Approving Updates in Collaborative Databases

Khaleel Mershad [†1], Qutaibah M. Malluhi [†2], Mourad Ouzzani [‡3], Mingjie Tang [⋆4], Walid G. Aref [⋆5]

[†] *Qatar University,* [‡] *Qatar Computing Research Institute,* [⋆] *Purdue University*

[1,2] {kwm03, qmalluhi}@qu.edu.qa, [3] mouzzani@qf.org.qa, [4,5] {tang49, aref}@cs.purdue.edu

***Abstract*—Data curation activities in collaborative databases mandate that collaborators interact until they converge and agree on the content of their data. Typically, updates by a member of the collaboration are made visible to all collaborators for comments but at the same time are pending the approval or rejection of the data custodian, e.g., the principal scientist or investigator (PI). In current database technologies, approval and authorization of updates is based solely on the identity of the user, e.g., via the SQL GRANT and REVOKE commands. However, in collaborative environments, the updated data is open for collaborators for discussion and further editing and is finally approved or rejected by the PI based on the content of the data and not on the identity of the updater. In this paper, we introduce a cloud-based collaborative database system that promotes and enables collaboration and data curation scenarios. We realize content-based update approval and history tracking of updates inside HBase, a distributed and scalable open-source cluster-based database. The design and implementation as well as a detailed performance study of several approaches for update approval are presented and contrasted in the paper.**

I. Introduction

In collaborative environments, e.g., scientific databases, large volumes of data are shared among scientists that collaborate to curate data coming from experiments and analytical processes. A typical scenario is when scientists produce some dataset, share it among collaborators that are not necessarily co-located, collectively update it, share the updates among themselves for commenting, and keep updating the data until they converge and agree on the final content. Once the dataset stabilizes, they eventually publish it. For example, this scenario takes place when deciding on the function of a particular gene or allele, or a protein binding site. While being discussed and commented on, the data along with all of its updates should be visible to all collaborators. Ultimately, the data custodian, e.g., the principal investigator (PI, for short), makes the final decision of approving some updates and rejecting others.

Current database technologies fall short in supporting the above collaborative environment. SQL supports GRANT/REVOKE access models that allow users to have data access rights based solely on the identity of the user [1], [2]. In this case, when an update takes place, it is not reflected into the database until the update-issuer commits the update, e.g., towards the end of the update transaction. Once committed, the update is visible to the collaborators. However, if this update needs to be approved by the PI, the updated value cannot be committed until approved. Hence the updated value cannot be shared with the other collaborators for commenting.

Assume that Users U1 and U2 collaborate with the PI in some task (See Fig. 1). At Time I_1, User U1 updates Object A and changes its value from 1 to 2. At Time (I_2), the PI is notified of the update. Both U1 and PI can see the update but U2 can only see the old value. The PI becomes a bottleneck as the PI is part of every update transaction, resulting in needless delays. Moreover, U2 does not have a chance to comment on or discuss the update before the new value of A is committed, which hampers the collaboration. Another drawback of the conventional approach is that if U2's experiments depends on the value of A, knowing the updated value of A ahead of time will allow U2 to setup and prepare for the experiment that needs to be re-performed in case A's update gets approved. Alternatively, if U2 is aware of A's new value, U2 may redo her experiments even before the approval takes place and provide feedback to the PI on the potential outcome of her experiment in case A gets approved. From the figure, at Time (I_3), the PI examines the update and decides to approve or reject the update on A. At Time (I_4), if the PI approves the update, then U2 can now see the new value of A after a prolonged time delay ($I_1 + I_2 + I_3$). However, if PI rejects the update, then U2 will continue to see the old value of A, and does not learn from this experience. Also, U2's seeing and commenting on the update beforehand may have affected the PI's decision on A's update, but U2 is unaware of this update attempt.

