MySQL Collaboration by Approving and Tracking Updates with Dependencies: A Versioning Approach

Dharavath Ramesh[1](✉) and Munesh Chandra Trivedi[2]

[1] Department of Computer Science and Engineering, Indian Institute of Technology (Indian School of Mines), Dhanbad 826004, India
drramesh@iitism.ac.in
[2] National Institute of Technology, Agartala, Tripura 799046, India
drmunesh.cse@nita.ac.in

Abstract. In recent times, data science has seen a rapid increase in the need for individuals and teams to analyze and manipulate data at scale for various scientific and commercial purposes. Groups often collaboratively analyze datasets, thereby leading to a proliferation of dataset versions at each stage of iterative exploration and analysis. Thus, an efficient collaborative system compatible with handling various versions is needed rather than the current most often used ad-hoc versioning mechanism. In a collaborative database, all the collaborators working together on a project need to interact together to perform extensive curation activities. In a typical scenario, when an update is made by one of the collaborators, it should become visible to the whole team for possible comments and modifications, which in turn aid the data custodian in making a better decision. Relational databases provide efficient data management and querying. However, it lacks various features to support efficient collaboration. In these databases, the approval and authorization of updates are based completely on the identity of the user, e.g., via SQL GRANT and REVOKE commands. In this paper, we present a framework well suited for collaboration and implemented on top of relational databases that will enable the team to manage as well as query the dataset versions efficiently.

Keywords: Collaborative databases · Versioning · Data cell · Database management

1 Introduction

Collaborative databases provide an efficient environment for team members working on a common dataset to work in a structured manner. It provides a mechanism allowing the collaboration members working on the same data to provide their inputs by interacting with each other. Each member is aware of all the updates made in the data by other collaborators and the current work carried out on the data. All the collaborators can discuss and comment on the data before it is finally approved and updated in the primary database after being thoroughly analyzed by an expert. With the advancement of data

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2022

https://doi.org/10.1007/978-3-031-10548-7_28

science, there is an increase in the need for a collaborative database that will allow the collaborators to work in a collaborative environment, such as scientific databases, to develop and curate data from experimental and analytical processes [1, 2]. A collaborative bioinformatics database provides a typical scenario [5]. Bioinformatics databases require storing, retrieving, and analyzing large amounts of biological information. It requires different types of specialists to produce biological datasets (e.g., gene annotation database), share them among collaborators, share the updates among themselves for commenting, and keep updating the data until they converge and agree on the final content [3]. The data should be visible to all collaborators for discussion and resolution.

Moreover, all the updates need to be evaluated and approved by an expert, typically a data curator, data custodian, or the principal project investigators (PIs), to avoid any ambiguity or conflict in the derived update. Once the PI accepts the update, it is reflected in the primary database for further experimentation and analysis. Current database technologies, such as relational databases, provide an efficient mechanism to store, update, and read data [4, 6]. Data in a relational database is organized into tables in the form of rows and columns. A relational database such as MySQL provides various functionalities to carry out operations like create, read, update, and delete. MySQL includes functions that maintain the security, integrity, accuracy, and consistency of the data [10, 11, 33, 34]. It allows for easy updates and maintenance of the data. However, it lacks any support for Collaboration as any update needs to be approved first by the administrator before it is reflected in the database and is visible to the other collaborators. In this paper, we provide a framework to allow Collaboration in relational databases, namely MySQL.

Data Scientists need to work with databases daily. Data analysts and engineers need to be proficient in SQL and Database Management. Using a Relational Database Management System such as MySQL as a collaboration framework helps them access, communicate, and work on data more efficiently and robustly. The relational Database Management System is the core for storing and updating data for most projects. Thus using RDBMS as a collaboration framework makes the framework faster and more efficient than other collaboration frameworks such as git. This highlights the importance of this paper for building a collaborative framework using RDBMS.

Currently, MySQL does not provide any support for Collaboration. If a user updates a data item, it is not reflected in the database until and unless it is accepted by an update-issuer (PI). Only after being accepted the data is reflected in the database and is visible to the other collaborators. This limits the use of MySQL from being used in collaborative environments as PI becomes the bottleneck in the update process.

To better understand the need for our approach, we first illustrate how conventional update approval fails to support an efficient collaborative environment. Later we present our approach to overcoming these shortcomings. Assume that users A and B collaborate with PI in some tasks (as shown in Fig. 1). Suppose User A updates a data item X at time T1 and changes its value from x to y. At time T2, the PI is notified about the update made by User A on the data item X. Now, the updated value of X is visible to both A and PI. Still, B can see only the old value of X. Thus, PI becomes the bottleneck as every update needs to be approved by him before it is reflected in the database. Moreover, B cannot comment or discuss the update made by User A on the data item X before it is approved by the PI and committed into the database.

