by calling commitRevisionScenario. This is an already existing scenario that commits all the changes.

Use cases

Requirements contain two different use cases in Section 3.3. The client algorithm we have just described focuses only on the first one-technology use case because that is the one that is supported in the implemented prototype (see Section 3.6). The second all-technology use case was also analyzed, and it is possible to implement it. The merge algorithm remains the same, but the client algorithm needs some changes. Also, a few underlying MANTA concepts need major changes (e.g., how scenarios interact with each other). The analysis of the second use case is not included because it would need a detailed explanation of some other MANTA concepts and because it is not essential for the implementation part of this thesis.

Parallelization

Changes in this thesis maintain both interscenario and intrascenario parallelization. The parallelizaton of the merge algorithm is being solved by another thesis. We have considered concepts from that thesis, but the analysis itself belongs to the other thesis.

5. Implementation

Based on the requirements, analysis, and design from previous chapters, a prototype was implemented that supports incremental updates for Oracle database technology. This chapter describes both the new and the changed code that was implemented. All source files created or updated during the implementation of this thesis are included in the online attachment of this thesis and the structure of the attachment is described in Appendix A.

Because the code was developed using almost exclusively test-driven development, there are tests for every change made. However, as they are not crucial for this chapter, we will not mention them when describing individual classes and changes. They are part of an attachment of this thesis and described later in Section 6.1.

5.1 Overview

Before describing the new implementation changes, let us look at a few already existing key implementation concepts and classes. We will need them to understand better the new code presented in this chapter.

Let us first look at the communication between the client and the server. Whenever the client sends a flow graph to the server to merge (as described in Section 2.1), the client uses the `MergerWriter` class to serialize the graph. We can see an example of a serialized graph in Figure 5.1, each nonempty line represents either a node or an edge in the flow graph. There are two important things to notice. First, all serialized objects are ordered by their type, and the order is always the same. In the example, all layers are sent together, then all resources, then all nodes, and so on. The second important thing is that if a row in the file references another row, then the referenced one is always in the file earlier. In the example, the node `Schema` references the node `Database`, and so the `Databse` node is in the file before the `Schema` node.

```
"layer","0","Physical","Physical"
"resource","1","Oracle","Oracle","Oracle Database","0"

"node","1","","ORCL","Database","1"
"node","2","1","BSL","Schema","1"
"node","3","2","IS_TABLE_EXIST","Function","1"
"node_attribute","3","CONNECTION","manta"

"edge","1","7","8","DIRECT","2","1"
```

Figure 5.1: Csv input example

Server merges this csv file using the `StandardMergerProcessor` class, which merges the graph line by line. Each object is merged independently, which is possible thanks to their ordering in the file. Whenever merging an input csv file, a new instance of the `AbstractProcessorContext` class is created, and it is accessible when merging all objects from that file. It is used whenever knowledge about previously merged objects from the same csv file is needed.

5.2 Model

The first significant model change was to modify edges so each edge stores a reference to its source segment. This was implemented on the client by adding `sourceSegmentId` property to every edge in the `Edge` class. Together with this change, we have also needed to update the merging and serialization of edges in `GraphCsvSerializationHelper` class to work with this new property correctly. Another related change was in `AbstractSourceFileTask` so it correctly sets the new `sourceSegmentId` to all edges before sending them to the server.

We also needed to cover these changes on the server, so it correctly handles edge's new `sourceSegmentId` property sent from the client. On the server, the source segment id is loaded from the input csv file sent by the client and stored as an edge attribute in the data flow graph. We have already discussed the reasons for this approach in Section 4.4.

We need to store source segment references not only for edges but also for nodes. To do that, we created a new edge type `HAS_DEPENDENCY` in class `DatabaseStructure` that connects segment root nodes with their segment id nodes as described in Section 4.4. These edges are created by the merge algorithm in `StandardMergerProcessor` class whenever it merges a segment root node.

The last thing to mention is changes in `RevisionRootHandler` class, which can now create not only revisions for major full updates but also revisions for minor incremental updates.