The above scenario demonstrates that collaborative environments require sophisticated support for approving updates. When a collaborator, say U1, updates a data item, it is marked as *pending approval* until the PI approves or rejects the update. In the meantime, any other collaborator, say U2, should be allowed to view the data pending its approval. Refer to Fig. 2 for illustration. At Time I_1, User U1 updates Object A from 1 to 2. At Time (I_2), the PI is notified of the update. At the same time, U2 can see A's new value. However, in order to distinguish between the old and new values of A, the new value for A, i.e., Value 2, is marked as "pending" approval. As a result, U2 is aware of the possible modification from time I_2. Now, the PI is not the bottleneck anymore as the pending updates are accessible by all collaborators for inspection and commenting. The PI can view the feedback from all other collaborators, e.g., from U2, before committing to a decision . At Time (I_3), PI examines the update and decides to approve or reject it. Next, at Time (I_4), if PI approves the update, then U2 can now see the new value of A, and the status of A with value 2 is changed to being "approved" as Fig. 2 illustrates. On the other hand, if PI rejects the update of User U1, the Value 2 of data object A is marked as "rejected". Compared to the standard update model in Fig. 1, the time delay for User U2 to view the update is improved from $I_1 + I_2 + I_3$ to 0. Data modification actions, i.e., update, approve, or reject, are recorded so that each collaborator can track the status of an object and the history of an update with each value in the history, whether approved or rejected. In the rest of this paper, we term this proposed update scheme as the *Update-Pending-Approval* model (or UPA, for short).

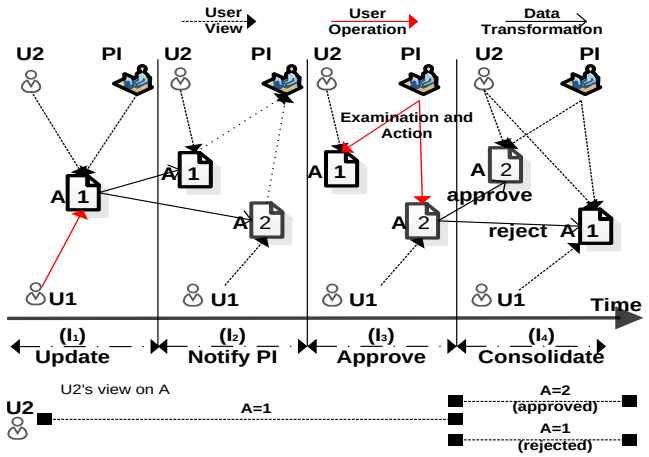


Fig. 1: An example of using conventional update processing.

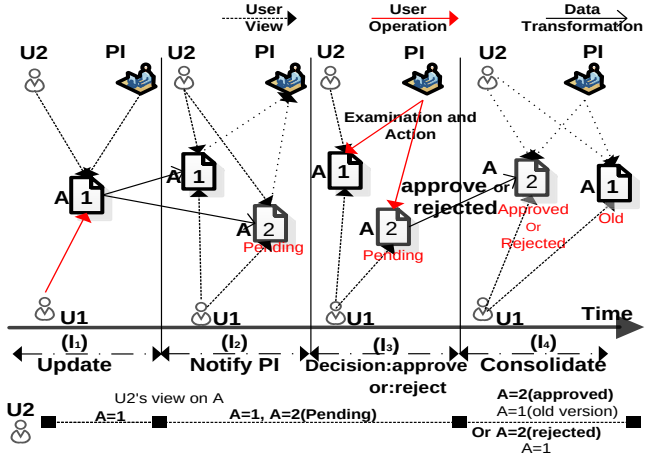


Fig. 2: The Proposed Update Pending Approval (UPA) Model.

UPA is very important for a variety of application scenarios. For example, assume that members of a scientific team are collaborating to collect vast amounts of data from a field experiment. Other scientists, who might be at distant locations, are conducting their own experiments and producing and saving results based on this collected data. These produced results, being saved to the database along with their associated experimental setups, should be checked for validity by one or more PIs before they can be approved and made public. Another example is content management systems that allow users to collaboratively edit shared content. UPA can be used in a content management system for accurate and efficient monitoring, approval, and history archival purposes. A third example is Wikis: While web users may continuously update their own pages in a Wiki, these updated pages can be publicly available while marked as "pending approval" until these updates are approved by the Wiki administrator(s).

