Conflict Detection, Avoidance, and Resolution in a Non-Linear RDF Version Control System

The Quit Editor Interface Concurrency Control

Natanael Arndt
Agile Knowledge Engineering and Semantic Web (AKSW)
Institut für Informatik, Leipzig University
Institut für Angewandte Informatik e.V.
Leipzig, Germany
arndt@informatik.uni-leipzig.de

Norman Radtke
Agile Knowledge Engineering and Semantic Web (AKSW)
Institut für Angewandte Informatik e.V.
Leipzig, Germany
radtke@informatik.uni-leipzig.de

ABSTRACT

The Semantic Web is about collaboration and exchange of information. While the data on the Semantic Web is constantly evolving and meant to be collaboratively edited there is no practical transactional concept or method to control concurrent writes to a dataset and avoid conflicts. Thus, we follow the question, how can we ensure a controlled state of a SPARQL Store when performing non transactional write operations? Based on the Distributed Version Control System for RDF data implemented in the Quit Store we present the Quit Editor Interface Concurrency Control (QEICC). QEICC provides a protocol on top of the SPARQL 1.1 standard to identify, avoid, and resolve conflicts. The strategies *reject*, *branch*, and *merge* are presented to allow different levels of control over the conflict resolution. While the *reject* strategy gives full control to the client, with *branch* and *merge* it is even possible to postpone the conflict resolution and integrate it into the date engineering process.

CCS CONCEPTS

• **Information systems → Query languages**; **Version management**; **Resource Description Framework (RDF)**; *Data management systems*; *Middleware for databases*; RESTful web services.

KEYWORDS

versioning, transaction, concurrency control, conflict detection, Git, RDF, Quit Store

ACM Reference Format:
Natanael Arndt and Norman Radtke. 2019. Conflict Detection, Avoidance, and Resolution in a Non-Linear RDF Version Control System: The Quit Editor Interface Concurrency Control. In *Companion Proceedings of the 2019 World Wide Web Conference (WWW '19 Companion), May 13–17, 2019, San Francisco, CA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3308560.3316519

WWW '19 Companion, May 13–17, 2019, San Francisco, CA, USA

ACM ISBN 978-1-4503-6675-5/19/05.
https://doi.org/10.1145/3308560.3316519

1 INTRODUCTION

Collaboration and exchange of information is deep-seated in our culture. With the Web, mankind has built a global communication system based on the exchange of data. Especially, the publication and exploitation of Linked Data is gaining popularity as it is apparent in the *Linked Open Data Cloud* with 1,234 datasets and 16,136 links between the datasets[1] also known as the Semantic Web. On the Semantic Web knowledge bases and ontologies are used to encode a common understanding of people on a distributed network. Collaborative editing of a common knowledge base is an evolutionary process of creating new versions of the knowledge base. In order to manage collaboration on Linked Data we face the problem of distributed versioning and parallel access to the knowledge base. Recently, the versioning of RDF data has gained attention [6, 7]. Evolution of data and contributions by distributed parties introduce the problem of overlapping update operations, called concurrency. The problem of concurrency control was extensively studied in the context of conventional databases and collaborative file exchange systems [4, 5, 8, 20, 21]. But the architecture of Semantic Web Applications is mostly Web oriented and similar to the architecture of Ajax/REST Web Applications. Web Applications as well as Semantic Web Applications consist of three layers: *persistence layer*, *data interchange & transaction processing*, and *user interface* [16]. The problem of distributed consistency is studied with regard to the transfer of distributed states in Web Applications [11, 15, 19].

However, the role of the persistence layer and thus the data interchange in Semantic Web Applications is different to the architecture of conventional applications. On the informal *Semantic Web Layer Cake*[2,3] the bottom layers cover the abstract representation to encode data as triples and their serialization (cf. *persistence layer* [16]). To query the data model the SPARQL 1.1 Query and Update Language is standardized [10, 12] (cf. *data interchange & transaction processing*). The top most layer represents *User Interface & Applications* that are used to read and change the data. The RDF data model allows Semantic Web Applications to encode the schema and the application logic within the abstract representation on the persistence layer [13, 14, 16]. This semantically rich

[1] http://lod-cloud.net/; As of June 2018
[2] https://www.w3.org/2007/03/layerCake.svg
[3] https://natanael.arndt.xyz/notes/semantic-web-layer-cake

data, stored in an abstract data model allows to handle concurrency on an abstract level while providing more flexibility to resolve conflicts. But, only little research is performed with regard to the management of concurrent updates on RDF data. For SPARQL 1.1 (cf. [22]) there is no standardized transaction interface available even though several approaches[4,5] exist. The related work mainly tries to adapt the protocols from conventional databases [17, 18] to SPARQL 1.1 by striving for serializability. Innovative concepts that take the possibilities of distributed versioning systems and the abstract concept of RDF knowledge bases into account are missing.

In this paper we present the Quit Editor Interface Concurrency Control (QEICC), a method to handle conflicting write operations on a SPARQL 1.1 Update interface of a distributed multiversion RDF quad store. The QEICC is build on top of *Quit*, our previously published distributed version control system for RDF knowledge bases [2, 3]. The Quit Store maintains multiple *diverged* version logs (*branches*) of an RDF dataset while each branch can represent a different state and latest version of the dataset. The branched versioning logs allow to postpone the *reconciliation* of diverged versions and to incorporate it into the data engineering process. Further, it is possible to retrieve any version on the Quit Store using its random access feature. With the QEICC extension we exploit the properties of the distributed version control system of the Quit Store. Using the branching system we postpone the conflict resolution as we are not constrained to reach serializability. The reconciliation system is used to still strive for an eventually conflated and serialized versioning log. In addition we exploit the commit system and the random access to arbitrary versions of the Quit Store to provide snapshot isolation for the operations. As we use these features, with QEICC we detect, avoid, and resolve conflicts of update operations based on a snapshot identification. To give full control to editing interfaces within the data engineering process we provide the three optimistic concurrency control strategies *reject*, *branch*, and *merge*. The strategies differ in their conflict handling which allows us to shift the resolution of a conflict between the data's creator, automatic processes, and collaborators to perform a data-centric reconciliation.

