Distrib Parallel Databases (2018) 36:81–119
https://doi.org/10.1007/s10619-017-7208-y

AUDIT: approving and tracking updates with dependencies in collaborative databases

Khaleel Mershad[1] · Qutaibah M. Malluhi[2] · Mourad Ouzzani[3] · Mingjie Tang[4] · Michael Gribskov[4] · Walid G. Aref[4]

Published online: 21 September 2017

Abstract Collaborative databases such as genome databases, often involve extensive curation activities where collaborators need to interact to be able to converge and agree on the content of data. In a typical scenario, a member of the collaboration makes some updates and these become visible to all collaborators for possible comments and modifications. At the same time, these updates are usually pending the approval or rejection from the data custodian based on the related discussion and the content of the data. Unfortunately, the approval and authorization of updates in current databases is based solely on the identity of the user, e.g., via the SQL GRANT and REVOKE commands. In this paper, we present a scalable cloud-based collaborative database system to support collaboration and data curation scenarios. Our system is based on

✉ Khaleel Mershad
km004@live.aul.edu.lb

Qutaibah M. Malluhi
qmalluhi@qu.edu.qa

Mourad Ouzzani
mouzzani@qf.org.qa

Mingjie Tang
tang49@cs.purdue.edu

Michael Gribskov
gribskov@purdue.edu

Walid G. Aref
aref@cs.purdue.edu

1 Arts, Sciences and Technology University in Lebanon, Beirut, Lebanon

2 Qatar University, 2713, Al Hala St, Doha, Qatar

3 Qatar Computing Research Institute, HBKU, Doha, Qatar

4 Purdue University, West Lafayette, IN, USA

an Update Pending Approval model. In a nutshell, when a collaborator updates a given data item, it is marked as pending approval until the data custodian approves or rejects the update. Until then, any other collaborator can view and comment on the data, pending its approval. We fully realized our system inside HBase, a cloud-based platform. We also conducted extensive experiments showing that the system scales well under different workloads.

Keywords Collaborative databases · Cloud computing · Data dependency · Multiversion data · Update authorization · Big data

1 Introduction

In collaborative environments, e.g., in scientific databases, large volumes of data are shared among multiple collaborators that cooperate to develop and curate datasets coming from experiments and analytical processes. A collaborative gene annotation database (CGADB), which is created for almost every organism whose genome (DNA) sequence is determined, provides a typical scenario. When the genome of an organism is determined, the sequence of millions of short DNA fragments (sequence reads) are determined, and combined, by overlapping, into longer sequences called assemblies or contigs (contiguous sequences). Such sequence assemblies are the primary data from which much of the information in a CGADB is derived. Derived data, collectively referred to as annotations, include information such as the location of genes (beginning and ending positions, in bases, within the assembly), the locations of exons (subparts of genes), the protein sequence inferred from the gene sequence, and curated text descriptions describing the function of the gene (initially inferred from sequence comparison programs such as Blast [1]). Much of this derived data is initially determined by computational analysis of the assembly sequence. Eventually, information from further computations or from lab experiments may confirm or contradict the initial, computationally derived data requiring updates to the derived information. Because these temporal updates may be ambiguous, or even in conflict, they must be evaluated by an expert, typically a data curator, data custodian, or the project PIs (principal investigators). For simplicity, we will refer to the person responsible for making curatorial decisions as the PI.

Over time, as more sequence reads are determined, or as improved assembly software is deployed, the sequence of the assemblies are updated and improved. Software used in computational annotation may also be updated. In consequence, primary or derived values in the database may change, in principle, invalidating other derived data that depend on these values. In practice, it is difficult to manage the updates of such a database because of the dependency between the primary and derived data, and because of multiple conflicting updates proposed by members of the collaborative group. Ultimately, expert decisions must be made by the PI; however, for efficient collaboration, the conflicting proposed updates must be available to the collaborative group for discussion and resolution. In addition to the collaborative aspect of the database, CGADBs also present a public face in which only the information that has

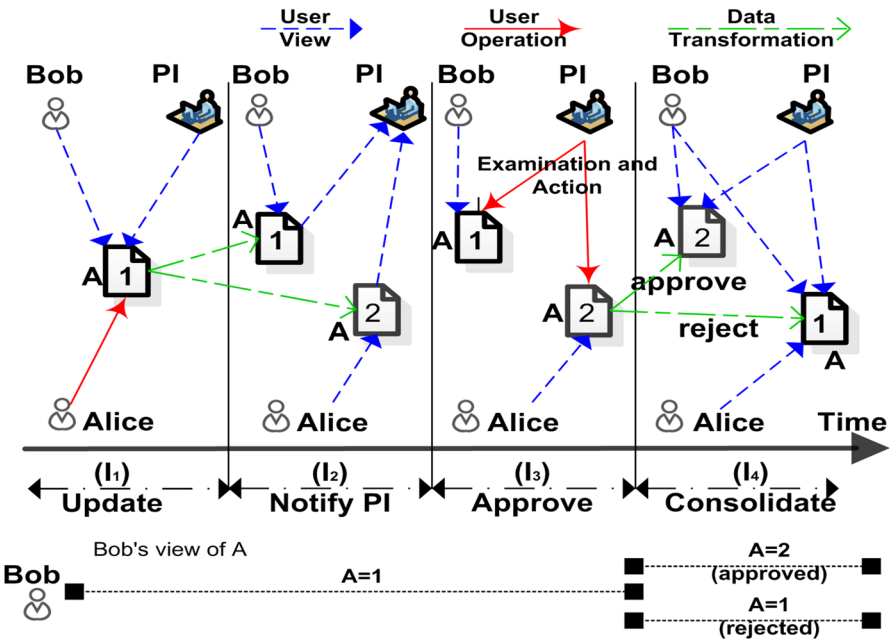


Fig. 1 An example of using conventional update processing

been curated and approved by the PI is visible. This is intended to avoid rapid flux of the database content, and to prevent confusion on the part of the users.

Current database technologies fall short in supporting the above scenarios. SQL supports GRANT/REVOKE access models that allow users to have data access rights based solely on the identity of the user [2,3]. 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 beforehand, the PI action becomes part of the update transaction, where the updated value cannot be committed until approved by the PI. Hence the updated value cannot be shared with the other collaborators for commenting. Note that we will be using the terms user and collaborator interchangeably throughout the paper.