In this paper, we present a prototype system that realizes UPA inside a cloud-based platform, namely HBase [3]. HBase is a distributed and scalable cluster-based database that is suitable to store big data on the cloud [3]. While HBase supports version and history tracking of updates, it does not support the notion of update approval or rejection that are essential features in data collaboration environments. We extend HBase with the following functionalities:

- We maintain the history of all the updates for a given data item or cell along with the associated meta-data.
- We mark each update as *Approved*, *Rejected*, or *Pending*, and we extend HBase to allow for querying (1) the history of all updates, (2) the approved data only, or (3) the most recent values only (approved or pending).
- We introduce three modes of operation for a data cell depending on most queried values, i.e., (1) last inserted values, (2) last approved values, or (3) both the last inserted and the last approved values. In addition, our system can dynamically switch between modes to adapts to the current query workload.

The rest of this paper proceeds as follows. Section II discusses the related work. Section III presents an overview of HBase and the procedures for supporting UPA. Section IV introduces the various design alternatives, the data organization, history tracking, and querying. We presents our experiments in Section V and conclude in Section VI.

II. Related Work

Several areas are related to our proposal ranging from active databases, multiversion databases, checkout and check-in systems, to data provenance management.

Active databases [4], [5] aim to respond to events either inside or outside the database. They can realize a database hold to make the database system aware of external activities. Conceptually, by using rules and triggers, active databases can be used to monitor data that is pending approval and mark it as potentially invalid. However, this would be rather inefficient and would not scale because a new rule would need to be added explicitly for each update that is pending approval.

Multi-version or temporal database systems [6], [7] keep track of the history of updated data. They are very efficient as they maintain the history of the updates at the disk-page level. However, it is hard to extend them to support the semantics for pending-approval updates as well as the rejection of the unapproved updates. This is also true for versioning support in cloud database systems.

Checkout and check-in systems, e.g., SVN [8] and GIT [9], maintain current and historical versions of files. The main drawback of adopting a similar approach is that the data will be residing outside the database system. Also, updating at the attribute level cannot be supported by such systems. In contrast, UPA allows the data to reside inside the database system and the history will be maintained at the right granularity. Hence, the proposed system will result in performance efficiency when querying *Approved*, *Rejected*, or *Pending* data.

Data provenance management [10] or provenance support inside scientific workflow systems [11] retain the derivation history of a data item from its original sources. The provenance of data can help a scientist understand the development and progression of a data item over time. However, a provenance-based system does not maintain the consistency of data items and hence rolling back and rejecting cascaded updates is not supported. In addition, UPA supports transaction consistency in the sense that updates that depend on a rejected update are tracked and will also be rejected. In addition, UPA is

implemented inside a cloud database, i.e., at the engine level rather than at the application level. Thus, it can co-exist with a provenance system to achieve stronger data tracking.

In building the proposed system, we leverage our previous work in [12], [13]. In contrast to the original centralized architecture, in this paper, we investigate the implications as well as the design and the performance issues related to providing the UPA model in a cloud-based environment. Unique challenges include the impact of partitioning the data within the cluster and how they are partitioned and the effort of each partitioning policy on the performance.

III. System Overview

A. An Overview of HBase

HBase is a column-oriented data store running on top of HDFS [14], [15]. Data is stored as fragments of columns, instead of complete rows in the form of <key, value> pairs. Fig. 3 illustrates the architectural diagram of HBase. HBase contains two main entities: (1) an HMaster, and (2) a set of Region Servers. The HMaster is responsible for administrative operations such as creating and dropping tables. The Region Server manages the Regions that are the actual data stores [3]. HBase operates alongside Zookeeper [16] that manages configuration and naming tasks. When a client wants to execute an HBase operation, he/she obtains from Zookeeper the Region Server address that hosts the data. Next, the client executes the required operation by accessing the corresponding region.

HBase contains four basic operations: *Put*, *Delete*, *Get*, and *Scan*. *Put* inserts the value of a data item into an HBase table. Each data item or cell is identified by four fields that constitute its key: *Row*, *Column Family*, *Qualifier*, and *Timestamp*, where *Row* is the main identifier. *Column Family* groups related *qualifiers* together, and *Timestamp* is usually the time at which the value is inserted. *Put* can either insert values into a newly created *qualifier*, or update the value of an existing *qualifier*. In the latter case, the previous value is not removed, but is saved as an older version. *Delete* marks data for deletion in the memory store. When data is flushed into disk, the actual data is deleted. *Get* retrieves data that is within a single row while *Scan* retrieves data from the whole table.