The remainder of this paper is structured as follows. First, we give an overview on the state of the art for data base systems and Web Applications as well as Semantic Web Applications in section 2. Second, we provide a problem description in section 3. The main contribution of our work is the Quit Editor Interface which is presented in section 4 with the three conflict resolution strategies *reject* in section 4.1, *branch* in section 4.2, and *merge* in section 4.3. The strategies are compared regarding their individual properties and the instance of control in section 5. Finally, we conclude the paper in section 6.

2 STATE OF THE ART

In the following we take a look at the related work. First we will provide a historical walk-through on conflict detection, concurrency, and transaction management for data base systems and derived concepts for RDF data stores in section 2.1. Afterwards we look into approaches for state transfer between clients and data stores respective back-ends in the application engineering area and in particular for Semantic Web Applications in section 2.2.

2.1 Database Management Systems

The problems of *concurrency control* and the *recovery problem* were broadly discussed by Bernstein et al. [5].

> Systems that solve the concurrency control and recovery problems allow their users to assume that each of their programs executes atomically – as if no other programs were executed concurrently – and reliably – as if there were no failures.

This abstraction is called *transaction* and an algorithm that executes the transactions atomically is called *concurrency control algorithm*. The concurrency control algorithm is implemented by executing concurrent interleaving transactions one after another *to give the illusion that transactions execute serially*. Interleaving executions of transactions that have the same effect as serial executions are called *serializable*, which is considered correct because they support the *illusion of transaction atomicity*.

Berenson et al. [4] introduce the *snapshot isolation* type. This isolation type allows reads or writes to be executed on a snapshot which ensures isolation from other transactions but is not serializable. Following from the isolation guarantee snapshot isolated write operations can only be performed on distinct data items.

The problem to ensure consistency across all replicas while clients are disconnected from the network was discussed for the *Coda* file system by Satyanarayanan et al. [21]. If a client is disconnected, be it intentionally or not, the client can still perform local changes in the cached file system. When the client is reconnected a reintegration process starts that tries to execute the cached updates on the file system of all replicas. If conflicts between updates are detected the conflicting files are temporarily stored in a *covolume* until the conflict is resolved by a user.

Demers et al. [8] present the *Bayou System*, a client/server platform to replicate databases for the usage on mobile devices like *personal digital assistants*. The system propagates all writes through a peer-to-peer protocol to a *primary* server which accepts writes to be *committed*. All other write operations on *secondary* servers that were not yet committed are *tentative* until they are serialized on the *primary* server and the order of writes propagated to the secondary servers. If conflicts arise *mergeprocs* are employed that understand the semantics of the data format and domain and can reconcile the conflict.

Another system to reconcile transactions performed on mobile clients is presented by Phatak and Badrinath [20]. Similar to the *Bayou System* all transactions need to be transmitted to a server to be globally committed. If a disconnected client performs local transactions, on reconnection they need to be tested for serializability and are rolled back on conflicts. Further they discuss the weakening of the serializability guarantee which can be enabled by a semantic aware reconciliation and by providing snapshot isolation (cf. [4]). To implement the multiversion reconciliation the conflict detection and resolution is decoupled, while the server is responsible for the conflict detection the client provides the conflict resolution function.

[4] https://jena.apache.org/documentation/txn/
[5] http://people.apache.org/~sallen/sparql11-transaction/

Muys [17] provides a discussion about a concurrency control protocol for multiversion RDF data stores. In contrast to relational or object databases in an RDF store the smallest "cell" that can be considered is the entire graph. If multiple graphs are present and can be targeted by an update, the whole RDF dataset needs to be considered as a unit of change. This results in a single global write lock when applying traditional concurrency control protocols on an RDF data store and thus provides very little concurrency. To avoid a single global write lock the usage of predicate locking is proposed with the basic graph pattern (BGP) as predicate that is tested against the update set of a transaction that needs to be serialized. Still the resolution of conflicts is a problem. For this purpose the *single version optimistic concurrency control* protocol is adapted to a multiversion store by specifying the *multiversion optimistic concurrency control* protocol. The multiversion protocol allows to exploit snapshot isolation and thus to execute read-only operations from the validation protocol.

Neumann and Weikum [18] are following a similar approach and extend the RDF-3X store to x-RDF-3X. The RDF store provides versioning with the ability for time travelling queries. The store supports snapshot isolation and full serializability with help of predicate locks based on the query BGPs. Due to the snapshot isolation no locks are needed for read operations. To avoid too small locks or unnecessary large locks a lock splitting algorithm is employed. Update operations are performed in a per-transaction workspace that is merged into the differential indexes if a savepoint is issued.

2.2 Application Engineering

Ousterhout and Stratmann [19] discuss the state problem in Web Applications using asynchronous AJAX requests. They analyze and verify possible solutions for managing states on the client side or on the server side in a Web Application. The authors conclude that both, client side and server side states have drawbacks. Client side state management has *overheads for shipping state between browser and server, and it creates potential security loopholes by allowing sensitive server state to be stored in the browser.* Whereas the server-based approach *introduces overheads for saving state as part of sessions, and it has garbage-collection issues that can result in the loss of state.* The authors prefer the usage of a server-based state management over that of a browser-based and predict upcoming challenges in state management of Web Applications.

Pardon and Pautasso [11] specify a protocol to support atomicity and recovery over distributed REST resources. The authors contribute to the debate in the REST community to whether or not transaction support is needed. The presented approach references and is similar to the position paper by Helland [15]. The need for the protocol is motivated by a business use case on booking two connecting flights from two different airlines. The presented approach is based on a Try-Cancel/Confirm (TCC) pattern using (a) an initial state, (b) a reserved state and (c) a final state. The reserved state (b) is called *tentative* in [15]. To form a transaction an arbitrary number of REST services are loosely coupled. A transaction is valid if all reserved states (b) are confirmed, after this the final state (c) is entered for each service. Compared to the two-phase commit lock the authors point out, that the TCC-approach *offers higher-level semantics and does not hold low-level database locks.* The participants *do not block any other work other than the one affected by the business resources they reserve.* Both approaches [11, 15] mention that an uncertainty is left if one of the loosely coupled systems fails before all systems have reached the final state. The management of uncertainty must be implemented in business logic to counter this issue.