To better motivate our approach, we first show how conventional update approval falls short in supporting collaborative scenarios such as in CGADB. We then present our proposal to remedy these shortcomings. Assume that Users Alice and Bob collaborate with the PI in some task (Fig. 1). At Time I_1, User Alice updates Object A and changes its value from 1 to 2. At Time (I_2), the PI is notified of the update. Both Alice and PI can see the update but Bob can only see the old value. The PI becomes a bottleneck as she is part of every update transaction, resulting in needless delays. Moreover, Bob does not have a chance to comment or discuss the update before the new value of A is committed, which hampers the collaboration. Another drawback is that if Bob's experiments depend on the value of A, knowing the updated value of A ahead of time will allow Bob to setup and prepare for the experiment that needs to be

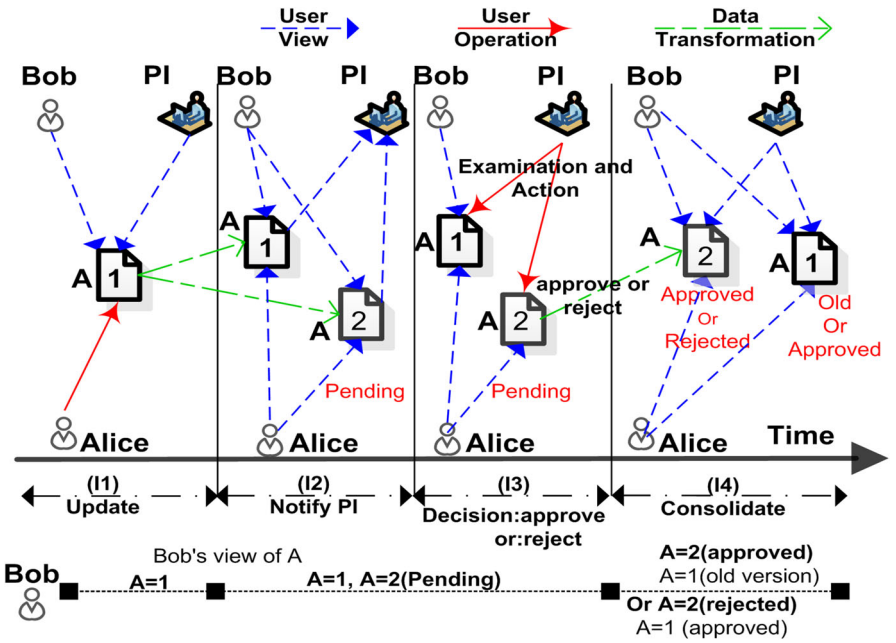


Fig. 2 The proposed Update Pending Approval (UPA) model

re-performed in case A's update gets approved. Alternatively, if Bob is aware of A's new value, she 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 it. At Time (I_4), if the PI approves the update, then Bob 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 Bob will continue to see the old value of A, and does not learn from this experience. This process is also wasteful as Bob may submit a similar change to A, unaware that Alice has already made the same update and was rejected. Also, Bob's seeing and commenting on the update beforehand may have affected the PI's decision on A's update, but Bob is unaware of this update attempt.

To remedy to the aforementioned shortcomings, we propose an Update Pending Approval (UPA) model (Fig. 2). When a collaborator, Alice, updates a given data item, it is marked as pending approval until the PI approves or rejects the update. Until the approval or rejection of the update takes place, any other collaborator, say Bob, can view and comment on the data, pending its approval. More explicitly, at Time I_1, User Alice updates Object A from 1 to 2. At Time (I_2), the PI is notified of the update. At the same time, Bob 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, Bob is aware of the possible modification from time I_2. The PI is no longer a bottleneck 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 Bob, before approving or rejecting the update. At

Time (I_3), PI examines the update and approves or rejects it. Next, at Time (I_4), if PI approves the update, then Bob can now see the new value of A, and the status of the version of A with Value 2 is changed to being "approved" as Fig. 2 illustrates. On the other hand, if PI rejects the update of Alice, the Value 2 of Data Object A is marked as "rejected". Compared to the standard update model in Fig. 1, the time delay for Bob 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).

In addition to the CGADB scenario described above, our proposed model is also applicable for other applications. 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 results, being saved to the database along with their associated experimental metadata, must be checked for validity by the PI before they can be approved and made public. Another example is a content management system that allows users to collaboratively edit shared content. UPA can be used for accurate and efficient monitoring, approval, and history archival. Wikis provide a third example, where UPA is applicable. While web users may continuously update their own pages in a Wiki, these updated pages may be publicly available while marked as pending approval until these updates are approved by the Wiki administrator(s). All of these scenarios, where data is published and is marked as pending approval so that collaborators can discuss and comment on the disseminated data as early as possible, even before the data is verified and approved or rejected by the PI, are potential use cases for UPA.

In this paper, we present an enhanced system that realizes UPA inside a cloud-based platform, namely HBase [4]. HBase is a distributed and scalable cluster-based database that is suitable for storing big data on the cloud [4]. While HBase supports version and history tracking of updates, it does not support the different features needed in the UPA model, namely update approval/rejection, version metadata, and dependency between different versions that are derived from each other. Hence, we extend HBase with the following functionalities:

- Maintain the history of all the updates for a given data item or cell along with the associated meta-data.
- Mark each update as either Approved, Rejected, or Pending Approval.
- Extend HBase to allow querying (1) the history of all updates, (2) the approved data only, or (3) the most recent values only (approved or pending).
- Introduce three modes of operation for a data cell depending on which values are mostly queried, i.e., (1) last inserted values, (2) last approved values, or (3) both the last inserted and the last approved values. For each mode, we cache frequently queried values where for efficient retrieval. In addition, we dynamically switch between modes to adapt to the current query workload.
- Introduce dependency between different versions of a data item and describe the dependency rules for different operations of the system.

This paper extends an earlier version of this work [5] along several directions. In [5], we presented a general overview of how to extend HBase for handling the history and allowing PIs to approve or reject certain versions of a data item. This paper include the following major enhancements: (i) a detailed description of the system operations, (ii) a new dependency model including scenarios in which a newly inserted version depends on an existing version, how the dependency model can be expressed using a dependency tree, and the necessary changes to the History table to enable version dependency. (iii) a detailed treatment of the dependency rules, and (iv) experimental evaluation performed on the entire system with dependency handling and on a much larger dataset. We call our proposed system Approving and Tracking Updates with Dependencies in Collaborative Databases (AUDIT).

The rest of the paper proceeds as follows. We discuss related work in Sect. 2. Section 3 introduces HBase and the UPA functionality. Section 4 describes the various design alternatives, the data organization and procedures for supporting AUDIT, history tracking, and querying. Section 5 presents dependencies between different versions and the rules to support them. Section 6 describes the new HBase operations of AUDIT. Section 7 presents the experiments. Section 8 illustrates the system usability. Finally, Sect. 9 concludes the paper.

2 Related work

Work related to our proposed AUDIT includes long-running transactions, active databases, multiversion databases, checkout and check-in systems, data provenance management, and collaborative databases.