HBase *coprocessors* are the basic building blocks for any custom operation in HBase. A coprocessor pushes computational logic into the HBase cluster, close to where data is stored. HBase coprocessors are of two types: *Observers* and *Endpoints*. An *Observer* executes certain code before or after executing an HBase operation. An *Endpoint* runs a custom function on a specified set of Region Servers and can be invoked at any time by the user. We use *Endpoints* for implementing various UPA operations.

B. Extending HBase to Support UPA

History Tracking: History tracking of Data refers to the process of monitoring data over time to track down (i) how the data changed over time, (ii) who changed it, at what times, and possibly (iii) the reasons, if any, behind each of the changes. Tracking the history of data is a very important operation in collaborative environments as well as in data auditing in database systems. The validity of future results depends on the correctness of the existing data and how it was derived.

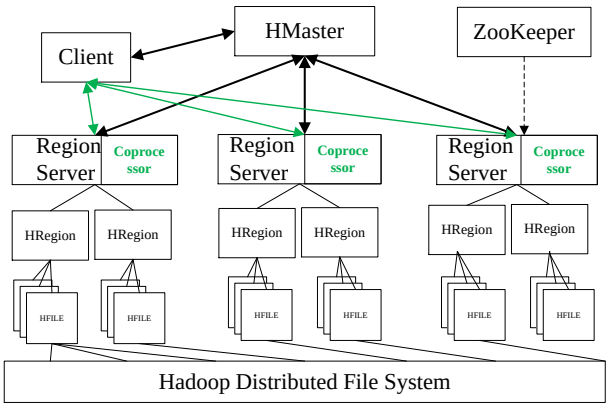


Fig. 3: HBase System Structure and EndPoint Coprocessors. Our extensions are labeled in green.

Existing database management systems do not provide means for history tracking. For instance, HBase saves the new value of a cell as a new version of the cell. The user can query each value of the cell and the time at which it was inserted. Other important data, e.g., the ID of the user who inserted the value and the parameters of the environment in which it was generated are not saved. Hence, the process of history tracking still misses important features and needs to be augmented to satisfy the requirements of researchers and collaborators.

UPA enables history tracking by saving, with each operation on a cell, the necessary metadata related to the operation, such as the user ID, the type and reason for the operation, the status of the data value (more about this field in the next section), and any other metadata related to the environment in which the data was generated, such as the machine ID. The proposed history tracking mechanism supports important operations, such as reverting a cell to a previous stable state and building a timeline of the lifecycle of the cell.

Approval of Updates: A set of privileged users, e.g., principal investigators, will monitor and approve/reject the data values being inserted or updated by collaborators. We refer to each of these privileged users as a principle investigator or PI. We differentiate between three types of inserted data values: (i) *pending*, a data value that is yet to be approved or rejected by a PI, (ii) *approved*, a data value that is approved by a PI and (iii) *rejected*, a data value that is rejected by a PI. A PI can approve or reject data items either individually or as groups through bulk approve/reject operations. In the latter case, the approve/reject interface will provide the PI with a set of data items that are pending. The PI selects the data items that should be approved, and the system adds them to an approve list. After the PI finishes, the remaining data items are automatically added to a reject list. The system performs a single operation which approves the items in the approve list and rejects those in the reject list.

Storing the history of data updates over time along with the metadata would adversely affect negatively the overall performance. Users that do not query the historical data should not be penalized by the fact that the system is maintaining the history. In order to address this issue, we save the history of a data cell in a separate storage entity. In the proposed HBase implementation, we create a History table in which the complete history and the metadata of a cell are saved. For the

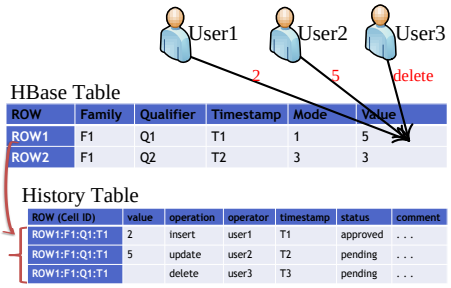


Fig. 4: Formats of records in the original and History tables.