Web Applications as well as Semantic Web Applications (SWA) consist of three layers *Persistence Layer*, *Data Interchange & Transaction processing*, and *User Interface* [16]. Martin and Auer [16] present a categorization model for Semantic Web Applications. Following this categorization model our focus is on intrinsic and extrinsic producing SWAs while the level of user involvement, semantic richness, and semantic integration is left open. In an empirical study performed by Heitmann et al. [13, 14] all SWAs have a graph access layer, an increasing amount of SWAs is concerned with data creation, and some applications even provide structured data authoring interfaces. The data flow to allow collaborative knowledge acquisition, visualization, and creation requires for a management of concurrent and possibly conflicting operations. The above presented research on non-semantic Web Applications is mostly concerned with distributed state management. Methods like Flux[6] can be used in client applications to bundle the data flow towards the back-end within an application. A store object ensures that all view component's requests to the back-end system are executed and controlled in a single place which can represent a client side state. While the issue of communicating the global state of the store between back-end and client application is underrepresented in research. A reason for a lack of research in this are in the Ajax/REST community might be the fact, that Ajax/REST applications usually encode parts of the business logic in the back-end. In contrast to that, SWAs strive to separate the data's semantics and implementation by searching for a generic semantic representation. With the RDF data model SWAs are able to encode semantics in the data store and thus are able to directly communicate with the RDF store as persistence layer.

3 PROBLEM DESCRIPTION

In user facing applications, in particular in Web Applications, the user interface is constructed of many components. Each of the user interface components might send read requests to the back-end system. When the user performs any action that induces an update operation the user might take into account any information displayed in the user interface for his update decision. These requests are sent by the individual components or can be coordinated by a method like Flux. In many traditional systems the back-end system implements parts of the business logic and can thus manage the state of the front-end in interaction with the store object. For a Semantic Web Application the data store can directly serve as back-end and be accessed using the SPARQL 1.1 Protocol [9]. In the following we demonstrate two scenarios of clients interacting with a SPARQL 1.1 store to manage a todo-list.

[6] https://facebook.github.io/flux/docs/in-depth-overview.html#structure-and-data-flow

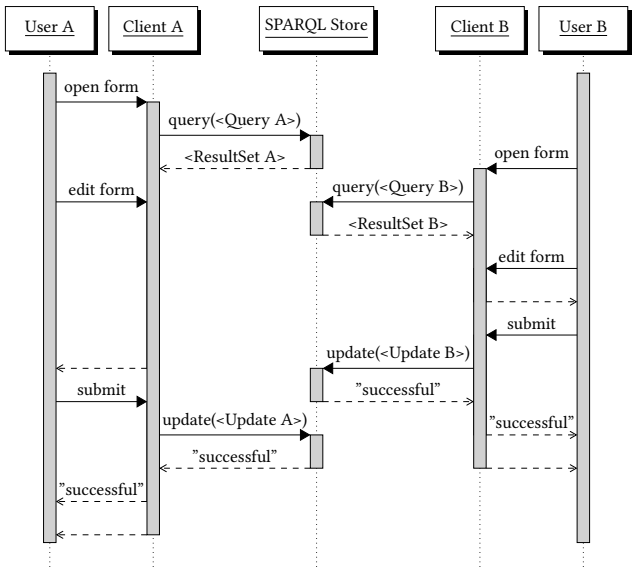

Figure 1: Sequence diagram of two overlapping update operations which result in an uncontrolled state of the shared data store.

```
@prefix ex: <http://example.org/> .
ex:garbage a ex:Todo;
  ex:task "Take out the organic waste" .
```

Listing 1: The initial data in the triple store to be considered for scenarios 1 and 2.

```
PREFIX ex: <http://example.org/>
INSERT {
  ?task ex:status ex:completed
}
WHERE {
  ?task a ex:Todo
}
```

Listing 2: SPARQL Update operation *<Update A>* to mark the task as completed in scenario 1.

```
PREFIX ex: <http://example.org/>
INSERT DATA {
  ex:chain a ex:Todo;
    ex:task "Lubricate the bike chain."
}
```

Listing 3: SPARQL Update operation *<Update B>* to add a new task in scenario 1.

Scenario 1. In fig. 1 an example of two conflicting operations based on updating data through a form is depicted. We assume the content of the SPARQL Store are the statements shown in listing 1. Both clients retrieve the data from the store with a SPARQL Query and receive the respective result set containing the task to do. While user A performs the single task in the store and changes it to "completed" with the update operation in listing 2, meanwhile user B adds a new task with the update operation in listing 3. In the final state of the store both tasks will be marked as completed. If the operations were executed in the opposite order the state of the store would be different. Thus the two operations are not serializable and would cause the operations to abort with failure in traditional transactional concepts. But the operations do not fail in current SPARQL 1.1 implementations.

Scenario 2. Let us consider another scenario following the execution of fig. 1 and with the assumed initial content of the SPARQL Store as shown in listing 1. Both clients retrieve the data from the store with a SPARQL Query and receive the respective result set containing the task to do. While user A performs the task `ex:garbage` and changes it to "completed" with the update operation in listing 4, user B has changed the task description with the update operation in listing 5. Even though both update operations have taken care to identify the data item to change, in the final state of the store the changed task will be marked as completed even though this is not correct. Further, also the execution of the operations in the opposite order results in the same result. A traditional transactional concept could not identify the operations as conflicting.

```
PREFIX ex: <http://example.org/>
INSERT DATA {
  ex:garbage ex:status ex:completed
}
```

Listing 4: SPARQL Update operation *<Update A>* to mark the task as completed in scenario 2.

```
PREFIX ex: <http://example.org/>
DELETE { ?todo ex:task ?task }
INSERT {
  ?todo ex:task "Take out the organic waste
      ↪ and the residual waste"
}
WHERE {
  BIND (
    "Take out the organic waste" as ?task)
  ?todo ex:task ?task
}
```

Listing 5: SPARQL Update operation *<Update B>* to change the description of the task in scenario 2.