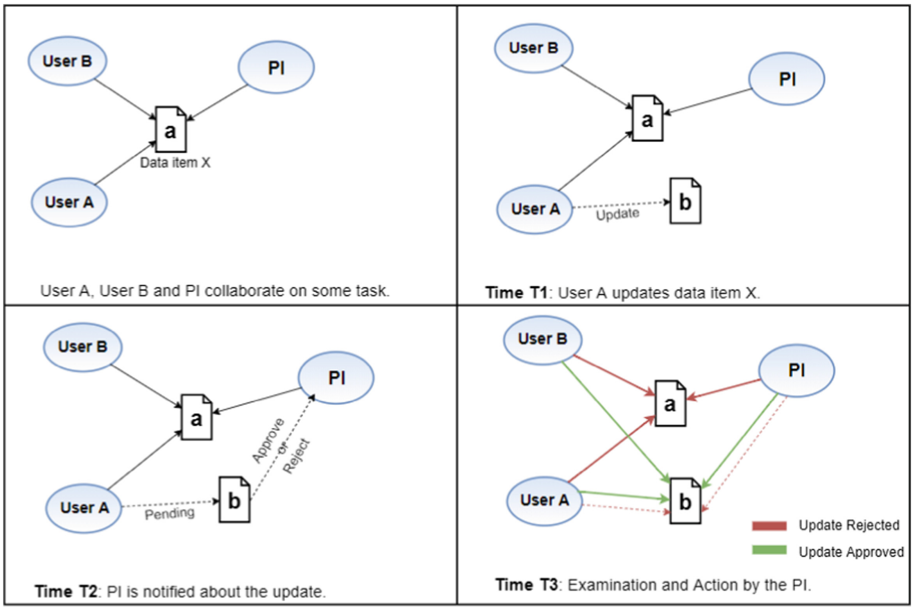

User B
PI
a
Data item X
User A
User A, User B and PI collaborate on some task.
User B
PI
a
User A
Update
b
Time T1: User A updates data item X.
User B
PI
a
Approve or Reject
User A
Pending
b
Time T2: PI is notified about the update.
User B
PI
a
User A
b
Update Rejected
Update Approved
Time T3: Examination and Action by the PI.

Fig. 1. An example of conventional update approval

This causes hindrance in the collaboration of work between Users A, B, and PI. Another drawback is that if B's work depends on the value of X, then knowing the updated value of X ahead of the time before it is committed will allow B to make necessary changes that need to be done in case the PI approves the update of X by A. Moreover, having prior knowledge of the new value of X will allow B to carry out his work with this new updated value and provide his feedback to the PI on the outcome of his experiment in case the update of X gets approved. From the figure, if the PI approves the new value of X at time T3, then B will be able to view the updated value of X after a time delay of T1 + T2 + T3. However, if at T3 PI rejects the update, then B will continue viewing the old value of X and will not learn from this experience. Also, as B is unaware of the change made on X by A, he may submit a similar change to X, which will be a waste of time both for B and PI.

To overcome the problem faced by the conventional update approval method in a collaborative environment, we propose a framework that will work on top of a relational database, namely MySQL, to handle the needs of a collaborative environment efficiently. As shown in Fig. 2, at a time, T1 collaborator A updates the value of data item X. Instead of directly committing the updated value in the database, it is marked pending approval. It is waiting for the PI to approve or reject the update. At the time, T2's PI is notified about the update. In the meantime, until the approval or rejection of the update takes place, any collaborator, say B, can see and comment on the update. Now the updated value is visible to B from time T2. Thus, PI is no longer a bottleneck as now all the collaborators can view and comment on the update even before it is committed in the database. Also, now PI can view the feedback of other collaborators on the update and accordingly decide whether to accept or reject the update. At time T3, the PI examines the update and approves or rejects it.

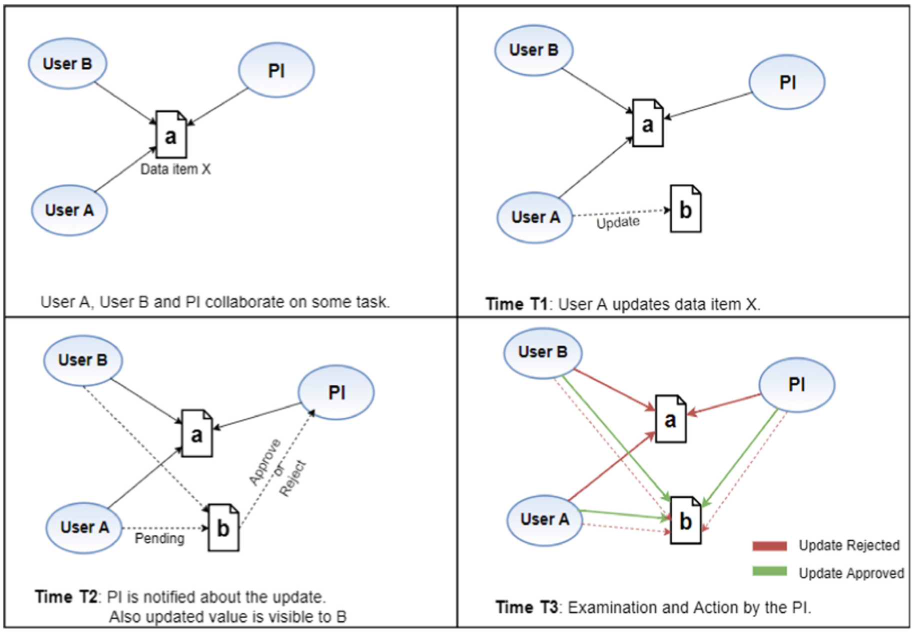

User B
PI
a
Data item X
User A
User A, User B and PI collaborate on some task.
User B
PI
a
User A
Update
b
Time T1: User A updates data item X.
User B
a
PI
Approve or Reject
User A
Pending
b
Time T2: PI is notified about the update. Also updated value is visible to B
User B
PI
a
User A
b
Update Rejected
Update Approved
Time T3: Examination and Action by the PI.