5.3 Merge algorithm improvements

This section describes the implementation of the new merge algorithm that was designed in Section 4.2. It goes through all its improvements and describes where and how they were implemented.

5.3.1 Changing the marked node

The first change was to mark procedure nodes instead of body nodes whenever performing an incremental update. We've discussed this topic in Section 4.2.4 and it's implemented in the `NodeFlagTask` class. In addition to marking the procedure node, this class can now mark other node types, which will be later used for many other node types, including script nodes.

5.3.2 Edge restoration

We've described edge restorations in Section 4.2.5 and Section 4.2.6. To summarize, whenever merging a segment, all deleted edges created by other segments are newly restored when either of their adjacent nodes is restored.

We first need to find and save all deleted edges created by other segments to implement this. The `GraphOperation` class is responsible for deleting edges. Whenever it deletes an edge created by another segment, it adds it into `AbstractProcessorContext` class. This class stores edges in a map grouped by the ids of their adjacent nodes. Each edge is in the map twice, each time with a different key (once with a source node, once with a target node).

Whenever a node is merged, all adjacent edges are picked up from this map and sent to `GraphOperation`'s function `restoreEdges`. This function restores these edges if both their adjacent nodes are still valid in the new revision.

5.3.3 Node removal

We've discussed node removal in Section 4.2.7. To summarize, whenever removing a node causes the removal of an edge and this edge is not restored in the same revision, then the source segment of this edge needs to be reanalyzed in the same revision.

Class `AbstractProcessorContext` already stores all deleted edges because of the edge restoration. It needed only one update to work correctly for node removal as well. This update was that whenever an edge is restored, class `StandardMergerProcessor` removes this edge from the list of the deleted edges. This means that after the whole segment is merged, this list contains exactly all the deleted edges.

After merging all nodes and edges from the client input, `ClientMerger` class uses this prepared list of all the deleted and not restored edges to find all their source segments. These source segments are added into `ProcessingResult` and `MergerOutput` classes. These segments are also sent to the client, and they are analyzed in the same revision.

5.4 Edge deletion

Whenever merging a segment, all edges created by this segment in previous revisions must be deleted. However, there is an issue with this approach and the Neo4j database. Neo4j cannot index attributes on the edges, so we cannot efficiently scan the flow graph for edges created by one specific segment. This feature should be implemented in the future, but we need to create a workaround for now. It is enough to create this workaround for edges that have their source segment different from both its adjacent nodes. All other edges are deleted when deleting their adjacent node with the same source segment.

Because it is possible to index nodes, we can mark one of its adjacent nodes whenever creating an edge. Later, when we need to find the edges to delete, we can instead find all marked nodes and then find the correct edges to delete by going through all of their adjacent edges and choosing those with the correct source segment. Let us now describe this workaround in more detail.

We first need to have a list of all nodes created by the currently merged segment. Whenever `StandardMergerProcessor` class merges a marked node, it adds this node into the list of nodes created by the current segment. This list can be found in `AbstractProcessorContext` class. Also, whenever merging a node, if its parent is in the list, the merged node is added to this list. This way, the list contains all nodes created by the merged segment.

Whenever `StandardMergerProcessor` class creates a new edge, using the previously created list, the merge algorithm can decide if the edge is adjacent to a node from its source segment. If not, when creating this edge, the merge algorithm adds an `EdgeDeleteFlag` attribute to the edge's source node. The value of this attribute is the id of the currently merged segment.

Whenever merging a segment, `GraphOperation` class should delete all edges created by the merged segment. Using the new attribute, it just searches all nodes with this attribute that have the attribute value equal to the id of the merged segment. Then, it deletes all their adjacent edges with the same source segment as the merged segment.

5.5 Data dictionary generator

The data dictionary algorithm was described earlier in Section 4.3. To summarize, its main idea is to send a graph from the client in multiple parts, and in each part, mark nodes that satisfy a specific condition for the first time. The server then receives this graph and behaves differently for these marked nodes. We created a new attribute `Replace` which is used by this algorithm to mark nodes. It's a part of the `DataflowObjectFormats` class.