For SPARQL 1.1 no transactional concept is standardized. The SPARQL 1.1 Update language allows to build update operations of the form `DELETE {} INSERT {} WHERE {}` [10]. The `WHERE` part can be used to encode preconditions for the update operations and bind variables used in the update operations, as shown in the listings 2 and 5. But enclosing all preconditions inferred from all read operations that were used to compose the user interface is not a practical way to ensure it is not conflicting with other operations.

Further more, write operations in SPARQL 1.1 can also be performed without any precondition (`INSERT DATA`, `DELETE DATA`). Our question is: how can we ensure a controlled state of the SPARQL Store when performing non transactional write operations?

4 THE QUIT EDITOR INTERFACE

In the following we present the Quit Editor Interface Concurrency Control (QEICC) a method to handle concurrency of write operations in a distributed multiversion RDF quad store. QEICC is built on top of the Quit Store [2, 3]. We first introduce the essential preliminaries of the Quit concept. Then, we propose a fully backward compatible[7] protocol on top of the SPARQL 1.1 Protocol [9] that allows us to control concurrency. Based on this protocol we present the three conflict resolution strategies *reject* in section 4.1, *branch* in section 4.2, and *merge* in section 4.3.

The QEICC builds on our previously published distributed version control system (DVCS) for RDF knowledge bases Quit [2, 3]. The underlying DVCS allows not only linear versioning logs but also non linear *diverged* versioning logs represented by an acyclic directed graph of multiple *branches*. Because of the non linear versioning log we are not constrained to reach serializability of all performed operations. Instead, the store maintains multiple branched versioning logs while each branch can represent a different state and latest version of the database. Each version in the versioning log is identified by a unique hashed *commit id*, further each version references its predecessor as parent commit. A *branch* is identified by its name and is a pointer to the currently latest commit id in its versioning log. The store allows random access to arbitrary versions in the versioning log for query and update operations. Thus it is possible to start a new diverged versioning log at any existing version in the log. The latest version (*Head*) of each version log respective branch can be queried and updated through virtual SPARQL Endpoints. The store provides one virtual SPARQL Endpoint for each branch, i.e. http://localhost:5000/sparql/<Branch>.

The QEICC method exploits the commit system and the random access feature of the underlying DVCS to provide a state identification of the data store to the client. As shown in fig. 2 the state is transferred on query and update operations. A query operation is sent to the store following the standard SPARQL 1.1 Protocol [9]. The client selects the branch to work on by using the respective SPARQL Endpoint for the branch. In fig. 2 the selection of the branch is depicted by the parameter *<Branch>* of the *query()* and *update()* methods.

When the client sends a query operation to the store, the result set for the query is returned along with the additional two HTTP-Headers `X-CurrentBranch` and `X-CurrentCommit` (*<Branch>* and *<CommitID A>* in fig. 2). The `X-CurrentBranch` header field contains the name of the branch that was requested. The `X-CurrentCommit` header field contains the commit id of the current Head of the selected branch in the store. With these two header fields the store encodes the current snapshot state of the store at the time of the execution of the query.

When the client sends an update operation to the store it transmits the snapshot state that it has received with the last query

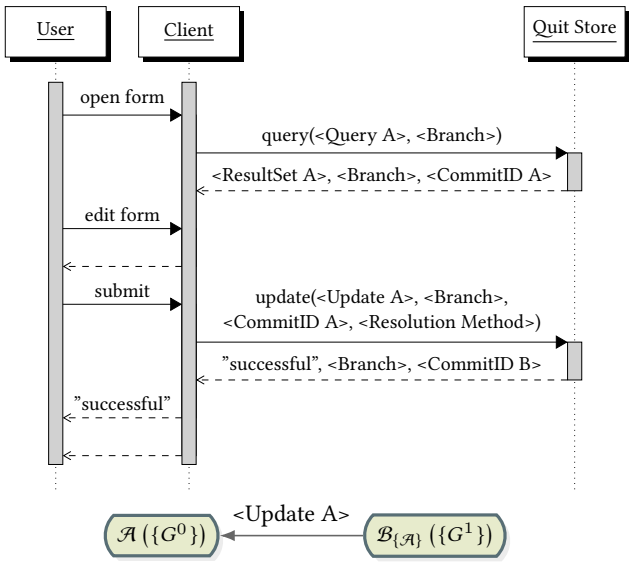

Figure 2: Sequence diagram showing the execution of an update operation with the parameters specified by the Quit Editor Interface. In the lower part, the commit graph is depicted which exists at the Quit Store after the update operation.

(`X-CurrentCommit`). To transfer the state from the client to the store the client sets the parameter `parent_commit_id`[8] along with the update operation to the store (*<CommitID A>* in fig. 2). This transfer of the state allows the client to express towards the store which is the currently most recent commit on the branch. Additionally the client selects a resolution strategy that the store has to apply in case of a conflict. The client sets the resolution method with the `resolution_method` parameter along with the `parent_commit_id` and update operation (*<Resolution Method>* in fig. 2). The `resolution_method` is one of the values `reject`, `branch`, or `merge`. When the store receives an update operation it compares the commit id of the currently latest commit (*Head*) on the branch with the value of `parent_commit_id` which was sent by the client along with the update request. By comparing the commit ids the store can identify whether a conflict exists or not.

No conflict. If no overlapping update operation was performed in the meantime both commit ids are equal. In this case the store commits the update operation and returns the commit id of the new Head of the selected branch. The now updated commit graph is depicted in the lower part of fig. 2. The state before the update is encoded in commit $\mathcal{A}$ with graph G^0, the update is performed using the update operation *<Update A>* which leads to the new state $\mathcal{B}_{\{\mathcal{A}\}}$ which was derived from $\mathcal{A}$ and points to its predecessor (for more details cf. [1, 2]).

Conflict. If the commit ids are different a conflict is detected. This case is depicted in fig. 3. The conflict is then handled according to the specified value for the `resolution_method`. To give control over the conflict resolution process we provide three optimistic

[7] The protocol still allows standard SPARQL 1.1 Query and Update operations, which of course will not benefit from the concurrency control features.