Fig. 2. Depiction of how things work in our collaboration framework

If the update gets accepted, then the new value of X is visible to B, and the status of the version of X with value b is changed to being "approved" (as shown in Fig. 2). Otherwise, if PI rejects the update, then the update status is marked as "rejected." After the value of the update is approved, it is finally reflected in the actual table and is removed from the pending table. We will discuss the role of the actual table and the pending table later in the paper. It is now compared with the conventional update model in Fig. 1. The delay for collaborator B to view the update is improved from T1 + T2 + T3 to 0. Such a model will provide efficiency in Collaboration as each user will give his feedback on the work being done by others before it is accepted and committed to the database.

1.1 Our Conributions

In this paper, we present an efficient system that realizes collaboration inside a relational database, namely MySQL. While MySQL allows easy updates and querying of data, it does not support different features needed in a collaborative scenario, namely update approval/rejection, commenting, versioning, history of updates, and dependency between different versions of data derived from each other. Hence, we extend MySQL with the following functionalities in the following manner.

- Perform Insert, Update, and Delete operations on the data item.
- Maintain the history of all the updates for a given data item or cell.
- Mark each update as either Approved, Rejected, or Pending Approval.
- Extend SQL to allow querying (i) history of all updates, (ii) Approved data only, and (iii) Most recent values only.
- Maintain different versions of the data item and multiple instances of the same version data.
- Establish dependency between different versions of data items and describe dependency rules.
- Provide metrics for the performance of users: (i) User with a maximum number of approvals, (ii) Users with a maximum number of rejections, and (iii) User with most numbers of updates.

The rest of the paper is organized in the following manner. Section 2 discusses the related work. Section 3 presents an overview of the presented collaboration framework. Section 4 discusses the design of this framework. Section 5 draws the life cycle of a suggested change. Section 6 presents the result obtained from the experiment. Section 7 draws the conclusion and lists down future work.

2 Related Literature

There have been few attempts to design databases frameworks to support a collaborative environment by including features like versioning. One such framework, which goes by the name AUDIT, has been proposed in the paper 'AUDIT: approving and tracking updates with dependencies in collaborative databases', and a query interface for this framework has been explained in the paper 'COACT: a query interface language for collaborative databases [14, 15]. The AUDIT framework has been implemented on top of Apache HBase [30–32]. HBase already supports version and history tracking of updates. However, nothing like this is facilitated by the commonly used relational databases like SQL. A drawback in the AUDIT framework is that it can consider only one instance of data at a particular version, i.e., if there is more than one instance of the data item at a particular version in the Pending table and one of them gets approved then all other instances will be automatically rejected. The framework presented in this paper considers all the instances of the data item at a particular version.

Orpheus database is another attempt at facilitating versioning capabilities on huge datasets [17, 18]. It is implemented on top of standard relational databases, and thus it supports most of the features supported by relational databases [16]. The OrpheusDB makes use of three models to represent different versions of the dataset:

- The combined table model supports versioning by including an attribute that stores a set of versions for each row.
- The second model uses a separate table that stores the versioning information by mapping versions to row IDs.
- The third model is used to store the mapping row IDs to versions.

OrpheusDB executes the operation ‘checkout’, which creates a new version of the table using the older versions. The other important operation is ‘commit,’ creating a new checkout version. However, the operations in the Orpheus database are performed on the complete dataset. In contrast, the framework suggested in this paper operates on a single data cell (a particular column of a particular row). Most of the frameworks proposed to date facilitate versioning only at the dataset level, contrary to the framework we are proposing, which supports versioning at the data cell level [7–9, 19], Apart from these two models, there are other related works like temporal databases, collaborative databases, checkout and check-in systems, and active databases [26–29]. Temporal databases [12, 13], or multi-version databases [20–22] can track the histories of data with each update. However, these databases present zero support for mandatory features like pending approval or rejecting approvals by the PI. Thus these databases are not exactly used in collaborative environments.

An active database management system (ADBMS) [24, 24] is an event-driven system in which schema or data changes generate events monitored by active rules. Active database management systems are invoked by synchronous events generated by user or application programs and external asynchronous data change events such as changes in sensor value or time. Using various rules [23], active databases can handle pending approval and reject the pending approvals if they are invalid. However, such a system would be highly inefficient. However, there are a lot of shortcomings in using git. Git requires technical excellence and is much slower. It has poor GUI and usability.

3 System Overview

To explain how collaboration is carried out in a database, we need to understand various terminologies and processes that are carried out in a collaborative environment.

3.1 Update

The update refers to any changes made to the database by any collaborators. A collaborator updates the data item or cell in the actual table. This update can be of three types:

a) Insert - Insert a new row of data in the actual table.
b) Modify- Modify or change previous data already present in the cell of the actual table.
c) Delete - Delete an entire row from the actual table.

Any update made by the collaboration needs to be approved by the PI before it is finally reflected in the actual table. Each update creates a new version of the data on which the update was made. The version number of the updated data item is decided by the version number of the data item on which the update was made. We will elaborate on the versioning of data in the subsequent topic.

3.2 Approval/Reject of Updates

In a collaborative environment, every update made by a collaborator needs to be reviewed and examined before committing it to the actual database. This is done to maintain the authenticity and consistency of the database. It is the role of the Principal Investigator (PI) to ensure that the database does not contain any ambiguity by examining each update carefully. Once the PI has examined the update made by a collaborator, he decides whether to approve the update or reject it. If the PI approves the update, then the changes are reflected in the actual database else; if the update is rejected, no changes are made in the actual database. The record of this is maintained in history.

3.3 Pending Approval

When a collaborator makes an update in a data item, the update is set up as a request awaiting PI approval to be accepted and reflected in the original database. The update request is moved to a Pending table (explained later), where the PI can examine the updates made by the collaborators and take necessary actions (approval or rejection) on the pending updates. All such updates pending approval of the PI are termed as Pending Approval updated in this paper. On the other hand, all the updates in the pending table can further be updated by a collaborator resulting in a new update request, which will then be stored in the pending table awaiting approval of the PI.