original data table in which the cell is created, we investigate the following three alternative design options: (i) Store the most recently *inserted* value of the cell. In this case, all other older values, whether approved or not, are stored in the history table, (ii) Store the most recently *approved* value of the cell. At the same time, all the more recent updates to the cell, which are still pending, are stored in the history table until they get approved, and (iii) Store both the most recently *inserted* and the most recently *approved* values of the cell in the original table and all other versions of the cell are stored in the history table. Using this alternative enables users that query the most recent value of a cell (or the most recent *approved* value) to obtain the result directly from the original table in an efficient way without having to access the history table. Users interested in History Tracking will have to query the data that is saved in the History table that resides in a separate storage entity. We refer to these three alternatives as the UPA Modes.

IV. System Design

A. History Tracking

In this section, we present the proposed format of entries in the History table and how they comply with HBase standards.

Record Structure: To illustrate the format of a history record, we present the example given in Fig. 4. Suppose that user1 inserts the cell: <Table:ROW1:F1:Q1> at Time T1. The value 2 will be inserted into the original table, and the first history record of the cell will be inserted in the History table. The status of the history record will be "pending". Suppose that the PI approves the history record at Time T2. Then, the status of the history record is set to "approved". Suppose that at Time T3, user2 inserts the value 5 to the cell, then the cell value in the original table will be set to 5, and a new history record of the cell is added to the History table. Finally, suppose that at Time T3, user3 deletes the cell. In this case, the cell in the original table is marked deleted, and a new history record, with operation equal to "delete" and status equal to "pending" will be added to the History table. The last two history records will be pending approval or rejection by the PI.

Table Structure: As stated in Section III, different values that are inserted into the same cell are saved as different versions of the cell. The creator of an HBase table defines the maximum number of versions, say n, that can be saved per cell. If the number of versions of the cell exceeds n, only the newest n versions will be kept and the older versions will be deleted. In UPA, we set the default value of n equal to 1 for any table other than the History table. The reason is that in each such table, which we refer to in this paper as the original table, we save only either the last inserted value or the last approved value. When we operate in UPA Mode 3, we need to save both the last inserted and the last approved values of the cell. In such a case, we save these values as two different cells that will point to the original cell. Using this approach, we keep the original tables as compact as possible, which will lead to more query efficiency. However, the maximum number of versions in the History table will be set to a high value since each newly inserted value of the cell will lead to the creation of a new record in the History table, and we want to keep all history records of each cell, as long as the cell is not deleted.

B. Approving Updates

The Approval Process: A history record will include, in addition to the cell value and the timestamp, the operation, i.e., 'insert', 'update', 'delete', or 'approve', the user ID, the record status, i.e., 'pending', 'approved', or 'rejected', and comments. A PI can approve or reject pending values using one of two methods. In the first method, the PI specifies, in addition to the ID of the cell, a list of timestamps of records to be approved (or rejected). The ScHistory operation sequentially approves (or rejects) these records. In the second method, the PI specifies two time instances. The ScHistory operation will search the History table for all records of the cell whose timestamps fall within the two time instances and subsequently approve (or reject) these records. The first method is useful for approving (rejecting) a limited number of records. The PI simply queries the History table to obtain these records, and then passes their timestamps to ScHistory. The second method is useful for a large number of values generated for the same cell, and the generation start and end times are known.

The UPA Model: Two important values of each data cell need to be distinguished; the last inserted value and the last approved value. Since these two values are mostly important for users who query the cell, we propose three operating modes as follows: (i) If users are mostly querying the most recently inserted value of the cell, the cell will operate in UPA Mode 1, in which only the most recently inserted value of the cell is saved in the original table. Each newly inserted value will overwrite the existing value in the original table. (ii) If users are mostly querying the most recently approved value, then this value is saved in the original table. Each newly approved value will overwrite the previous approved value. Then the cell will be operating in UPA Mode 2. (iii) If users heavily query both the most recently inserted and the most recently approved values, then both values will be saved separately in the original table, and the cell will be operating in UPA Mode 3. Regardless of the UPA Mode of the cell, each time an operation is conducted on the cell, a history record that contains all necessary metadata is written to the History table.