[8] The client uses HTTP form-encoded or query string request parameters to encode the values.

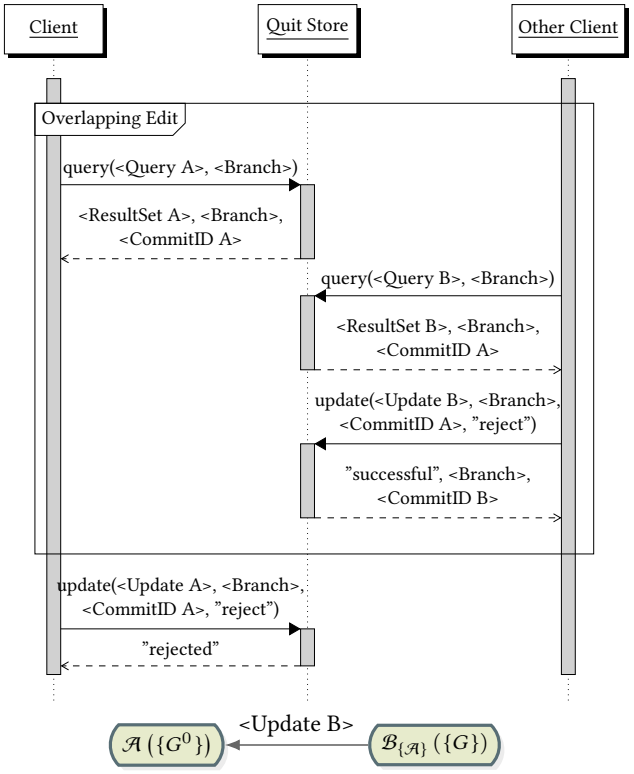

Figure 3: Sequence diagram showing two overlapping update operations, where the first operation is committed, while the second operation is aborted according to the selected strategy *reject*. The commit graph shows the state of the store after the successful update operation was committed.

strategies which differ in their conflict handling. In fig. 3 the previous queries and the conflicting update operation are enclosed in the *Overlapping Edit* block. For brevity, this block is not repeated but referenced in the later figures, figs. 4 to 6.

4.1 Reject

If a conflict is detected on the server side, choosing the *reject* strategy will abort the execution of the operation and will leave the store unchanged. This allows to perform update operations in a try-catch or trial-and-error manner. Figure 3 shows a rejected update operation performed by the client. The commit log in the lower part of fig. 3 shows the state of the store after the successful commit of the update operation from the other client. After the rejected operation it is up to the clients implementation how to proceed.

One possibility to resolve the conflict on the client side is depicted in fig. 4. The *Overlapping Edit* block is reused by reference from fig. 3. To resolve the conflict this method involves a new retrieval of the data that is been edited on the client side. Just after the update operation was rejected the query operation is performed again and returns now an up-to-date result set *ResultSet A'*. This result set is compared to the original result set in two alternative blocks. The top block is for the case that the result set of the

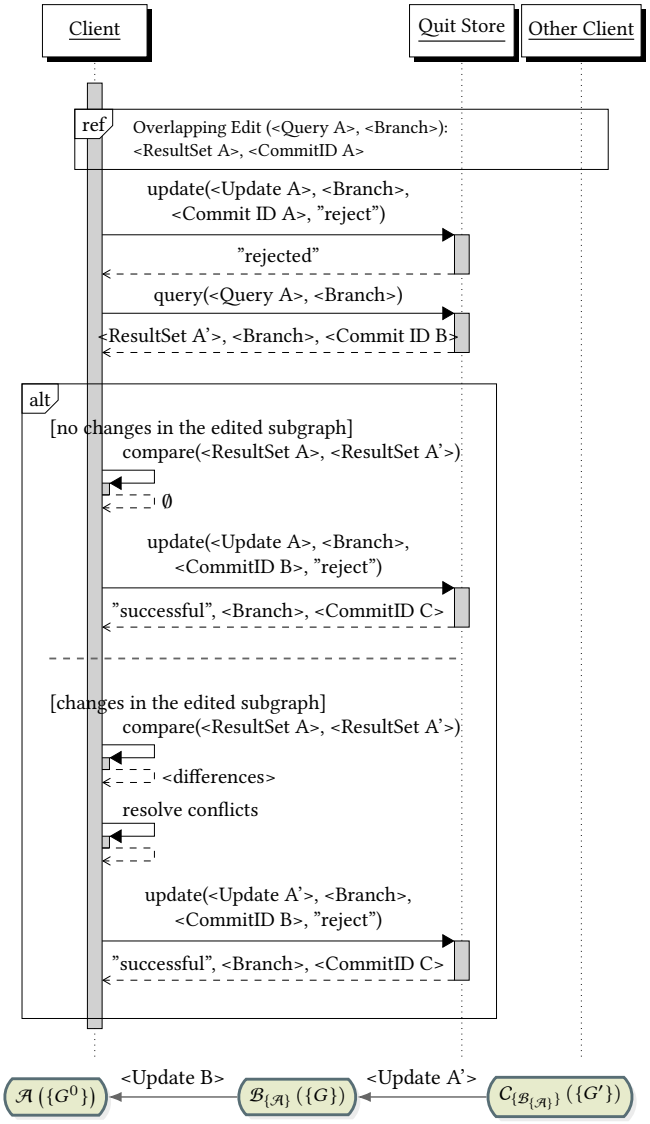

Figure 4: Sequence diagram showing the retry of an update operation after it was rejected due to an overlapping update operation. The alternative flows depict the first case that the result set of the original query did not change and thus the update operation can be resent. The second case in which the client needs to perform a resolution of the conflict before it can resent an updated operation. The commit graph shows the state of the store after the client has performed a resolution and the new update operation was committed.

query operation did not change and the bottom part for if the result set changed. If the result set is the same as the original result set the update operation is resent now with the updated commit id as parameter. By comparing the result sets we can ensure that the operation with which we are conflicting did not change the part of the graph (subgraph) that was queried with our query operation. This involves the assumption that the update operation was