A long-running database transaction [6,7] is one that usually interferes with external users' activities e.g., waiting for a manager's authorization to update a record. Such a transaction might have long periods of inactivity, often due to waiting for external messages to arrive or external actions to take place. It guarantees consistency and durability, but not atomicity or isolation. Upon abortion, a compensation action is triggered for rolling back the activity of the transaction. However, a long-running transaction system does not provide query and notification mechanisms for data inconsistency. Hence, long-running transaction systems cannot support AUDIT.

Active databases [8,9] can respond to events that take place either inside or outside the database system. 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 inefficient and may not scale because a new rule would need to be added explicitly for each update in the database that is pending approval.

Multi-version or temporal database systems [10–12] keep track of the histories of updated data. They are efficient as they maintain the history of the updates at the disk-page level. However, it is hard to extend these systems 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, e.g., as in [4], as we illustrate further in Sect. 3.

Checkout and check-in systems, e.g., [13,14], maintain current and historical versions of the data. The main drawback of adopting a similar approach is that the

data must reside outside the database system. In addition, updating at the attribute or cell level is a finer granularity than what SVN systems offer. Hence, more extensive filtering and further processing are needed to extract and isolate the updated data, whether it is pending approval or is already approved, or for retrieving the entire update history. Extracting the data versions from a Checkout-Check-in-like system, thus, introduce long delays and unacceptable response times. In contrast, the proposed AUDIT mechanism allows the data to reside inside the database system, where the history is maintained at a useful granularity. Hence, our proposed system results in efficient performance when querying Approved, Rejected, or Pending Approval data, and long response time delays can be avoided.

Recently, infrastructures that are designed specifically for scientific data applications are becoming more popular. These include CKAN (ckan.org), Domo (domo.org), Enterprise Data Hub (cloudera.com/enterprise), Domino (dominoup.com), Amazon Zocalo and Dat Data (dat-data.com). More recently, DataHub [15] extends the idea of checkout and checkin system to support version control over complete datasets. The system in [15] provides collaborators with the ability to load, store, query, analyze, visualize, interface with external applications, and share datasets. The three main components of DataHub are the version manager, which allows users to store and retrieve different versions of a dataset; the data-processing manager, which allows for building SQL queries by direct manipulation of tables and also for transforming unstructured data into a tabular form; and external-application manager, which supports exchanging datasets with external tools for various analysis.

OrpheusDB [16] extends DataHub by exploiting the capabilities of relational database systems. The system in [16] proposes and analyzes three models for representing dataset versions. The first model, Combined Table, enables versioning by means of a special array attribute that contains the set of versions of each record. In the second model, Split-by-vlist, a separate table is used to store the versioning information. This table maintains the mapping between versions and row IDs (rids) by using an array-type attribute that contains the versions of each rid. The third model, Split-by-rlist, is similar to the second model, but maps rids to versions by storing an array of rids for each version. The two main operations in OrpheusDB are checkout, which creates a new materialized version of a table from one or more previous versions, and commit, which adds a checkout version to the collaborative versioned dataset (CVD). Similar to our system, OrpheusDB maintains version dependency by means of a *Version Graph*, whose concept is very similar to the dependency tree in AUDIT. In order to enhance the performance of OrpheusDB, the authors present LYRESPLIT, a partitioning algorithm that optimizes the trade-off between storage and checkout latency, by breaking up the *Version Graph* into smaller subgraphs in a way that reduces the accessing of irrelevant records.

There are many differences between our proposed system and the DataHub and OrpheusDB mechanisms. First, while DataHub and OrpheusDB are specifically designed for dataset versioning, AUDIT targets versioning at the data item or data cell level. Hence, we maintain version history and dependency tree for each data item separately, as we will explain in the next sections. The applications supported by DataHub and OrpheusDB require the checkout and commit commands as operations on the whole dataset. On the other hand, the applica-

tions targeted by AUDIT, mainly large-scale collaborative environments such as Genomics and Wikis, require support for unstructured data and versioning per data item. For example, consider a scientific database in which the genomic features of different organisms are stored. When a new organism is added to the database, different scientists will work on different features and will need to continuously update specific data cells rather than creating a new version of the whole dataset. In addition, the size of the tables in these applications is much larger than those targeted in DataHub and OrpheusDB. Hence, it becomes inefficient to create a new version of the whole table which differs only in a single value from a previous version.

The second main difference is that DataHub and OrpheusDB still suffer from the commit limitation, which we explained in the Introduction. This limitation prevents collaborators from examining others' inserted versions until they are committed. As stated before, this feature is very important in collaboration environments and still missing in current systems such as DataHub and OrpheusDB. The third main difference is that while systems as DataHub and OrpheusDB can use relational database models, our approach relies on NoSQL Big Data stores for two reasons: (1) to support unstructured data, and (2) to maintain efficiency when operating on very large databases. As we will show in Sect. 7.4, systems like OrpheusDB that use relational database for versioning, will suffer from degraded performance as the size of data increases.

Data provenance management, e.g., as [17–19], or provenance support inside scientific workflow systems, e.g., as in [20,21], retain the derivation history of a data item from its original sources. Data provenance can help a scientist understand the life cycle of a data item over time, enabling the identification and correction of errors. However, provenance-based systems do not maintain the consistency of the data items, and rolling back and rejecting cascaded updates are not supported. Our system supports transaction consistency in the sense that updates that depend on rejected updates are tracked and also rejected. Also, our system is implemented inside a database system, i.e., at the engine level rather than at the application level. Therefore, the proposed system can coexist with a provenance system to achieve stronger data tracking and provenance functionalities.

Wikis have been widely used for collaborative editing of web content. They often offer version control and some history tracking features. For example, the list of all revisions of a Wikipedia page, including various metadata, can be accessed using the "View history" tab. The user can compare between any two versions, and revert to any previous version [22]. Another example is XWiki [23] which offers similar features. The main difference between AUDIT and Wiki environments is the concept of pending, approved, or rejected versions. In Wikis, an update to a page is saved directly as a new approved version. Later, if the update was found to be partially or totally wrong, a rollback is performed. In contrast, in AUDIT an update is not directly saved as a new approved version. Rather, it is marked as pending, meaning that the update has not been checked for correctness yet. In the future, when the update is checked by the PI(s), it is either approved or rejected. Another important feature that distinguishes AUDIT from Wikis is the concept of version dependency. Some Wikis, such as XWiki, allow the user to define which version of

the page was used to create a new version, which is the same as the concept of the depended-on version in our system. However, there is no Wiki system with rules for approving/rejecting a page version when it depends on other versions. In Sect. 5, we define the various dependency rules in AUDIT, which describe what should happen to child versions when their depended-on versions (i.e., parents) are approved or rejected. This feature is unique in AUDIT and distinguishes it from other collaborative systems.