The system will dynamically change the cell from one UPA Mode to another as follows: for each cell, the system maintains and updates three parameters: a mode variable, the cell's search-for-last-approved-value rate (S_{LA}) and search-for-last-inserted-value rate (S_{LI}). When S_{LI} is higher than a certain threshold while S_{LA} is low, the system changes the cell to operate in UPA Mode 1 by setting mode to 1. If S_{LA} is high and S_{LI} is low, mode will be set to 2. If both rates are high, mode will be set to 3. All UPA operations (update, delete, approve, ...) will read the mode variable before execution to

know the current Mode of the cell and perform the operation according to that. For example, when the ScGet operation searches for the last inserted value, it examines mode. If mode is equal to 1 or 2, it searches in the original table of the cell. If mode is 3, it searches in the History table.

V. Experimental Study

A. Dataset and Test Environment

We realize and test our system in an HBase Cluster with six virtual machines (VMs). A single VM is used as the Hadoop and HBase masters, while the other five VMs act as Hadoop slaves and HBase clients. We use data from Wikipedia [17], which provides the history of updates performed on Wikipedia pages. Each Webpage represents a data cell and each update on the Webpage is a new value of the cell. The size of the dataset is around 2.1 GB. We test three scenarios, where we insert into HBase 1, 10, and 100 % of the data respectively, leading to three data sizes: 21 MB, 210 MB, and 2.1 GB respectively. Note that this is the size of the original data from the training file. After adding the metadata in the History table, the database size reaches about 5 times the size of the original file. The three data sizes were tested for all three UPA Modes. We also tested for each data size a dynamic-Mode scenario in which the workload parameters were varied over time.

In the fixed-Mode scenarios, each of the five slaves runs two processes in parallel: the first process inserts one fifth of the data into the Wiki table, while the second process continuously executes two queries that respectively retrieve the last inserted and the last approved values of a randomly chosen cell. On the other hand, the master VM executes, while the five slaves are inserting data, a PI process, which looks for pending values in the History table and randomly approves or rejects them, according to a certain rejection percentage *RP*. We vary *RP* between 0.01 and 0.8.In the dynamic-Mode scenario, we divide the simulation time into three equal parts: In the first part, the five clients target 90% of their search queries to the last inserted values, and the other 10% to the last approved values. In this case, the best UPA Mode to use is Mode 1. In the second part, the five clients target 90% of their queries to the last approved values and the other 10% to the last inserted values. The best UPA Mode here is Mode 2. In the third part, the five clients target the last inserted and the last approved values with equal percentages. Hence, the best UPA Mode for the third part is Mode 3. A continuously running ScMode thread periodically calculates S_{LI} and S_{LA} for each cell and dynamically changes the UPA Mode of the cell to the best suitable one according to the values of the two rates.

B. Results and Discussion

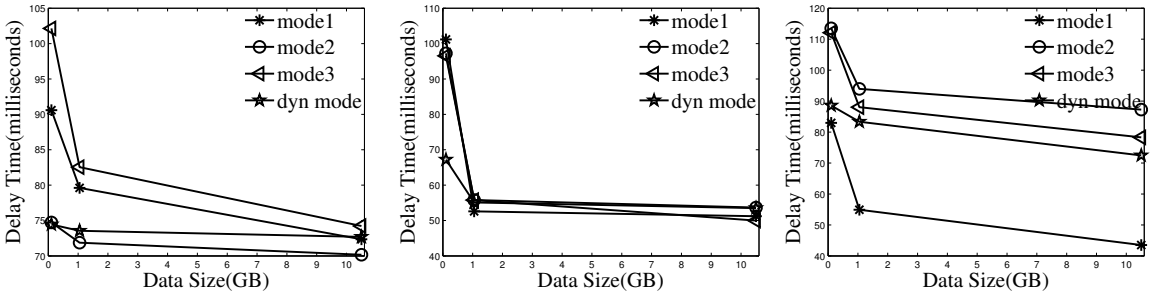
Fig. 5 and 6 show the total delay of the Insert, Delete, Approve, Reject, Search-last-inserted, and Search-last-approved operations. Fig. 5a shows that Mode 2 has the least Insert delay because it inserts data to the original table only when approved, while Mode 1 inserts data to the original table when data is inserted and when a history record is rejected. In contrast, Mode 3 inserts data to the original table at insertion, approval, or rejection and hence yields the highest delay. The Insert delay of dynamic-Mode is equal to that of Mode 2 when the data size is small and slightly less than that of Mode 1 when data size is large. For all four Modes, the delay converges as the data size increases. This behavior is also observed for the Delete, Approve, and Reject operations (Fig. 5b, 5c, 6a), which reflects the stability of the system.