3.4 Version

Each update of the data item results in the creation of a new version of that data item. Version no. of the updated data item is determined from the previous version of the data item from which the update has been derived. For example, if a collaborator updates a data item at version 1, then the newly updated data item will have version no. 2 (shown in Fig. 3a). In our collaboration framework, we are considering multiple instances of a version to exist simultaneously. This allowed introduction efficiency in the collaborative environment as multiple users could work on the same version of a data item. This will result in multiple instances having the same version no, all of which need to be considered. For example, assume a data item X having version 1 is updated by different collaborators. User A updates the data item X and creates version 2 of this data item (say version 2.1). Now, if another update is made by User B on data item X having version 1, then this will also result in the new version of data item X having version number 2 (say version 2.2). Thus, both instances of version 2 of data item X need to be considered to account for the work done by both the users on the same version of the data (Fig. 3b). Now to provide consistency among different instances of the same version, we assume the role of the Merger. We further explain the merging and role of a merger in the next topic.

3.5 Merging

Different users can create different instances of the same version derived from the previous data item. This can lead to inconsistencies and incompleteness in the version of data, as selecting a particular instance of the version of the data item will not show the entire changes made in that version. For example, assume two versions of X, version 2.1 and version 2.2, both derived from version 1 of X. Now version 2.1 may not contain the changes made in version 2.2 of the data item X and vice versa. Thus if a user views version 2.1 of X, he will not be getting complete knowledge about the changes made in the data-item X in version 2. This will result in inconsistency in the data among different versions. To overcome this issue, we introduce the concept of merging different instances of the same version to keep the data at a particular version consistent and complete. Thus all the different instances of a version, contributed by different collaborators, are combined, and only one instance of each version is maintained (shown in Fig. 4).

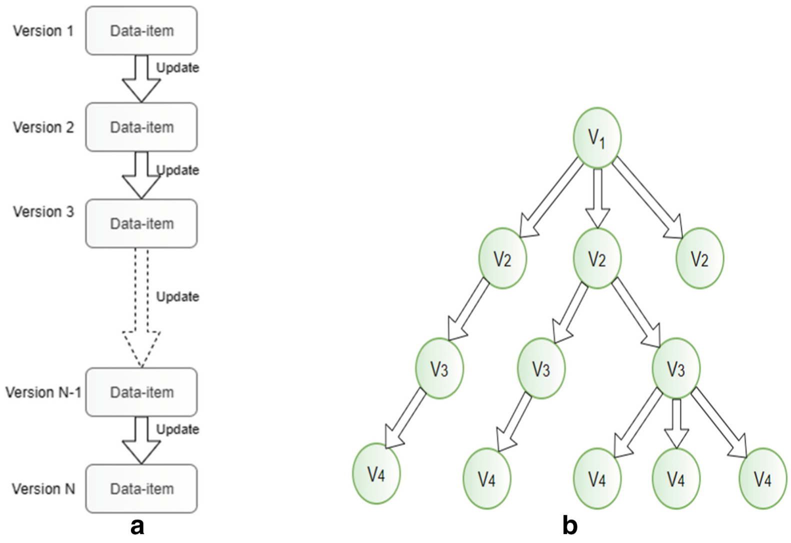


Fig. 3. **a** Numbering of different version of a data item, **b** Different instances of the same version of the data-item after update

3.6 Working of Collaborations in Databases

We have implemented our collaboration framework in a relational database. Data is stored in a relational database in the form of tables. Collaborators work on a common database that is accessible to all. In our collaboration framework, any changes made to a data item are not directly updated to the original table until and unless the PI approves it. We have assumed that the update is carried out in an Actual table that contains the data on which all the collaborators are working. We have implemented the framework for each cell of the Actual table. Thus, they allow the collaborators to change a particular column of a row from the table. Initially, the Actual table is empty, and when the data is inserted into the table, it is treated as the 1st version of the data. When a user updates a data item (or cell) in the Actual table, it is set as pending approval. This request is maintained in a separate table called the Pending table. So for each cell in the Actual

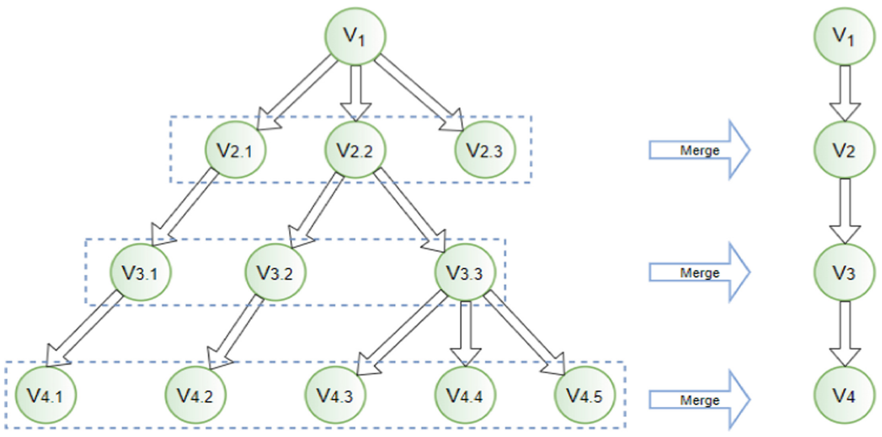


Fig. 4. Merging different instances of the same version into a single version

table for which an update is done, a row is created in the Pending table for that update, which contains the cell's data and other details. Updates pending in the Pending table can also be taken up for further work by a collaborator. We have maintained a Final table along with the Pending table. The Final table contains all the records regarding the latest update of a cell in the Actual table. Each row of both the Final and Pending table corresponds to the cell of the Actual table on which the update is carried out. Once the update in the Pending table is accepted/rejected by the PI, it is moved to the History table. The History table maintains the record of all the updates carried out in a data item over time.