Client

The `DictionaryDataflowGeneratorReader` class is used to create the data flow graph sent to the server after analyzing data dictionaries. We refactorized this class so it could be subclassed. It newly contains two protected methods `addStepForFutureProcessing` and `shouldContinue` that can be changed by a subclass.

The new class `IncrementalDictionaryDataflowGeneratorReader` is the subclass that implements the solution from Section 4.3.4. Both previously mentioned protected methods are changed. First, this new subclass changes the algorithm from the DFS to the BFS by using a queue instead of a stack in the `addStepForFutureProcessing`. Second, this new subclass sends a part of the created graph to the server only if the condition from Section 4.3.4 is satisfied by changing the `shouldContinue` method.

The last change in this new subclass is that whenever any node is for the first time sent to the server with all its children, this node is marked with the new `Replace` attribute.

Server

On the server, the merging is handled by the `StandardMergerProcessor` class. Whenever this class merges a node with the `Replace` attribute, it should delete all its children that are not in the new revision. However, when merging the node, this class does not know which of these children are in the new revision and which are not because they have not been merged yet. Therefore, this class stores all these children in the `nodesToDelete` map, and whenever a node from the new revision is merged, it is removed from this map. Consequently, after all the nodes are merged, this map contains all children of nodes marked with a replace flag that aren't in the new revision anymore. Then the `StandardMergerProcessor` class deletes all these children in its new method `finishMerge`, which takes place after everything is merged.

5.6 Analyze only changed files

Every incremental update should start by analyzing changed files. However, because there is currently no incremental extraction, both changed, and unchanged files are extracted. Consequently, incremental updates need to recognize which files have changed since the last update and which did not.

The `DdlBundle` class is responsible for extracting Oracle scripts. It newly has the `shouldCreateHashes` flag. If on, this class uses the newly created `FileHashes` class to create and save a hash for every extracted Oracle script. There are another two new classes, `HashFileLine` class that represents all information about a single file and `HashFileParser` that loads saved hash information from a file.

Before the extraction, the `DdlBundle` class loads hashes of all files extracted in previous updates. This class creates a hash for every extracted file during the extraction and finds if it is a changed file (i.e., either a new file or a file with a changed hash). After the extraction, this class saves all information about every extracted file to a text file. This includes file name, file hash, and a changed flag. This saved text file is used at the beginning of the next extraction to load hashes. Note that this process must happen not only for every incremental update but also for every full update.

Later during an incremental update analysis, the analysis should analyze only changed files. This is done using a `ChangedFilesFilter` class, which stops all unchanged files from being analyzed. To do that, this filter class internally loads and uses the saved text file with hashes.

5.7 Analyze files returned by the server

The full update analyzes all extracted files. However, during an incremental update, analysis of a segment can return a segment ids of other segments. The incremental update needs to analyze the source files of these segments as well. This is quite a different behavior that had to be implemented.

The `FileReader` class is responsible for choosing which files are analyzed by providing all extracted files. We implemented the new `ReaderWithQueue` class that not only provides extracted files using an inner `FileReader`, but it also has a queue and provides files from this queue.

This queue is filled by the `MergerWriter` class that parses responses from the server. And if a response contains a request to analyze a segment, then this class adds the source file of that segment to the queue in `ReaderWithQueue`. This means that later the reader provides this file for analysis.

The server can request a file to be analyzed multiple times, but it is enough to analyze each file only once. We have created the `NonDuplicateFileFilter` class which stops a file from being analyzed if the file was already analyzed in the same revision. The `ReaderWithQueue` uses this filter, so every file is analyzed at most once. This reader also uses the `ChangedFilesFilter` class we have mentioned previously so that only unchanged files are analyzed.

6. Evaluation

This chapter describes the evaluation of results for the whole thesis, including both the analysis and the implementation. The first part of this chapter describes all automated tests to show that the implemented solution works correctly. The second part introduces performance testing, its test data, and test results to show the performance benefits of the implemented solution. This chapter ends with the limitations of the presented incremental update solution and a discussion of possible future extensions.