Fig. 5b shows that delete has a similar end-to-end delay for the four Modes. The reason is that the delete algorithm is very similar among the three UPA Modes. In addition, we notice in Fig. 5c that Mode 1 has the minimum approve delay, because it approves the cell only in the History table, while Modes 2 and 3 need to write to the original table when approving. The approve delay of the dynamic-Mode is somehow in the middle between the delays of the three Modes, and is less than the delays of Modes 2 and 3 for all data sizes. As for reject, Mode 2 (Fig. 6a) has minimum delay, because it does not need to write to the original table in case of a reject. Modes 1 and 3 need to restore the previous approved value to the original table when the current value is rejected. Hence, they consume additional delay. Similar to approve, the reject delay of the dynamic-Mode scenario is less than the delays of Modes 1 and 3 for all data sizes. Also, the reject delay of the dynamic-Mode is less than that of Mode 2 when the data size is small.

For searching, Fig. 6b shows that Modes 1 and 3 have a much less delay than Mode 2 when searching for the last inserted value, since they save this value in the original table, while in Mode 2 it is retrieved from the History table. Fig. 6c shows that Modes 2 and 3 have much less delay than Mode 1 when searching for the last approved value, for the same reason. The figures also illustrate that the Search-last-inserted and Search-last-approved delays of dynamic-Mode are close to the best possible search delay and much less than the highest search delay for both operations. Note that the delays in Fig. 6b and 6c increase as the data size increases, since the number of regions increases, and the regions will be distributed on different VMs. In this case, there is a higher probability that the region that contains the target data is on another VM.

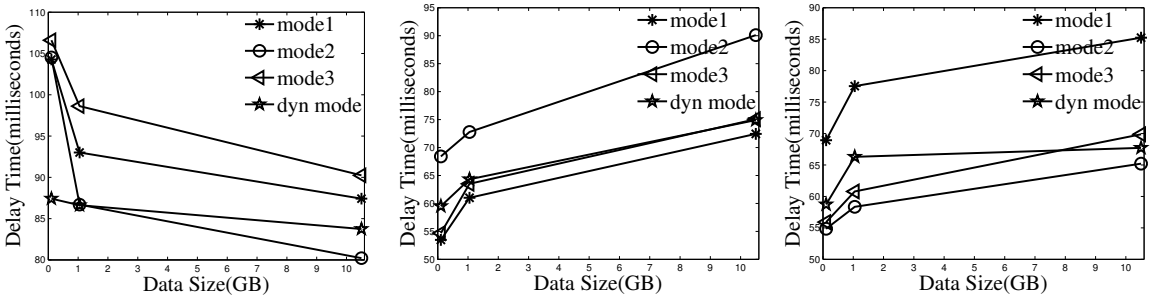
Finally, Fig. 7 gives the delay of the six operations when varying *RP* between 0.01 and 0.8. Notice that Delete, Approve, and Search are the most affected by increasing *RP*, e.g., the Delete delay decreases from 60 to 48 ms. The reason is that deletes that are found in the training file will not be executed if the data has already been rejected. Thus, the total delete delay decreases. The delays of Approve and Search decrease because as more rejections are made, more data is deleted from the original and History tables and the database size decreases. However, the Reject delay increases because more reject jobs are injected into the database, while the Insert delay is not affected, since the same insertions are made, regardless of the rejection percentage. In general, the performance of the system is acceptable for both high and low values of *RP*.

From the results, we notice that each of the three UPA Modes has good performance for some operations but weak performance for others. On the other hand, as Fig. 5 and 6 show, the dynamic-Mode scenario has a delay which is equal or slightly larger than the best delay, but much less than the worst delay, for all six operations. Hence, it is the best scenario that guarantees good performance for all operations.