In this paper, we leverage some of our previous work on data dependency in scientific databases [24,25]. In contrast to the centralized architecture we adopted earlier, we investigate the design and the performance issues related to providing the AUDIT model in a cloud environment. Unique challenges include the impact of data partitioning within the cluster (historical vs. approved, pending approval vs. current, and the effect of different partitioning policies on performance). In addition, AUDIT is the first system to define a dependency model for different versions of the same data item, and to define the rules of approval and rejection of dependent versions.

3 System overview

In this section, we provide an overview of the HBase architecture along with its data access and manipulation operations. Then, we illustrate how we extend HBase to support the AUDIT model.

3.1 An overview of HBase

HBase is a column-oriented data store running on top of HDFS [26,27]. Instead of complete rows, data is stored as fragments of columns in the form of <key, value> pairs. Fig. 3 gives the architecture of HBase. HBase contains two main components: (1) an HMaster, and (2) a set of Region Servers. The HMaster is responsible for administrative operations, e.g., creating and dropping tables, assigning regions, and load balancing. Region Servers manage the actual data stores. A Region Server is responsible for region splitting and merging, and for data manipulation operations [4]. An HBase table may have one or more regions, where each region contains HBase Store Files that are the basic data storage units. In addition, each region contains a temporary Memory Store and a Write-Ahead-Log file for fault tolerance. HBase operates alongside Zookeeper [28] that manages configuration and naming tasks.

When a client executes an HBase operation, it contacts Zookeeper to obtain the operation address of the Region Server that hosts the data on which the client will operate. Next, the client executes the required operation by accessing the region on which the desired data is stored. Although HMaster does not have a role in data manipulation operations, it operates in the background to manage and monitor administrative tasks related to tables and regions.

HBase contains four basic operations: *Put*, *Delete*, *Get*, and *Scan*. *Put* inserts the value of a data item or cell into an HBase table. Each data item or cell is identified by four fields that constitute its key, namely, *Row*, *Column Family*, *Qualifier*, and *Timestamp*. *Row* is the main identifier in an HBase table, similar to a primary key

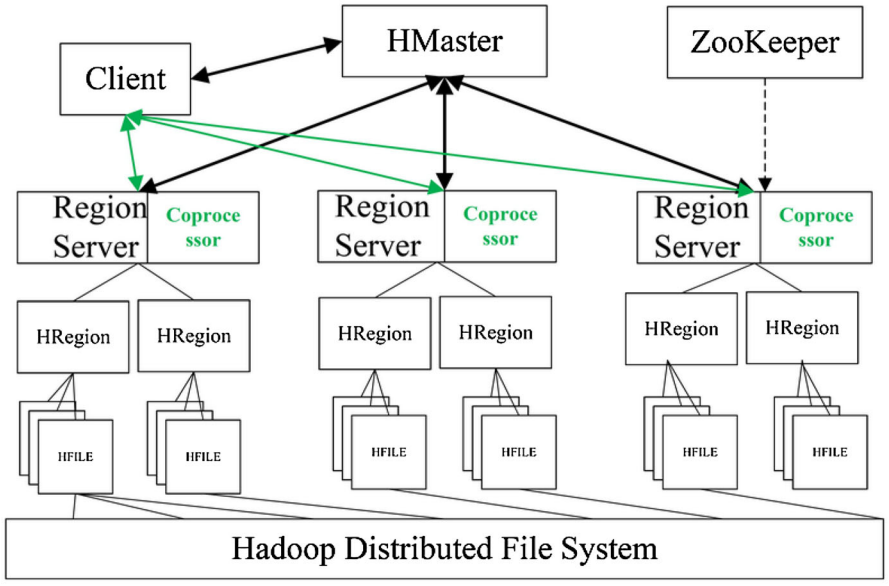


Fig. 3 HBase system structure and EndPoint Coprocessors. Our extensions are labeled in green (Color figure online)

or a tuple identifier in a relational database, *Column Family* groups related *qualifiers* together, and *Timestamp* is usually the time at which the value has been 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 of the data. Hence, HBase uses the concept of versioning in which a set of previous versions of a data cell are saved as specified by the table specifications. This feature of HBase is very useful for the history tracking required by many scientific database applications. *Delete* places a marker, termed a *tombstone*, on the data values in the memory store. When data is flushed into disk, e.g., during an Hbase major compaction operation, the data is actually deleted. Finally, *Get* retrieves data that is within a single *Row*. In contrast, *Scan* retrieves data from an entire table.

For fault tolerance, HBase implements multiversion concurrency control (MVCC) [29]. First, the new data is written into a Write-Ahead-Log (WAL) file. Then, it is written into the HBase's Memory Store. When the Memory Store reaches a certain size, data is flushed to the Store Files.

A *coprocessor* allows computational logic to be pushed into the HBase cluster, close to where data is stored [30]. *Coprocessors* extend HBase with custom functionalities by extending the operations of base classes. *Coprocessors* are divided into two main types: *Observers* and *Endpoints*. An *Observer* resembles a trigger in a relational database and executes a certain code before or after executing a client command, e.g., before executing a *Put* command or after executing a *Scan* command. An *Endpoint* runs on a specified set of Region Servers. It can be invoked at any time by the client and the results of its execution are returned to the client. In this paper, the proposed AUDIT model uses *Endpoints* for executing various operations on Region Servers.

3.2 Supporting AUDIT

3.2.1 History tracking

Data history tracking refers to the process of monitoring (i) how the data has changed since the time it has been first created, (ii) who changed the data, at what times, and possibly (iii) the reasons, if any, behind the changes. Tracking the history of data is an important operation in collaborative environments as well as in data auditing in database systems. The validity of future results depends on the correctness of existing data and how it has been derived.

History tracking is beneficial to many application scenarios. It helps answer a variety of questions including: Which users mostly insert correct data? Which inserted data is found to be faulty and hence should be rejected? How many values were rejected during the lifetime of a cell, i.e., what is the rejection percentage for the cell? What is the most common reason for rejecting values of a given cell? Has a given value been assigned before for a given cell? and if so, why has it been refuted or changed? and finally, how is the evolution history of values of a given cell?

Existing DBMSs do not provide means for answering these questions related to history tracking. For instance, HBase saves the new value of an existing cell as a new version of the cell. The user can query the value of the cell and the time at which it is inserted. Other important provenance information, e.g., the user who inserted the data, and the parameters of the environment in which the data was generated (which is useful for scientific experimentation), are not saved. Hence, the process for data history tracking misses important features and needs to be augmented to satisfy the requirements of researchers and collaborators.

AUDIT enables history tracking features by saving, with each operation on a data cell, the metadata related to the operation, e.g., the timestamp, the user ID, the type and reason for the operation, the status of the data value (see Sect. 4.2), and any other metadata related to the environment in which the data is generated, e.g., the machine ID. The proposed history tracking mechanism supports operations on saved data, e.g., reverting a data cell to a previous stable state, and building a timeline of the lifecycle of data cells.