4 System Design

This section describes the various tables used in designing the required framework. It discusses the schemas of the tables and their utility in making the framework sturdy and definitive. Apart from the Actual table (which stores the Actual data in its cells), we have used the following six tables:

Final Table: This table stores as a tuple, particularly the latest data approved by the PI corresponding to one of the Actual table cells. We identify a cell of the Actual table using the tuple <row number, column name>. Each tuple in this table has values corresponding to the following columns:

- Row no: It keeps the cell's row number in the Actual table to which this tuple belongs.
- Column name: It stores the name of the column of the cell in the Actual table to which this tuple belongs.
- Approved timestamp: It has the timestamp at which the PI approved this data.
- Data: This column has the md5 hash of the latest approved data.
- Latest contributor: It saves the id of the contributor who made the most recent change.
- Version no: It has the current version number of the data.

History Table: This table keeps the details of the changes which were either approved or rejected for all the cells in the Actual table. Each tuple in this table has values corresponding to the following columns:

- HS no (History Serial Number): This is an auto-increment column for identifying a row uniquely in this table.
- Row no: It keeps the cell's row number in the Actual table to which this tuple belongs.
- Column name: It stores the name of the column of the cell in the Actual table to which this tuple belongs.
- Data: This column has the md5 hash of the data.
- Version no: The current version number of the data.
- Operation: The operation is intended by the contributor to the cell's data, and it can take values from {'Insert,' 'Update,' 'Delete'}.
- Operation timestamp: The timestamp at which the change was implemented depends on whether it was approved or rejected by the PI.
- Status: The status of the change suggested, i.e., whether it was approved or rejected.
- Version seq no: The current sequence number of the data in a particular version (Assuming that there can be more than one data having the same version).
- Contributor:The Id of the contributor who suggested this change.

Pending Table: This table stores the changes submitted by the collaborators as a tuple, and these changes are yet to be approved or rejected by the PI. Each tuple in this table has values corresponding to the following columns:

- PS no (Pending Serial number): This is an auto-increment column for identifying a row uniquely in this table.
- Row no: It keeps the cell's row number in the Actual table to which this tuple belongs.
- Column name: It stores the name of the column of the cell in the Actual table to which this tuple belongs.
- Data: This column has the md5 hash of the data.
- Version no: It has the current version number of the data.
- Operation: The contributor intends the operation to the cell's data, and it can take values from {'Insert,' 'Update,' 'Delete'}.
- Submission timestamp: The timestamp at which the change was submitted.
- Version seq no: The current sequence number of the data in a particular version (Assuming that there can be more than one data having the same version).
- Contributor: The Id of the contributor who suggested this change.

Merge Table: Since we allow more than one data to have the same version, efficient merging of all the data having the same version number to get a single data is needed, and this Merge table just facilitates that. Each tuple in this table has values corresponding to the following columns:

- MS no (Merge Serial number): This is an auto-increment column for identifying a row uniquely in this table.
- Row no: It keeps the cell's row number in the Actual table to which this tuple belongs.

- Column name: It stores the name of the column of the cell in the Actual table to which this tuple belongs.
- Version no: It has the current version number of the data.
- Sequence count: It stores the number of data having the same version number, which has been merged for each cell corresponding to tuple <row number, column name>.
- Merged timestamp: It has the latest timestamp at which a new data having that particular version number was merged with already existing data at that version no.

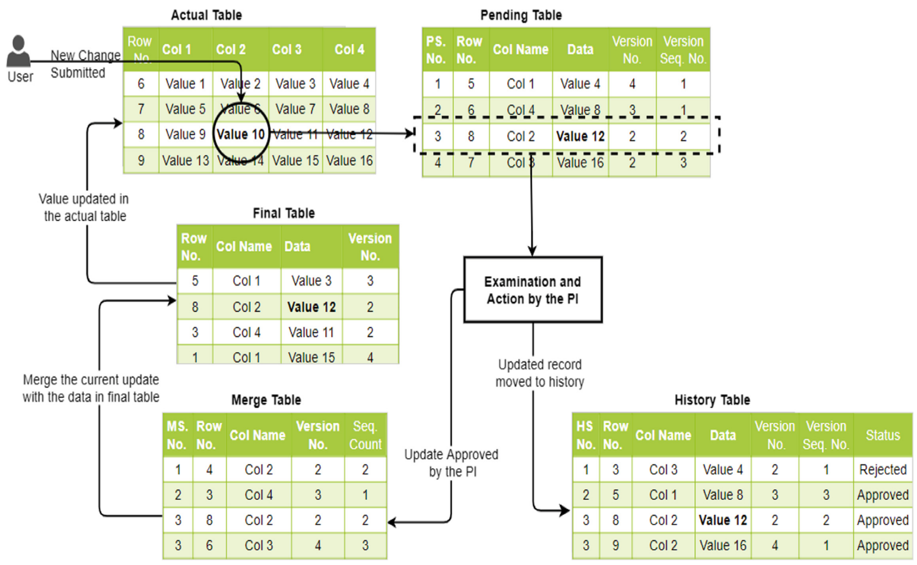
Comment Table: This table keeps the details of all the comments made by collaborators on data waiting for approval in the Pending table. Each tuple in this table has values corresponding to the following columns:

- CS no (Comment Serial number): This is an auto-increment column for identifying a row uniquely in this table.
- PS no (Pending Serial number): It keeps the serial number of that row in the Pending table on which the comment was made.
- Comment: It has the comment made by one of the collaborators.
- Commenter: It stores the Id of the collaborator who commented.
- Timestamp: It keeps the timestamp at which this comment was made.