(a) Insert (b) Delete (c) Approve

Fig. 5: The effect of data size on query delay: insert, delete, approve



(a) Reject (b) Search-last-inserted (c) Search-last-approved

Fig. 6: The effect of data size on query delay: reject, search-last-inserted, search-last-approved

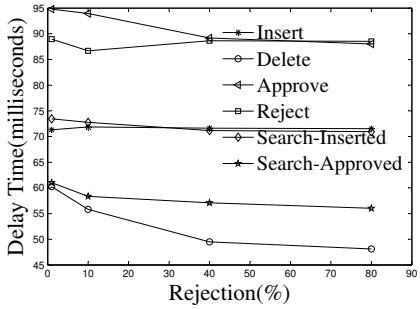


Fig. 7: Varying the rejection percentage for different queries

VI. Conclusion

In many applications, the correctness of inserted data is based not only on the identity of the user, but also on the value of the data itself. The underlying database system should handle the approval of data and enable users to view its status and track its history. In this paper, we presented an Update Pending Approval model for collaborative databases. We presented the basic operations of the system and discussed its various modes of operation. Most importantly, our system is adaptive, i.e., it can dynamically switch between modes based on changes in the workload. We tested the various operations using a custom-built HBase cluster. Our results illustrate the advantages and disadvantages of each of the proposed modes and the superiority of the adaptive mode over the fixed modes.

Acknowledgment

This publication was made possible by the support of an NPRP grant from the QNRF and the support of the National Science Foundation under Grants IIS-1117766 and IIS-0964639. The statements made herein are solely the responsibility of the authors.

References

[1] R. Fagin, "On an authorization mechanism," *ACM Trans. Database Syst.*, vol. 3, no. 3, pp. 310–319, Sep. 1978.

[2] P. P. Griffiths and B. W. Wade, "An authorization mechanism for a relational database system," *ACM TODS*, vol. 1, no. 3, pp. 242–255, Sep. 1976.

[3] "Apache hbase." [Online]. Available: https://hbase.apache.org/

[4] A. Aiken, J. Hellerstein, and J. Widom, "Behavior of database production rules: Termination, confluence, and observable determinism," in *SIGMOD*, 1992.

[5] N. W. Paton and O. Díaz, "Active database systems," *ACM Comput. Surv.*, vol. 31, no. 1, pp. 63–103, 1999.

[6] D. Lomet, R. Barga, M. Mokbel, and G. Shegalov, "Transaction time support inside a database engine," in *Data Engineering, 2006. ICDE '06. Proceedings of the 22nd International Conference on*, April 2006.

[7] "Oracle flashback." [Online]. Available: http://www.oracle.com/technetwork/issue-archive/2008/08-jul/o48totalrecall-092147.html/

[8] "Apache subversion." [Online]. Available: http://subversion.apache.org/

[9] "Git." [Online]. Available: http://git-scm.com/

[10] P. Buneman, A. Chapman, and J. Cheney, "Provenance management in curated databases," in *SIGMOD '06*. Chicago, IL, USA: ACM, 2006.

[11] Y. Amsterdamer, S. B. Davidson, D. Deutch, T. Milo, J. Stoyanovich, and V. Tannen, "Putting lipstick on pig: Enabling database-style workflow provenance," *VLDB Endow.*, vol. 5, no. 4, pp. 346–357, Dec. 2011.

[12] M. Y. Eltabakh, M. Ouzzani, and W. G. Aref, "bdbms - a database management system for biological data," *CIDR 2007*.

[13] M. Eltabakh, W. G. Aref, A. Elmagarmid, and M. Ouzzani, "Handson db: Managing data dependencies involving human actions," *IEEE TKDE*, no. PrePrints, p. 1, 2013.

[14] "Apache hadoop." [Online]. Available: http://hadoop.apache.org/

[15] "Hadoop distributed file system." [Online]. Available: http://hadoop.apache.org/docs/r2.3.0/hadoop-project-dist/hadoop-hdfs/HdfsUserGuide.html

[16] "Apache zookeeper." [Online]. Available: http://zookeeper.apache.org/

[17] "Wikipedia challenge." [Online]. Available: http://www.kaggle.com/c/wikichallenge/data