6.1 Tests

Because the code was developed using almost exclusively test-driven development, there are unit tests for every change made. All unit tests are created using `JUnit` testing framework [7], all Java classes containing them have suffix `Test` and can be found in the attachment of this thesis. Almost all unit tests follow the same `nameOfTestedUnit_stateUnderTest_expectedBehaviour` name convention. This way, we can easily know what each test is testing just from its name.

Server unit tests

First group of unit tests are server tests, they can be found in attachment in the package `manta-dataflow-repository-extensions-neo4j` and they all share the common parent class `IncrementalUpdateBaseTest`. This class creates a new minor revision for each server test. Then the test merges one or more merge input files. There are 18 different testing merge inputs in the server folder `resources`. After merging, tests usually check if the generated flow graph contains expected nodes and edges.

First server test class is `ModelChangesTest` which tests model changes described in Section 5.2. Second server test class is `EdgeRestorationTest` testing edge restoration implementation from Section 5.3.2. Third test class is `NodeRemovalTest` that tests node removal implementation from Section 5.3.3. And the last server test is `EdgeDeletionTest` class which tests edge deletion changes described in Section 5.4.

Client unit tests

The second group of unit tests is client tests that share the common parent class `AnalyzeFileTest`. Using this class, these tests read a source code file as their input, analyze the file using the client and save the created merge input in the file instead of sending it to the server. In the end, these tests check the saved merge input file and assert that its content is correct.

First client test class is `AbstractSourceFileTaskTest` that tests the implementation of model changes described in Section 5.2. Other client test classes are `NodeFlagTaskTest` and `DictionaryDataflowTest`, both of them testing implementation of changing the marked node from Section 5.3.1.

Third group of unit tests are client tests testing newly created client filters and reader described in Section 5.7 and Section 5.6. These tests are stored in classes `ChangedFilesFilterTest`, `NonDuplicateFileFilterTest` and `ReaderWithQueueTestdirectly`. They directly test the correct behaviour of methods of tested filter and reader classes.

There are two other new test classes. `OracleCreateHashesTest` is the first one, it covers creation of hash file with hashes for every extracted file in Section 5.6. `IncrementalDictionaryDataflowGeneratorReaderTest` is the second one, it covers changes in the creation of the data dictionary merge input explained in Section 5.5. These tests have a data dictionary file content as their input, they analyze it, then assert that the generated merge input is correct.

End to end tests

End-to-end tests were performed manually both on the test data for performance testing and other inputs. We originally wanted to automatically compare the generated graph by the newly implemented incremental updates with the generated graph by the current unchanged version of MANTA. However, this turned out to be much harder to implement because of all the objects that incremental updates add to the flow graph (e.g., dependency edges or source segment data). Based on this, it was decided not to automate this process.

6.2 Performance testing

The implemented prototype was tested by performance tests to compare the effectiveness of incremental updates compared to the original full update implementation. On top of that, we also tested speed differences between the original full update and the current full update that computes a few additional things needed for the correctness of later incremental updates.

Testing was performed on the following configuration:

- Operation system - Windows 10 Pro (64 bit)
- Processor - 11th Gen Intel(R) Core(TM) i7-1185G7 @ 3.00GHz 1.80 GHz
- RAM - 16 GB
- Manta Flow version - 36.0.0 (updated with incremental updates code)

Test cases

All the cases use the same test scenario:

1. Empty the whole database.
2. Perform full update on Oracle test data (revision 0).
3. Change $X\%$ of test data.
4. Perform incremental/full update (revision 1.1/2).

In test cases 1 and 2 is performed a full update on all scripts. Then, 1% of scripts is changed, and the full update is performed again. It does not matter how many scripts are changed as the full update analyzes all the scripts anyways. In the first test case, the old unmodified full update is used. In the second test case, the slightly modified full update from this thesis is used, so it computes a few necessary things, so the following incremental updates work correctly.