5 Life Cycle of the Proposed Framework

We assume that the Actual table is that table that has the original data, and a cell in this table is identified by the tuple <Row number, Column name> where Row number is an auto-increment column in the Actual table, as shown in Fig. 5.

Since this framework supports updates at a cellular level, so a collaborator might want to change the data in any of the cells and submit the changed data. The changed data of a cell, along with other details, is stored as a tuple (row) in the Pending table waiting for verification by the PI. In Fig. 5, the cell corresponding to the tuple <7, Col 2> has data equal to 'value 10'. A collaborator wants to update the cell data to 'value 12', hence corresponding to that cell, a row is inserted in the Pending table. The 'Data' column of the newly inserted row has data equal to 'value 12'. The change can be either rejected or approved by the PI depending on the authenticity of the new data suggested by the contributor and the feedback from other collaborators on this new data. Either way, this changed data, along with other details, is moved to the History table with 'Approved' or 'Rejected' status. If the PI approves the new data, then the version count is updated in the Merge table, and the new data is merged with other data of the same version so that a single unified data exists for a given version. As shown in Fig. 5, the change is approved by the PI, and hence 'value 12' and other columns are moved to the History table with status = 'Approved.' The data is updated in the Final table, which has < row no, column name > corresponding to the cell whose data is being changed and so corresponding to the tuple <7, col2 > in the Final table, we now have 'Data' equal to 'value 12'.


User
New Change Submitted
Value updated in the actual table
Merge the current update with the data in final table
Examination and Action by the PI
Updated record moved to history
Update Approved by the PI

Actual Table

Row No.	Col 1	Col 2	Col 3	Col 4
6	Value 1	Value 2	Value 3	Value 4
7	Value 5	Value 6	Value 7	Value 8
8	Value 9	**Value 10**	Value 11	Value 12
9	Value 13	Value 14	Value 15	Value 16

Pending Table

PS. No.	Row No.	Col Name	Data	Version No.	Version Seq. No.
1	5	Col 1	Value 4	4	1
2	6	Col 4	Value 8	3	1
3	8	Col 2	**Value 12**	2	2
4	7	Col 3	Value 16	2	3

Final Table

Row No.	Col Name	Data	Version No.
5	Col 1	Value 3	3
8	Col 2	**Value 12**	2
3	Col 4	Value 11	2
1	Col 1	Value 15	4

Merge Table

MS. No.	Row No.	Col Name	Version No.	Seq. Count
1	4	Col 2	2	2
2	3	Col 4	3	1
3	8	Col 2	2	2
3	6	Col 3	4	3

History Table

HS No.	Row No.	Col Name	Data	Version No.	Version Seq. No.	Status
1	3	Col 3	Value 4	2	1	Rejected
2	5	Col 1	Value 8	3	3	Approved
3	8	Col 2	**Value 12**	2	2	Approved
3	9	Col 2	Value 16	4	1	Approved

Fig. 5. The life cycle of a suggested change

6 Results and Discussions

In this section, we evaluate the performance of different framework Algorithms in the following manner. We evaluate the time taken by different functionalities under insert, update, and delete for the different numbers of records in the table. The corresponding output of the experiment is shown in Fig. 6, Fig. 7, and Fig. 8, which shows the total query delay for each operation.

Insert Operation: Issuing an INSERT algorithm by a collaborator involves inserting new rows in the Pending table having data corresponding to the row's cells, which are to be inserted. Issuing the APPROVE algorithm for the Insert operation will move the corresponding rows from the Pending table to the History table and insert the row in the Actual table. Invoking the REJECT algorithm instead of APPROVE will simply move the rows in the History table without affecting the Actual table. Figure 6 shows that the INSERT has the least delay, which is expected because it inserts data to the Pending table only. In contrast, REJECT inserts data to the History table and removes data from the Pending table.

Update Operation: Issuing an UPDATE algorithm by a collaborator involves inserting a new row in the Pending table having data corresponding to the cell of the Actual table, which is to be updated. Issuing the APPROVE algorithm for the Update operation will move the corresponding rows from the Pending table to the History table and updation of the row in the Actual table. Invoking the REJECTION algorithm instead of APPROVE will simply move the rows from Pending to the History table without affecting the Actual table. Figure 7 shows that the UPDATE operation on the Actual table results in the least delay. This is expected because it inserts a single row in the Pending table compared to

REJECT or APPROVE. This results in the removal of data from the Pending table and the insertion of a row corresponding to the update in the History table. Approve also results in the updation of the record in the Actual table and thus has the highest delay as visible in Fig. 7.