3.2.2 Approving/rejecting updates

The process of approving or rejecting new updates is performed by privileged users, e.g., principal investigators, lab leaders, or data administrators, who monitor and approve or reject the data values inserted or updated by the various collaborators. For simplicity, we refer to a user with approval and rejection authorities as the *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 has been approved by a PI, or (iii) *rejected*: a data value that has been rejected by a PI. A PI can approve or reject data items either individually or as groups of data items through bulk approve and reject operations. In the latter case, the approve/reject interface provides the PI with a set of data items that are pending. The PI selects the data items to be approved, and the system adds them to an approve list, while the remaining items are

implicitly added to a reject list. The system performs a single operation that approves the items in the approve list and rejects those in the reject list.

Storing the data and the history of the updates over time along with the other related metadata would adversely affect the overall performance of the system. Users not interested in querying or accessing historical data should not be penalized by the potential overhead. To address this issue, we store the history of a data cell in a separate storage entity, termed the History table. We support the following three alternatives when relating history data to the original table data.

1. Store the most recently *inserted* value of a cell into the data table and mark it as *pending approval* until approved by the PI. All the other older values, whether approved or not, are stored in the history table.
2. Store the most recently *approved* value of a cell, i.e., the value that is guaranteed to be ready for consumption, into the data table. All the more recent updates to the same data cell that are still pending approval, and all older values of the cell, are stored in the history table.
3. Store both the most recently *inserted* and the most recently *approved* values of the cell in the data table and store all other older values, or the less recent values that are pending approval, in the history table. Using this alternative enables queries that use the most recent value of a cell or the most recently approved value to obtain the results directly from the original table efficiently without having to access the history table. Under this option, users explicitly interested in History Tracking must query data saved in the History table.

We refer to these three alternatives as modes for approval of updates, and explain them further below.

An important difference between our proposed AUDIT model and the update model in traditional database management systems is how the database engine treats stored data. In a traditional DBMS, the creator of the data cell is its owner and has full control over updates and deletion. The changes made to the cell will directly, or eventually, appear to all users querying the cell. In AUDIT, we treat any operation on a cell as temporary until it is approved. For example, if a user (with delete privileges) deletes the cell, we temporarily mark the cell in the original table as deleted; however, the delete operation is conditional pending the approval of the PI. Only when the delete is approved, will the actual deletion of the cell in both the original and the History table take place. If the deletion is rejected, the previous value is restored to the cell in the original table and the status of the delete record is set to “rejected” in the History table.

4 System design

4.1 History tracking

In this section, we present the proposed format of the History table and how it complies with HBase standards.

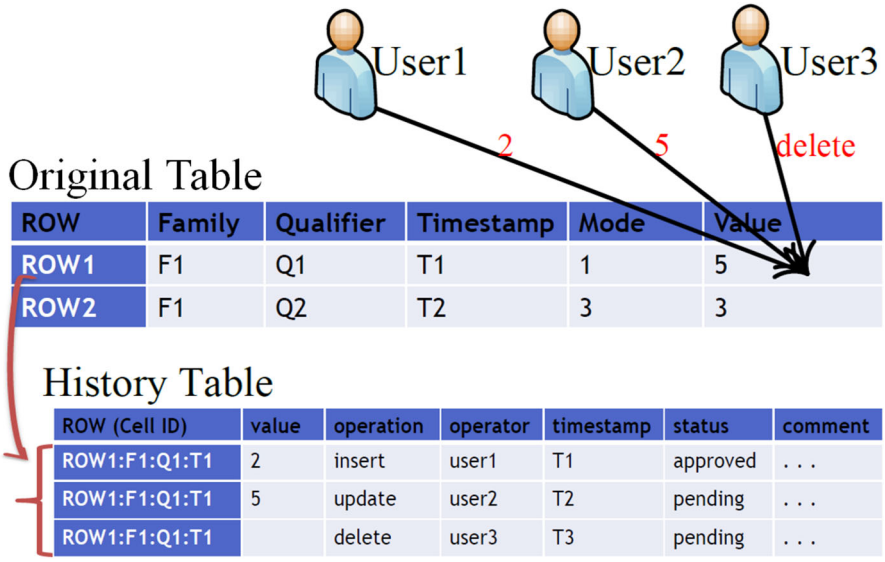


Fig. 4 Formats of records in the original and history tables

4.1.1 Record structure

To illustrate the format of a history record, we present an example in Fig. 4. Suppose that User1 inserts the cell <Table:ROW1:F1:Q1> at Time T1. The value of 2 is inserted into the original table and the first history record of the cell, with *status* equal to "pending", is inserted into the History table. Suppose that the PI approves the history record at Time T2, setting the status of the history record to "approved". Suppose that at Time T3, User2 inserts the value 5 into the cell. The cell value in the Original table is set to 5, a new history record is added to the History table with a value equal to 5, and status is set to "pending". Now, suppose that at Time T3, User3 deletes the cell in the Original table. In this case, the cell in the Original table is marked as *deleted*, and a new history record, with operation equal to "delete" and status equal to "pending" is added to the History table. The last two history records are both conditional, pending approval or rejection by the PI.

4.1.2 Table structure

In HBase, each cell is identified by its <Table, Row, Family, Qualifier> tuple, while each version of the cell is defined by its timestamp. Hence, different values that are inserted into the same cell are stored as different versions of the cell. The creator of an HBase table defines the maximum number of versions per cell, say n, that can be kept; if n is exceeded, only the newest n versions of the cell are kept and the older versions are automatically deleted.

In our system, versions are managed through the History table. Thus, data tables will all have a default value for n set to 1. In such tables, which we refer to as the *original* tables, we save either the last inserted value or the last approved value (see below). In

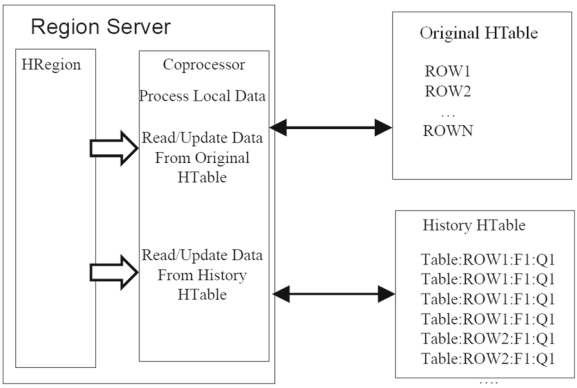


Fig. 5 The EndPoint coprocessor reads data in the local HRegion from either the original table or the History table

AUDIT Mode 3, as stated in Sect. 3.2.2, 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, each pointing to the original cell. For example, if we want to save these two values for a cell <Table, row1, family1, qualifier1>, then the last inserted value of the cell is saved in a qualifier termed 'qualifier1:current' and the last approved value of the cell is saved in a qualifier termed 'qualifier1:approved'. Using this approach, we keep the original tables as compact as possible, leading to efficient queries.