In test cases 3 to 6 is at the beginning again performed a full update on all scripts, and then 1%, 3%, 10%, or 100% of input scripts is changed. After the change, the incremental update is performed.

All test cases use the same test data. It is a collection of 500 Oracle procedures.

Test results

In all test cases, we measured the time spent updating the graph database on the server. We did not measure the times necessary for extraction, data flow analysis, input creation, sending input to the server, and other processes connected with the whole update. This is because updating the graph database on the server is the current bottleneck for processing large inputs.

Measuring of each test case was performed five times. In Table 6.1 we can see the summarized average merge times of the second update (second revision) for every test case. We can also see the same data visualized differently in the graph in Figure 6.1.

Test case number	1	2	3	4	5	6
Time [s]	10.415	11.074	0.278	0.628	1.629	16.856

Table 6.1: Performance test results

Let us first compare data from the first two test cases. We can see that the full update with the new changes from this thesis is about 6% slower. This is an expected slowdown because a full update has to newly process a few additional things like source segments for every edge and node or create dependency edges described in Section 4.2.8.

If we look at the last four test cases, we can see that the time of the incremental updates depends on the number of changed inputs as expected. There is some

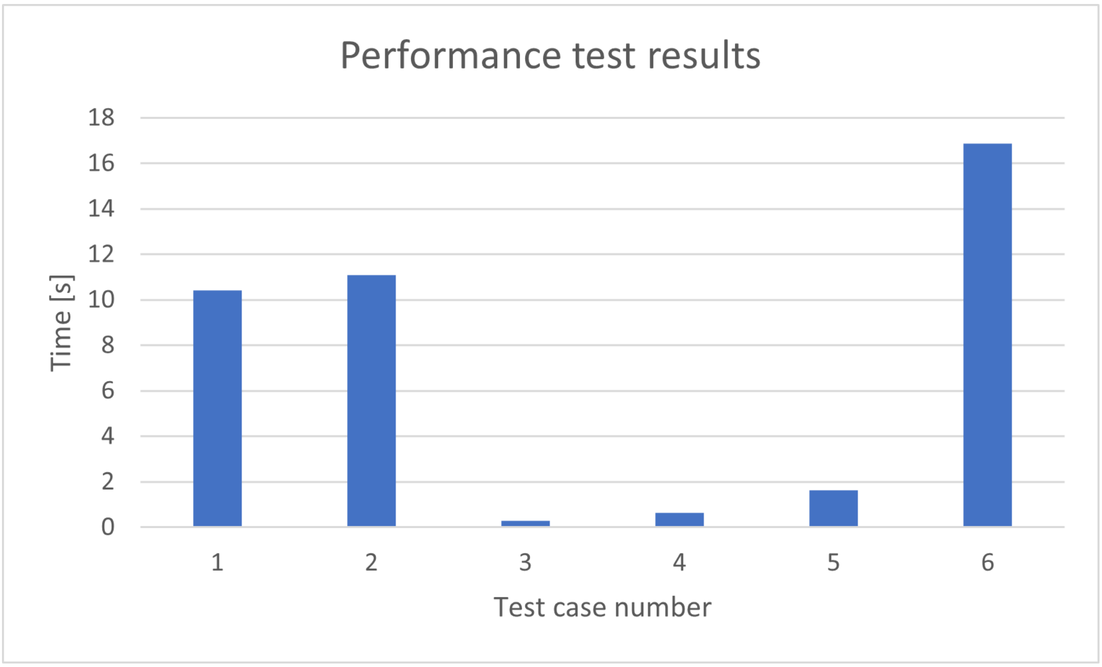

Figure 6.1: Performance test results

expected overhead compared to the full update running only for the changed inputs. This overhead depends on the number of changed scripts and is between 50% (for the higher amount of changed scripts) and 170% (for the lower amount of changed scripts). The size of this overhead is a good result because, for example, in the test case 3 (and 4) that are close to the expected real-world use cases, the incremental update is 37 (and 17) times faster than the full update.