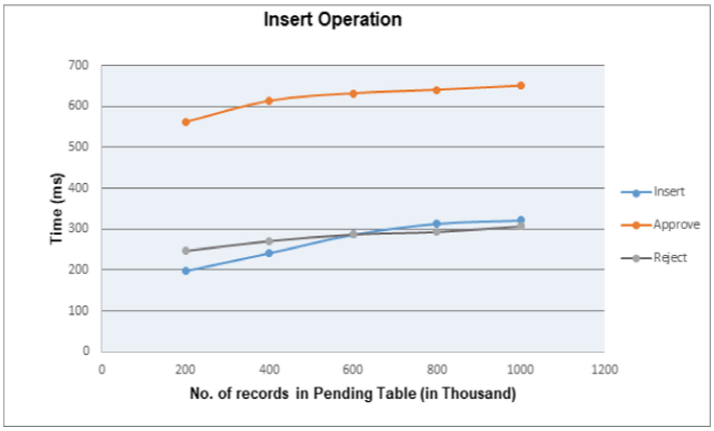


Fig. 6. Evaluation of INSERT, APPROVE and REJECT for insert operation

Delete Operation: Issuing a DELETE algorithm by a collaborator involves inserting a new row in the Pending table having data corresponding to the row of the Actual table, which is to be deleted. Issuing the APPROVE algorithm for the Delete operation will move the corresponding rows from the Pending table to the History table and updation the row in the Actual table. Invoking the REJECTION algorithm instead of APPROVE will simply move the rows from Pending to the History table without affecting the Actual table. Figure 8 shows that the Delete operation has similar behavior to the Update delay. Delete operation involves inserting a single row in the Pending table corresponding to the row to be deleted in the Actual table. APPROVE and REJECT both result in adding data in the History table and removing data from the Pending table with APPROVE, resulting in the deletion of the row from the Actual table. This is the same as the case discussed in the Update operation, resulting in the same trend in the graph. However, the delay of the Delete operation is less than that of the Update operation. This is because an update involves writing operations on the database, which increases the delay as no such operation occurs in the Delete operation.

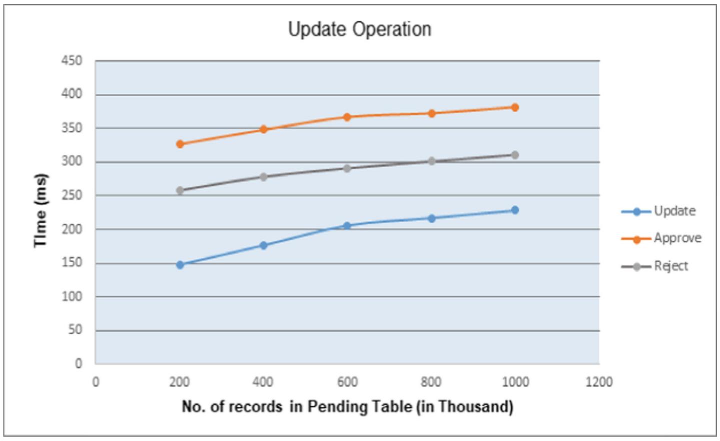


Fig. 7. Evaluation of UPDATE, APPROVE and REJECT for update operation

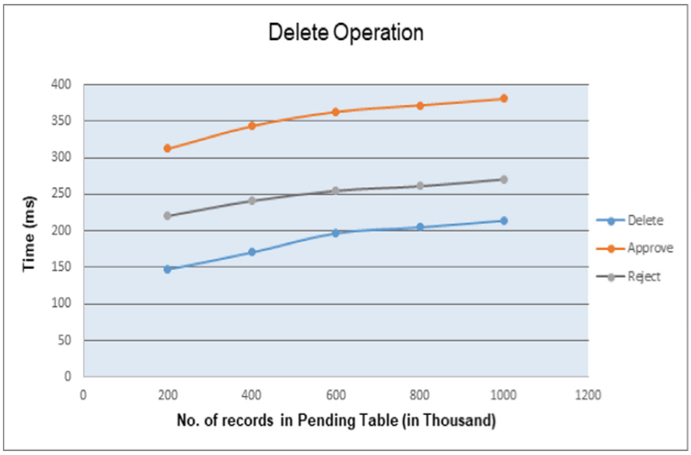


Fig. 8. Evaluation of DELETE, APPROVE and REJECT for delete operation

7 Conclusion and Future Scope

The framework suggested in this manuscript is implemented on top of standard relational databases like SQL, and unlike most other frameworks, it supports versioning at the data cell level. One more significant advantage of this framework is that it considers all the instances created by different collaborators for a data item at a particular version instead of just considering one of the instances and rejecting all others. The framework even includes certain contribution metrics that enable the PI to get an idea of the performance of different collaborators. A well maintainable interface can be easily concluded for the framework. We have described the algorithms and the prototypes for all the functionalities, making this framework easier to implement in a real-time collaboration scenario.

Since multiple instances of the data item at a particular version are considered, and the data item at the next version can originate from any of those instances at the previous version, it becomes difficult to trace the origination of data at a particular version. We intend to work on this aspect of the framework in the future. Another part of the framework which deserves a bit more work is the merging feature, as the merging feature needs to be efficient and accurate for the overall efficiency of the framework.

Acknowledgments. This research work is supported by the Indian Institute of Technology (Indian School of Mines), Dhanbad, Govt. of India. The authors wish to express their gratitude and heartiest thanks to the Department of Computer Science & Engineering, Indian Institute of Technology (ISM), Dhanbad, India, for their research support.