Furthermore, the maximum number of versions in the History table is set to the maximum expected number of versions per cell. Since each newly inserted value of the cell creates a new record in the History table, and we want to keep all history records as different versions of the cell, as long as the cell is not deleted; hence, the value of n for the History table is set to the maximum expected number of versions per cell for all cells of the original table. n will be specified by the creator of the original table at creation time. For example, suppose that the maximum expected number of versions of any cell in the original table will be much less than 10,000, then n can be set by the table creator to 10,000. When a cell is deleted and the delete is approved by the PI, the history records of the cell are moved from the History table to an Archive table. In this way, an archive of deleted cells is kept, enabling the restoration of a deleted cell whenever necessary. The management of the History table and the original table is handled by EndPoint Coprocessors. Figure 5 illustrates the scenario for processing data locally through a coprocessor in an HRegion in the region server.

4.2 Approval of updates

4.2.1 The approval process

A history record includes, in addition to the cell value and the timestamp at which the value was inserted, the operation (i.e., insert, update, delete, or approve), the user ID, the record status (i.e., pending, approved, or rejected), and comments, e.g., the ID of the machine used to generate the values and/or experimental parameters. History records of the same cell are automatically sorted according to their timestamps (i.e., the insertion time). Hence, it is efficient to jump to the next or to the previous records.

Notice that an HBase data cell denotes an item that is identified by its <Table, Row, Family, Qualifier>.

A PI can approve or reject pending values using two methods (see *ScHistory*, Sect. 6). In the first method, the PI specifies, in addition to the ID of the cell (i.e., Table, Row,Family, and Qualifier), 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 a range using two timestamps, the *ScHistory* operation searches the History table for all records of the cell whose timestamps fall between the specified timestamps, and then approves (or reject) these records.

Note that an inserted value should always be approved by one of the PIs; i.e., the system doesn't automatically approve or reject any value (unless in special cases explained in Sect. 5). If at any time a PI takes the role of an ordinary user and executes some *insert*, *delete*, or *update* operations, the updates remain pending until the PI switches to the PI role and performs the approval/rejection, or another PI does that.

4.2.2 AUDIT modes

Two important values of a cell need to be distinguished, namely, the last inserted value and the last approved value. These two values are typically distinct and will remain different until the PI approves (or rejects) all existing pending values of the cell. We propose three separate query modes as follows: (i) If users primarily query the most recently **inserted** value of the cell, the cell operates in **AUDIT Mode 1**. In this mode, only the most recently inserted value of the cell is saved in the original table. Each newly inserted value overwrites the previous value in the original table. (ii) If users primarily query the most recently **approved** value, this value is saved in the original table. Each newly approved value overwrites the previous approved value of the cell in the original table. In this case, the cell is operating in **AUDIT Mode 2**. (iii) If users heavily query **both** the most recently inserted and the most recently approved values, then both values are saved in the original table, and the cell operates in **AUDIT Mode 3**. Note that regardless of the current AUDIT mode of the cell, each time an operation is conducted on the cell, a history record that contains all necessary metadata is always written to the History table.

The system dynamically changes the operation of a cell from one AUDIT mode to another as follows: For each cell, the system maintains and updates three parameters: a *mode-identifier*, the *cost-rate* for searching for the most recently inserted value, and the *cost-rate* for searching for the most recently approved value. The cost-rate is calculated as the total cost based on the number of searches that occurred within a period of time divided by the length of that period in seconds. In order to calculate the total cost within a time period for searching for the most recently inserted or most recently approved values, the system periodically calculates, from the user queries, the average delay for a query that searches for a value in the original table (S_O) and the average delay for a query that searches for a value in the History table (S_H). Depending on the current AUDIT mode of the cell, the system estimates the total cost by multiplying S_O or S_H by the total number of search queries. For example, suppose a cell is currently operating in AUDIT Mode 1 (i.e., its mode-identifier = 1). Then,

the system calculates the total cost for searching for the most recently inserted value as the number of queries that search for this value within the observed period of time (t_i) multiplied by S_O, and the total cost for searching for the most recently approved value by multiplying the number of queries that search for the most recently approved value within t_i by S_H.

When the cost for searching the most recently inserted value is much higher than that for searching the most recently approved value, the system changes the cell to operate in AUDIT Mode 1 by setting *mode* to 1. If the cost for searching the most recently approved value is much higher than that for searching for the most recently inserted value, *mode* is set to 2. If both costs have close values, *mode* is set to 3. The system operations (Sect. 6) read the *mode* variable to know the current AUDIT Mode of the cell before performing the operation's tasks. For example, when the *ScGet* operation searches for the last inserted value, it examines *mode*. If *mode* is equal to 1 or 3, it searches the original HBase table of the cell. If *mode* is 2, it searches in the History table.

5 Version dependency

In the kind of environments we are considering, it is often the case that changes to a data item depends on previous versions of that same data item. In AUDIT, we consider the case where each collaborator examines the existing approved and pending versions of a cell, and performs his calculation or creates his observations based on a certain version of the cell, for example, the last inserted version, the last approved version, the first approved version ... Thus, it is important, for a user who inserts a new value or observation to the database, to specify which version it was derived from or based on (the depended-on version).

The process of identifying the depended-on version, from which each new version was derived, is important for the history of the cell. For collaborators, it helps understand the way others observe and deduce derived information. For PIs, it helps decide whether to approve or reject a new version.

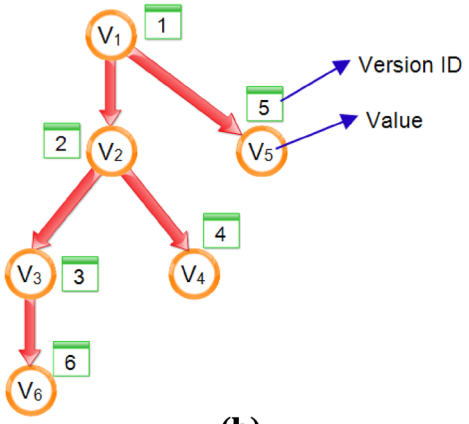
Whenever an operation is performed on a certain cell, it is important to generate a dependency tree that shows the various dependencies between the different versions of the cell. The dependency tree will help both collaborators and PIs. For collaborators, visualizing the complete dependencies between existing versions helps them decide on a start point for their experiments or calculations. For PIs, it is important to view the complete set of pending branches between the oldest pending value and the last inserted pending values to decide which branch should be approved.