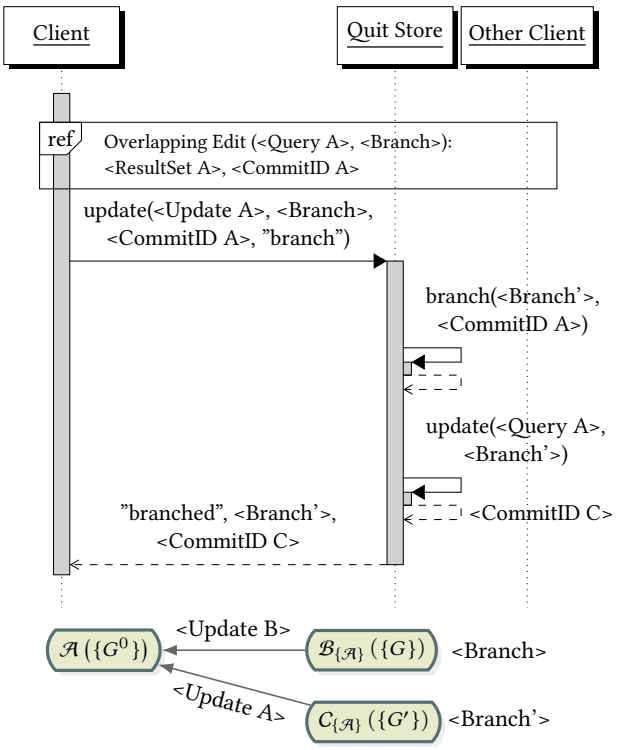

Figure 5: Sequence diagram of the overlapping update operation with the resolution strategy *branch*. The update operation is committed to a new branch that can be used to postpone the resolution. The commit graph shows how the state of the store has diverged into two branches after committing both update operations.

constructed based on the result set retrieved by the query operation *Query A* (cf. fig. 3). If the update affected the subgraph of our interest the client needs to resolve the conflict locally. This resolution can happen by informing the user about the changes and let the user review its input under the new circumstances and resubmit the form. After the conflict was resolved on the client side the adapted update operation can be submitted to the store along with the new commit id. The bottom part of fig. 4 shows the version log with the update operation by the other client (as shown in fig. 3) and the adapted update operation after the client side conflict resolution and the new commit C.

4.2 Branch

The *branch* strategy actually exploits the possibilities of our non-linear version log system. Figure 5 shows the execution of an update operation that conflicts and is resolved with the branch strategy. In contrast to the other two strategies the branch strategy does never fail. Instead, if a conflict arises a new branch is created that diverges from the original branch from the last common commit before the conflict on the branch. This divergence is shown in the lower part of fig. 5, each of the commits $\mathcal{B}$ and C are derived from the same original commit $\mathcal{A}$. In the sequence diagram the new branch is created with the *branch()* method. After the new branch was created the conflicting update operation is committed to the new branch *<Branch'>*.

This strategy allows fire-and-forget operations in a way that the update operation is committed, no matter if it conflicts or not. Also, this strategy allows to postpone a conflict resolution for instance if the correct resolution is not yet known or needs to be found based on a community discussion. This allows a data-centric reconciliation, in contrast to time based reconciliation, of the conflict using appropriate, e.g. domain specific, merge operations (cf. section 4.3 and [2]). Another asset of allowing a versioning log to branch is that it reflects the actual lineage of the update operations and changed dataset. By analysing the version log it is possible to reconstruct in which logical order operations were performed.

4.3 Merge

The *merge* strategy is similar to the branch strategy as it creates a new branch to commit the update operation when a conflict arises. In addition it subsequently performs a reconciliation of the diverged versioning log. To perform the merge, a merge method can be selected according to the methods supported by the respective system. The Quit Store has specified the merge methods *Three-Way-Merge* and *Context Merge* [2]. The two possibilities for a successful and a failing merge process are depicted in the two alternative blocks. If the reconciliation is successful a new commit is created as shown in the lower part of fig. 6, the commit $\mathcal{D}$ is derived from the two diverged commits $\mathcal{B}$ and C and combines the two branches again. If the reconciliation fails the update operation fails and the system ends up in the branched state. This means that the update operation is committed but the diverged states could not be automatically reconciled. The result of a failed merge operation is the description of the *merge conflicts* and the references to the new branch and commit. The merge conflict now needs to be resolved by the client or can be postponed to a later state. If the merge conflicts are resolved a new *merge* operation is performed by the client.

A merge method compares the two versions of the dataset as they are produced by the conflicting operations. This comparison usually takes into account the last common ancestor of the two newly created commits. In our case the newly created commits are $\mathcal{B}$ and C and the last common ancestor is $\mathcal{A}$. Each of the datasets $\{G\}$ and $\{G'\}$ is compared to the dataset $\{G^0\}$, this allows to break down the conflict from the level of overlapping operations to the actual data in the graphs. Besides the predefined merge methods *Three-Way-Merge* and *Context Merge* additional methods can be implemented to reflect special properties of a domain model. This allows a semi-automatic data-centric reconciliation of overlapping operations.

5 COMPARISON

The presented strategies have different properties with regard to the control over the resolution process. The resolution strategies are operating on a DVCS which provides non-linear versioning logs. Due to the distribution and because we are not limited to work on a linear versioning log we gain flexibility. The flexibility allows us to distribute the responsibility to resolve a conflict and to freely define a point in time at which a conflict resolution has to be performed.