References

1. Mershad, K., Malluhi, Q.M., Ouzzani, M., Tang, M., Gribskov, M., Aref, W.G.: AUDIT: approving and tracking updates with dependencies in collaborative databases. Distrib. Parallel Databases **36**(1), 81–119 (2017). https://doi.org/10.1007/s10619-017-7208-y
2. Mershad, K., et al.: COACT: a query interface language for collaborative databases. Distrib. Parallel Databases **36**(1), 121–151 (2017). https://doi.org/10.1007/s10619-017-7213-1
3. Huang, S., Xu, L., Liu, J., Elmore, A.J., Parameswaran, A.: Orpheus DB: bolt-on versioning for relational databases. Proc. VLDB Endow. **10**(10), 1130–1141 (2017)
4. Xu, L., Huang, S., Hui, S., Elmore, A.J., Parameswaran, A.: ORPHEUSDB: a lightweight approach to relational dataset versioning. In: Proceedings of the 2017 ACM International Conference on Management of Data, pp. 1655–1658. ACM, May 2017
5. Navathe, S.B., Patil, U., Guan, W.: Genomic and proteomic databases: foundations, current status and future applications. J. Comput. Sci. Eng. **1**(1), 1–30 (2007)
6. Goldberg, D., Nichols, D., Oki, B.M., Terry, D.: Using collaborative filtering to weave an information tapestry. Commun. ACM **35**(12), 61–70 (1992)
7. Howe, B., Halperin, D., Ribalet, F., Chitnis, S., Armbrust, E.V.: Collaborative science workflows in SQL. Comput. Sci. Eng. **15**(3), 22–31 (2013)
8. Halperin, D., Ribalet, F., Weitz, K., Saito, M.A., Howe, B., Armbrust, E.: Real-time collaborative analysis with (almost) pure SQL: a case study in biogeochemical oceanography. In: Proceedings of the 25th International Conference on Scientific and Statistical Database Management, p. 28. ACM, July 2013
9. Eirinaki, M., Abraham, S., Polyzotis, N., Shaikh, N.: QueRIE: collaborative database exploration. IEEE Trans. Knowl. Data Eng. **26**(7), 1778–1790 (2014)
10. Harrington, J.L.: Relational Database Design and Implementation. Morgan Kaufmann (2016)
11. Coronel, C., Morris, S.: Database Systems: Design, Implementation, & Management. Cengage Learning (2016)
12. Nascimento, M.A., Sellis, T., Cheng, R.: Special issue on spatial and temporal database management. GeoInformatica **19**(2), 297–298 (2015). https://doi.org/10.1007/s10707-015-0224-z
13. Radhakrishna, V., Kumar, P.V., Janaki, V.: An efficient approach to find similar temporal association patterns performing only single database scan. Revista Tecnica De La Facultad De Ingenieria Universidad Del Zulia **39**(1), 241–255 (2016)
14. Collaborate your way to better SQL queries and data visualizations: https://blog.modeanalytics.com/collaborate-your-way-to-better-sql-queries/

15. SqlDBM's Latest And Greatest: Team Project Collaboration: http://blog.sqldbm.com/team-collaboration/
16. Painless Data Versioning for Collaborative Data Science: https://medium.com/data-people/painless-data-versioning-for-collaborative-data-science-90cf3a2e279d
17. Bioinformatics Databases: https://www.ebi.ac.uk/training/online/course/bioinformatics-terrified-2018/what-bioinformatics
18. MySQL Database - very good thesis: http://www.engpaper.com/mysql-database-very-good-thesis.html
19. Relational Database Management System: https://searchdatamanagement.techtarget.com/definition/RDBMS-relational-database-management-system
20. Nambiar, U.B., Deshpande, P.M., Halasipuram, R.S., Iyer, B.R.: U.S. Patent No. 9,262,491. U.S. Patent and Trademark Office, Washington, DC (2016)
21. Alromema, N.A., Rahim, M.S.M., Albidewi, I.: Temporal database models validation and verification using mapping methodology. VFAST Trans. Softw. Eng. **11**(2), 15–26 (2016)
22. Alromema, N., Rahim, M.S.M., Albidewi, I.: An Efficient approach for modeling temporal database with interval-based timestamping in conventional database management systems. J. Comput. Theor. Nanosci. **14**(9), 4569–4575 (2017)
23. Kalanat, N., Kangavari, M.R.: Data mining methods for rule designing and rule triggering in active database systems. Int. J. Datab. Theory Appl. **8**(1), 39–44 (2015)
24. Berndtsson, M., Mellin, J.: Active database knowledge model. In: Liu, L., Özsu, M.T. (eds.) Encyclopedia of Database Systems. Springer, Boston (2009). https://doi.org/10.1007/978-0-387-39940-9_508
25. Veldhuizen, T.L.: U.S. Patent No. 9,424,304. Patent and Trademark Office, Washington, DC (2016)
26. Meagher, M.: U.S. Patent No. 8,822,848. U.S. Patent and Trademark Office, Washington, DC (2014)
27. Vance, J. R., et al.: U.S. Patent No. 9,277,833. Patent and Trademark Office. Washington, DC (2016)
28. Collins Jr., D.A., Amada, J.: U.S. Patent No. 8,925,811. U.S. Patent and Trademark Office. Washington, DC (2015)
29. Active Databases: http://web.cs.ucla.edu/classes/winter04/cs240A/notes/node1.html
30. Apache HBase Reference Guide: https://hbase.apache.org/book.html
31. The Architecture of Apache HBase: https://intellipaat.com/blog/what-is-apache-hbase/
32. Gómez, A., Benelallam, A., Tisi, M.: Decentralized model persistence for distributed computing. In: 3rd BigMDE Workshop, July 2015
33. Ramesh, D., Kumar, C.: An incremental protocol approach for secure collaboration between Byzantine processes in heterogeneous distributed processing systems. Glob. J. Technol. **3** (2013)
34. Ramesh, D., Khosla, E., Bhukya, S.N.: Inclusion of e-commerce workflow with NoSQL DBMS: MongoDB document store. In: 2016 IEEE International Conference on Computational Intelligence and Computing Research (ICCIC), pp. 1–5. IEEE, December 2016