5.1 Dependency model

To implement data dependency between versions, we add three fields to each record in the History table: 'Version ID', 'Depended-on Version ID', and 'Dependency Function Name'. The first version of a cell has 'Version ID' equal to 1, and has no values for the 'Depended-on Version ID' and 'Dependency Function Name'. Each new inserted version of the cell will have 'Version ID' equal to the number of existing versions of

Version ID	Depended-on Version ID	Value	Status	Time-stamp	Operation
1		V_1	approved	T_1	insert
2	1	V_2	pending	T_2	update
3	2	V_3	pending	T_3	update
4	2	V_4	pending	T_4	update
5	1	V_5	pending	T_5	update
6	3	V_6	pending	T_6	update

(a)



(b)

Fig. 6 Illustration example of **a** the History table of a cell with versions derived from each other, and **b** the corresponding dependency tree

the cell plus 1. In addition, the user inserting a new version must specify the 'Version ID' of the version from which it derives, and the name of the derivation function. These two parameters are saved in the 'Depended-on Version ID' and 'Dependency Function Name' fields of the new version. A list of dependency functions is stored in a separate table, which contains, for each dependency function, its 'Name' (unique identifier), 'Type' (for example, "computation", "experiment", "correction", etc.), 'Code' (if applicable), 'Inputs', 'Outputs', 'Estimated Cost', 'IAA' (abbreviation of *Is Automatically Approved*), and 'IAR' (abbreviation of *Is Automatically Rejected*). The fields 'IAA' and 'IAR' can have a value of either 0 or 1. More details about these two fields will be presented later.

Figure 6 illustrates an example of a cell in which versions are derived from each other, and how collaborators insert new versions. In the figure, collaborator C_1 creates cell c and inserts into it the value V_1 at time T_1. After a short while, the PI approves V_1. After that, collaborator C_2 reads the value V_1, uses it in his experiment to generate the value V_2 and then updates the cell at time T_2. Next, collaborator C_3 reads the value V_2 and starts generating a new value. During that, collaborator C_4 reads V_2 and also starts

generating a new value. Now, collaborator C_3 updates the cell with the value V_3 at time T_3. After that, collaborator C_5 reads the value V_3 and starts generating a new value, while collaborator C_6 reads the value V_1 and starts generating a new value. At time T_4, collaborator C_4 updates the cell with the value V_4, and after a while, collaborator C_6 updates the cell with the value V_5 at time T_5. Finally, collaborator C_5 updates the cell with the value V_6 at time T_6. Figure 6a illustrates the History table of the described cell while Fig. 6b shows its dependency tree.

5.2 Dependency rules

We describe now the rules that govern the insertion of new versions and the approval/rejection of versions.

5.2.1 Insertion rules

When inserting a new version, the system must check whether the depended-on version entered by the user is valid. For example, if the depended-on version doesn't exist, or if the depended-on version exists but its status is "rejected", the system produces an error message, and does not insert the new version. The system must also check that the dependency function name entered by the user is a valid 'Name' in the 'Dependency Functions' table.

AUDIT Rule 1 One of the most important rules in AUDIT is that each cell must have a single last approved state at any point of time, and not a set of *last approved* values. In other words, we cannot approve two versions that are derived from the same version. For example, in Fig. 7b, we cannot approve both V_2 and V_5. Rather, if the PI approves V_2, V_5 is automatically rejected by the system and *vice versa*.

A corollary of this rule must be applied when inserting new versions. Consider the example in Fig. 7a. Suppose the PI approves V_2, and V_5 is automatically rejected by the system. Now, suppose that before the PI approves V_2, a collaborator examines the cell and decides to do an experiment based on V_1. After the experiment is completed, the collaborator wants to insert a new value, V_7, under V_1. The system should not allow this insertion because, if V_7 is inserted and approved under V_1, there would be two *last approved* values (V_2 and V_7), which violates AUDIT Rule 1.

AUDIT Rule 2 If a new version is inserted under an old approved version V^* (other than the last approved version), then a new branch that contains two nodes must be created under the last approved version (Fig. 7b). For the example presented above, in order for the collaborator to insert his new value (V_7) and specify that it was derived from V_1, A new branch containing V_1 and V_7 must be created under V_2. The first node in this branch is a copy of the depended-on version (V_1) of the new version (V_7), and the second node is new version that is to be inserted (V_7).

By applying this solution, we avoid the necessity of rolling back to a previous state of the cell. Rather, we consider the possibility of continuing with a value, while operating under the current state. If the PI approves V_7, then the last approved value of the cell is V_7, and V_2 becomes an old approved value (Fig. 7b). Note that V_1 in cell version 7 is distinguished from other versions by the 'Operation' field in its record in

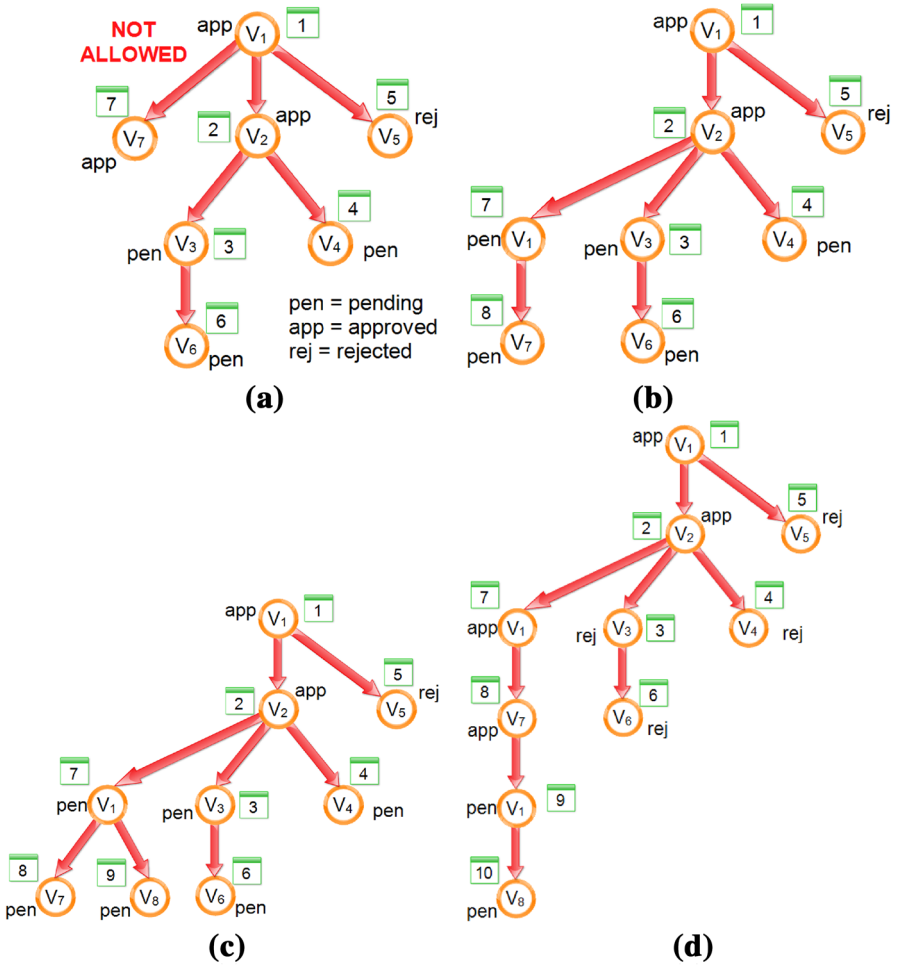


Fig. 7 Inserting a new version under an old approved version: **a** not allowed method, and **b** correct solution, and inserting another version that is derived from the same old approved version **c** before approving a new version, and **d** after approving a new version

the History table which labels it as a *new_branch*, indicating that it is not a new version, but a previously approved version that was re-added to avoid multiple last approved values. Also, this version (cell version 7) will not have a value for the 'Dependency Function Name' field.