Comparing the results of test cases 2 and 6 shows that the incremental update is about 50% slower than the full update when every input is changed. Although the incremental update is not recommended in such a case, it is still possible to use it.

6.3 Limitations

Some situations are not handled correctly when performing an incremental update, and they most likely cannot be fixed. The reason is that some dependencies between segments are too complex to be captured. These situations are edge cases, but they will happen. We need later to decide how severe these situations are and how much of an issue is that they will sometimes happen. This largely depends on expected use cases for incremental updates.

In the following text, we present two issues we have discovered so far. There can be, of course, more similar issues. We will need to test incremental updates on complex test cases and compare the resulting flow graph to the full update result to find if they exist. However, we can do that only when all the missing features of incremental updates are implemented.

Ambiguity resolution

This first issue relates to the fact that a reference in *SQL* does not always need to be unambiguous. Consider two segments in Figure 6.2. The first segment contains the script creating the table `t1`. The second segment contains the script creating the table `t2` and a select. The select is valid (even though it might not make much sense semantically), `y` in its subquery is resolved to `t2.y`. Note that `t1.a` is never referenced in the second segment, so there are no edges at all between the nodes in the two segments.

```
// Segment 1:
create table t1 (a int);

// Segment 2:
create table t2 (x int, y int);
select x from t2
  where exists (select 1 from t1 where y = 0);
```

Figure 6.2: Ambiguity resolution example - before changes

Now imagine the first segment is changed in a new revision as shown in Figure 6.3, it now has a second column `y`. This changes the way `y` in the second segment should be resolved - the new column created an ambiguity with `t2.y` in the select, and this ambiguity is resolved in this case to `t1.y` (the rule of a closer scope - `t1` is in the inner scope of the subquery). So the select should now create an edge from `t1.y`, but the second segment did not change, and there are no edges between the two segments that could be used to trigger the analysis of the second segment. The second segment will not be analyzed in the new revision, and the lineage will be incorrect.

```
// Segment 1:
create table t1 (a int, y int);
```

Figure 6.3: Ambiguity resolution example - after changes

Deduction miss

The second issue is called deduction miss. Whenever resolving an unknown reference, the analysis deduces an entity. However, there are often more possibilities of what the referenced object could be. In such a case, the analysis chooses one of the possible alternatives. This is done by assigning priorities to different alternatives based on some heuristics. For example, let us imagine a segment with a simple select, as we can see in Figure 6.4.

```
SELECT a FROM foo;
```

Figure 6.4: Deduction miss example

The analysis cannot resolve `foo` to any known entity, so it can either guess it to be a table or a view. It decides to deduce it as a table (the heuristic being that tables are more frequent in databases than views). However, if an incremental update later adds a definition of a view `foo`, it will not trigger a reanalysis of the segment from the example because there is no connection between the example segment and the new segment.

This could be theoretically solved by adding a dependency edge to parents of all entities the reference could be resolved to, but that could be tens of edges per every deduced entity. This is rather an unrealistic implementation, and there is probably no better way to solve this issue.

6.4 Future work

As the topic of incremental updates is a huge one, many improvements can be made in the future. The first group of improvements is what is already analyzed in this thesis but not yet implemented. This group contains mostly of node creation from Section 4.2.8, shared segments from Section 4.2.9 and post processing from Section 4.5. The second group of improvements is what has not been analyzed in this thesis. We are introducing the most interesting ones in the following text.

Incremental extraction

One of the requirements of this thesis is the same extraction as the full update. This means that both changed and unchanged data are extracted. However, it is unnecessary to extract all these files again, but we could extract only changed files. The first challenge of incremental extraction is to find what has changed and extract it. We need support from the scanned technologies to successfully do this, but many technologies already provide an API to find or extract only changed files. The second challenge is needing access to some unchanged files from the last revision when using incremental extraction. This is not currently possible because extracted data are not persistently saved, and they can be deleted at any time. Our graph database is the only persistent storage, which is not a great solution for saving vast amounts of text files.