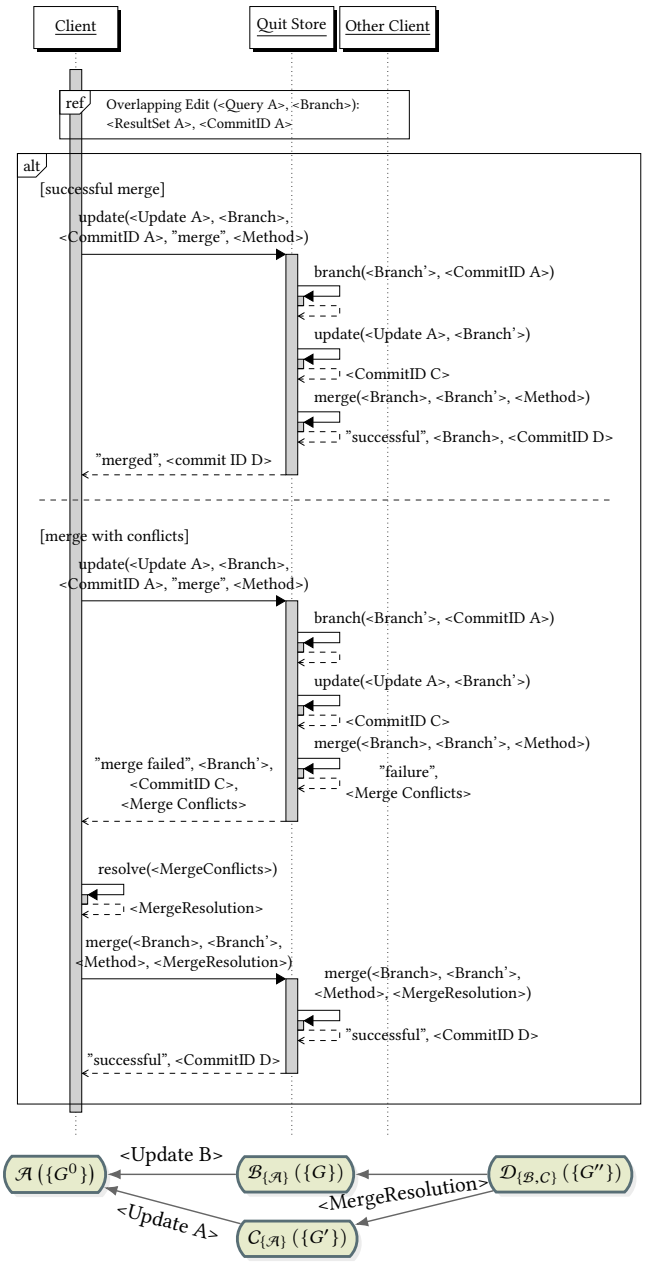

Figure 6: Sequence diagram of the overlapping update operations and server side resolution by merging the changes. The merge is successful in the first case, while in the second case the merge conflicts are reported to the client and need to be resolved on the client side. The commit graph shows the merged state of the store after a successful merge operation was performed with an optional *<MergeResolution>*.

In table 1 the three strategies reject, branch, and merge are categorized according to the aspects *performer of the resolution*, *point in time*, and *on failed resolution*. In the case that an operation is detected to conflict with a previous operation the server performs

Table 1: Comparison of the three conflict resolution strategies *reject*, *branch*, and *merge*.

Strategy	Performer of the Resolution	Point in Time	On Failed Resolution
reject	client/user	client's choice	retry
branch	any collaborator	postponed	n/a
merge	merge method	immediate	resolve/branch

the strategy that was specified by the client through the parameter `resolution_method`. For the *reject* strategy the server rejects the operation and thus gives the control back to the client or user to perform the resolution. The client can consequently decide at which point in time the conflict should be resolved, if also the resolution fails the client can retry or abort. With the method *branch* a new branch is created that is part of the non-linear versioning log, which enables any collaborator in the DVCS to perform a conflict resolution on this branch. The resolution of a branched conflict can be postponed to any later point in time. The *merge* strategy relies on a merge method that performs a predefined resolution procedure and can thus automatically control the resolution of the conflict. The merge method is applied immediately after the conflicting update operation was applied on a temporary branch. A successful conflict resolution is not always guaranteed by the merge method, depending on its implementation. In case of a failed resolution the merge method gives the control back to the client in the form of a *merge conflict*. If the client can resolve the merge conflict the resolution was successful, if not we end up at a branched state whose resolution is postponed in the same way as for the branch strategy.

In summary, the control of the resolution for the *reject* strategy is the client's responsibility, for the *branch* method the control can be distributed among collaborators, and for the *merge* strategy the control is at an automatic merge method but given back to the client or the collaborators in case of failure. We thus can shift the resolution of a conflict between the data's creator, an automatic process, and collaborators. The collaborator to resolve the conflict can be any participant in a team or a specified role according to a data engineering workflow. In contrast to traditional concurrency control systems when we deal with RDF data, we are able to incorporate the semantics of the data into the resolution process. In different usage scenarios different stakeholders, agents, or collaborators are responsible or capable to interpret the semantics of the data. With the presented strategies we gain flexibility to provide the control of a conflict resolution to the respective role in charge.

6 CONCLUSION

The usage of the SPARQL 1.1 Query, Update, and Protocol standards allows interoperability across software stacks in different usage scenarios. But this standard is not meant to deal with problems occurring in collaborative scenarios. The overall problem is how can we ensure a controlled state of the SPARQL Store when performing non transactional write operations. In the collaborative usage scenario we thus have additional requirements on top of the functionality covered by the SPARQL 1.1 standard.

With the Quit Editor Interface Concurrency Control (QEICC) we have presented a method to identify and avoid overlapping update operations respective conflicts. The conflicts can not only be detected on operations that would not be serializable in traditional transaction protocols, but also in cases where the operations could be executed in an isolated way. This is achieved by the possibility for the client to express towards the store which state it assumes to be the most recent respective which is the currently most recent commit on a branch. On top of this we have defined three strategies to handle the conflicts. The presented system allows to strive for a linear execution of the operations and only diverges if overlapping operations are detected. In contrast to conventional systems our system is based on non-linear versioning logs. The branched versioning logs allow us to postpone the serialization of operations (*reconciliation*) and incorporate it into the data engineering process. Due to the different strategies provided, the control over the resolution and reconciliation process can be assigned to the respective roles. The reconciliation can happen in a data-centric way that is aligned with the domain model.

As a side-effect, using the parameters defined by QEICC for conflict detection on updates the client can store the received values of the current branch and current commit id as the state of the store. This state representation allows the client to detect updates to the store while it performs query operations. This detection allows the client to trigger local updates of the user interface components when the store changes, to always present up-to-date information.

The QEICC mainly focuses on conflicts on local instances of a Distributed Version Control Systems for RDF data i.e. the Quit Store. The Quit Store provides, besides its local branching features, the possibility to run globally distributed networks of data repositories. Between the instances there is no steady connection, but committed update operations can be distributed among the instances via a synchronization protocol. This synchronization protocol transmits the individual commits as well as the branches. This synchronization of branches allows to even reconcile update operations that happened at different locations in the same way as following the *branch* strategy.

With our Quit system we hope that Distributed Version Control Systems find their way into the RDF data engineering and knowledge engineering domain to allow more collaborative and agile processes. With QEICC on top we provide an interface to control overlapping update operations and assign the resolution task to the responsible entity. Due to the abstract conception of the QEICC method it could also be adapted to RDF archiving and linear versioning systems. The `X-CurrentCommit` header field can be set to any string that distinctively identifies a state of the graph in the version log. The `X-CurrentBranch` header field can be set to a fixed value. This would allow to implement the reject strategy but also the merge strategy can be implemented for successful merging by performing the branch and merge operations in the cache. While this would not allow to postpone the reconciliation, it provides the possibility to detect conflicts and provides the flexibility of domain specific merge methods.

ACKNOWLEDGMENTS

This work was partly supported by a grant from the German Federal Ministry of Education and Research (BMBF) for the LEDS Project under the grant number 03WKCG11C, by the Federal Ministry of Transport and Digital Infrastructure (BMVI) for the LIMBO project under the grant number 19F2029G, and the Federal Ministry for Economic Affairs and Energy (BMWi) for the Platona-M project under the grant number 01MT19005A.

REFERENCES

[1] Natanael Arndt and Michael Martin. 2017. Decentralized Evolution and Consolidation of RDF Graphs. In *17th International Conference on Web Engineering (ICWE 2017)*. Rome, Italy. https://doi.org/10.1007/978-3-319-60131-1_2

[2] Natanael Arndt, Patrick Naumann, Norman Radtke, Michael Martin, and Edgard Marx. 2018. Decentralized Collaborative Knowledge Management using Git. *Journal of Web Semantics* (2018). https://doi.org/10.1016/j.websem.2018.08.002

[3] Natanael Arndt and Norman Radtke. 2017. A Method for Distributed and Collaborative Curation of RDF Datasets Utilizing the Quit Stack. In *INFORMATIK 2017*. Chemnitz, Germany. https://doi.org/10.18420/in2017_187

[4] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. 1995. A Critique of ANSI SQL Isolation Levels. In *SIGMOD Conference*.

[5] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley.

[6] Jeremy Debattista, Javier D Fernández, Jürgen Umbrich, and Maria-Esther Vidal. 2018. Preface of MEPDaW 2018: Managing the Evolution and Preservation of the Data Web. In *MEPDaW 2018*.

[7] Jeremy Debattista, Javier D Fernández, Maria-Esther Vidal, and Jürgen Umbrich. 2019. Managing the evolution and preservation of the data web. *Journal of Web Semantics* (2019). https://doi.org/10.1016/j.websem.2018.12.011 editorial.

[8] Alan J. Demers, Karin Petersen, Mike Spreitzer, Douglas B. Terry, Marvin Theimer, and Brent B. Welch. 1994. The Bayou Architecture: Support for Data Sharing Among Mobile Users. *First Workshop on Mobile Computing Systems and Applications* (1994).

[9] Lee Feigenbaum, Gregory Todd Williams, Kendall Grant Clark, and Elias Torres. 2013. *SPARQL 1.1 Protocol*. Recommendation. W3C. https://www.w3.org/TR/2013/REC-sparql11-protocol-20130321/

[10] Paula Gearon, Alexandre Passant, and Axel Polleres. 2013. *SPARQL 1.1 Update*. Recommendation. W3C. https://www.w3.org/TR/2013/REC-sparql11-update-20130321/

[11] Cesare Pautasso Guy Pardon. 2011. Towards Distributed AtomicTransactions over RESTful Services. (2011). https://doi.org/10.1007/978-1-4419-8303-9_23

[12] Steve Harris and Andy Seaborne. 2013. *SPARQL 1.1 Query Language*. Recommendation. W3C. https://www.w3.org/TR/2013/REC-sparql11-query-20130321/

[13] Benjamin Heitmann, Richard Cyganiak, Conor Hayes, and Stefan Decker. 2012. An Empirically Grounded Conceptual Architecture for Applications on the Web of Data. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* (2012).

[14] Benjamin Heitmann, Richard Cyganiak, Conor Hayes, and Stefan Decker. 2014. Architecture of Linked Data Applications. In *Linked Data Management*.

[15] Pat Helland. 2007. Life beyond Distributed Transactions: an Apostate's Opinion.. In *CIDR*.

[16] Michael Martin and Sören Auer. 2010. Categorisation of Semantic Web Applications. In *proceedings of the 4th International Conference on Advances in Semantic Processing (SEMAPRO2010)*. Florence, Italy. http://svn.aksw.org/papers/2010/SEMAPRO_Categorisation_SWA/public.pdf

[17] Andrae Muys. 2008. A Concurrency Control Protocol for Multiversion RDF Datastores (Discussion Paper).

[18] Thomas Neumann and Gerhard Weikum. 2010. x-RDF-3X: Fast Querying, High Update Rates, and Consistency for RDF Databases. *Poceedings of the VLDB* 3 (Sept. 2010). https://doi.org/10.14778/1920841.1920877

[19] John Ousterhout and Eric Stratmann. 2010. Managing State for Ajax-Driven Web Components. In *Proceedings of the 2010 USENIX Conference on Web Application Development (WebApps'10)*. Boston, MA, USA.

[20] Shirish Hemant Phatak and Badri Nath. 2004. Transaction-Centric Reconciliation in Disconnected Client-Server Databases. *Mobile Networks and Applications* (2004). https://doi.org/10.1023/B:MONE.0000034700.03069.48

[21] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. 1990. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Trans. Comput.* 39 (1990). Issue 4. https://doi.org/10.1109/12.54838

[22] The W3C SPARQL Working Group. 2013. *SPARQL 1.1 Overview*. Recommendation. W3C. https://www.w3.org/TR/2013/REC-sparql11-overview-20130321/