Incremental updates for reporting technologies

The main focus of this thesis was on performing incremental updates for a database technology (which was an Oracle database). However, only half of MANTA scanners are for database technologies. The second half is for reporting technologies (e.g., PowerBI or Cognos). In general, this topic should be more straightforward for two reasons. First, there is usually no central data element like a data dictionary in databases. Second because boundaries between segments are usually very strictly regulated and defined.

We have created a concept of $L1$ dependencies for reporting technologies in

Section 3.2.1 and how to use them by the client algorithm in Section 4.8. But there is much analyzing, designing, and implementing yet to be done.

All-technology use case

The all-technology use case, which allows incremental updates to update multiple technologies together, is already mentioned in Section 4.8. A big part of this topic was already analyzed, and it should be possible, but some design and the whole implementation are yet to be done. In the full update, analysis of different technologies is independent. The biggest challenge of the all-technology use case is changing this paradigm because it is no longer true with incremental updates. That is because a change in one technology can trigger a reanalysis of an unchanged segment from another technology. Another interesting part of this use case is integrating it in the GUI and how the user can configure it.

Deduction

Another topic that was partly analyzed is deduction. It was mentioned in Section 6.3. Whenever a reference is not found, a new deduced object is created in its place. The first big challenge is that the created deduced object can have many different types (e.g., a procedure parameter, a whole table, only one table column, etc.) and can be in many different places in the flow graph. The second challenge is that multiple segments can generate the same deduction nodes. And the last one is that one segment can, for example, create a deduced table, and a second segment can create only a deduced column in this deduced table. This is similar to shared segments but more complex because there is no single shared root node.

7. Conclusion

This thesis presented how the data lineage analyzer can be improved using incremental updates to analyze only a fraction of all input files while still producing correct results. We changed how the whole analysis is done by changing the granularity of the analysis to much smaller pieces from scenarios to segments. This allows the analysis to much more precisely pick which inputs it analyzes and which not. We also introduced new terms like source segment, the segment root node, and segment id node to have a shared vocabulary for incremental updates.

We designed a new automatic detection of which segments changed since the last update, and we improved the merge algorithm so it can handle any input changes. The new merge algorithm recognizes when an unchanged file could generate a different data lineage using new concepts like node removal and node creation. This is possible thanks to the addition of the new source segment concept that tracks for every node and edge which segment created them. Using the new client algorithm for incremental updates, MANTA now can analyze only changed segments and a few unchanged segments instead of wasting time by analyzing inputs that would generate the same data lineage.

This topic was not analyzed in a vacuum but as a part of an already existing and complex environment. Consequently, we needed to update other existing concepts like data dictionary updates or post processing and make them incremental. We needed to consider the details of the current implementation, for example, by introducing shared segments.

We also implemented a prototype for the MANTA Oracle scanner that contains most of the new ideas introduced in this thesis. It was tested for both the correctness and the performance. As incremental updates are a huge topic, we plan to continue improving the prototype and later add it to the MANTA production environment.

Bibliography

[1] About the manta data lineage platform. `https://getmanta.com/about-the-manta-platform/`, Apr 2022.

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

[3] Titan - home. `http://titan.thinkaurelius.com/wikidoc/0.4.4/Home.html`, May 2022.

[4] Neo4j graph data platform. `https://neo4j.com/`, Apr 2022.

[5] H2 database engine. `https://www.h2database.com/html/main.html`, May 2022.

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

[7] Junit 5. `https://junit.org/junit5/`, May 2022.

A. Attachment structure

The attachment of this thesis contains mostly source files created or updated during implementation of the Oracle prototype for this thesis. The sources files are in the `sources` folder. The `sources` folder contains 9 additional folders, each representing a single module with one or more source files. The modules are further divided into the `src` folder with the actual code and the `test` folder with unit tests. The `test` folder also optionally contains the `resource` folder with test resources, for example maven test configurations or test inputs.

Apart from the source files, the attachment contains a copy of this thesis and the `README` file describing the attachment structure.