Cell version 7 will be the parent of all new versions that depend on V_1. For example, if a new value, V_8, which is dependent on V_1 is added, V_8 is inserted under cell version 7, in cell version 9 (Fig. 7c). This remains true as long as V_2 remains the last approved version. If before inserting V_8, a child of V_2, for example, cell version 7 is approved (in this case cell version 8 will be automatically approved by the system, as we will explain in the next section), the branch replication procedure is repeated for V_8 (Fig. 7d).

5.2.2 Approval rules

AUDIT Rule 3 A non-root version cannot be approved unless its depended-on version is approved. Hence, the system examines the status of the depended-on version of V, before any approval operation on V. If the status is not equal to "approved", the approve operation is rejected. For example, in Fig. 7b, V_6 cannot be approved before V_3. This insures that, when approving a certain version, all of its antecedent versions (i.e., all previous versions in its branch) are approved.

AUDIT Rule 4 Based on AUDIT Rule 1, which states a cell can have only one last approved version, the system automatically rejects all siblings of a version V when V is approved. For example, in Fig. 7b, if V_3 is approved, both Version 4 and Version 7 are automatically rejected by the system.

AUDIT Rule 5 The single child C of a version V, is automatically approved when V is approved, if the 'IAA' field of the dependency function between V and C is equal to 1, or if there is no dependency function between V and C . More precisely, this rule states that when a version V is approved, the system checks the number of children of V. If V has only one child C, the system automatically approves C in two cases: first, the system retrieves the 'Dependency Function Name' field from the history record of C, and then examines the 'IAA' attribute of this function from the 'Dependency Functions' table. If 'IAA' is equal to 1, the system automatically approves C. For example, if C was derived from V using a mathematical function, then the system can approve C automatically with V, since computationally derived values can be directly approved when their derived-from values are approved. The choice of automatically approving a derived value depends on the derivation function. This choice is specified by the user by setting the 'IAA' field of the dependency function to 0 or 1.

The second case of automatic approval arises when there is no 'Dependency Function Name' field in the history record of C. This can occur in only one case—when V is an old approved value that was re-added as a new version to avoid having multiple last approved values. In this case, C is automatically approved as long as C is the only child of V. For example, in Fig. 7b, when the PI approves V_3, V_6 is automatically approved if the 'IAA' of the dependency function between V_3 and V_6 is equal to 1. Alternatively, if the PI approves version 7, version 8 is automatically approved. On the other hand, if the PI approves version 7 in Fig. 7c, no version is automatically approved, since version 7 has two children.

5.2.3 Rejection rules

Unlike approval, in which a version cannot be approved until its depended-on version is approved, rejection doesn't require any such condition. Hence, a certain version can be rejected at any time, regardless of whether its depended-on version is approved or pending.

AUDIT Rule 6 When a version V is rejected, the system examines each child C of V, and retrieves the 'IAR' field of the dependency function between V and C. If 'IAR' is equal to 1, the system automatically rejects C. Then if C is rejected, the system recursively rejects all children of C, in which the 'IAR' is equal to 1, and their children, etc. However, in some cases, the user might specify that the 'IAR' of a function is equal

Fig. 8 *Check New Branch* Algorithm: determines whether the new version should be inserted directly under its depended-on version or a new branch should be created under the last approved version

Algorithm: (*Check New Branch*).
Inputs: D = depended-on Version ID, d_f = dependency function name, v = value of new version.

```
(1) Begin
(2)   For each version V* of the cell
(3)       Get the 'Depended-on Version ID' of V* (D*)
(4)       If D* is equal to D
(5)                 Get the status of V*
(6)                 If status is equal to "approved"
(7)                         Set flag to true
(8)                         Break from for loop
(9)   If flag was set to true
(10)      Get the value of the depended-on version (v_d).
(11)      Get the version-ID of the last approved version (V_la).
(12)      Get the number of versions of the cells (N_v)
(13)      Call Insert New Branch function, passing to it: D, d_f, v, v_d,
(14)          V_la, and N_v
(15)  If flag was not set to false
(16)      Get the number of versions of the cells (N_v)
(17)      Call Insert New Version function, passing to it: D, d_f, v,
(18)          and N_v
(19) End
```

to 0, such as when the derived value is a correction of the derived-from value. In this case the system doesn't automatically reject the derived value when its derived-from value is rejected. Rather, the system moves such value from its current place in the tree to under the last approved value, as explained in AUDIT Rule 2, to avoid having multiple last approved values. In other words, all values that are not automatically rejected when their root is rejected, are moved from their places to new branch(es) under the last approved value.

5.3 Dependency algorithms

In this section, we present the algorithms for the AUDIT operations to implement version dependency. The first algorithm *Check New Branch* (Fig. 8) is part of the *ScPut* operation, and determines whether the new version should be inserted as a single version under its depended-on version, or whether a new branch should be created under the last approved version to avoid having multiple last approved versions. This algorithm determines whether any of the children of the depended-on version are approved. If a child of the depended-on version is approved, then the depended-on version is an old approved version and hence a new branch must be created under the last approved version.

The second AUDIT algorithm is the '*Approved and Rejected Versions*' algorithm (Fig. 9). It is used by the *ScHistory* operation when approving a version and it takes as inputs the 'Version ID' and the 'Depended-on Version ID' of the version to be approved. The algorithm consists of two parts. The first part determines the versions (or nodes) that are within the dependency subtree rooted at the version to be approved, and saves these versions in a *PotentiallyApproved* list. Nodes that are within a set of subtrees, S, where each subtree in S is rooted at a sibling of the version to be approved, are saved in a *Rejected* list. *PotentiallyApproved* is used by the second part of '*Approved*

Fig. 9 *Approved and Rejected Versions* Algorithm: determines all version that should be automatically approved and all versions that should be automatically rejected, when approving a version

```
Algorithm: (Approved and Rejected Versions).
Inputs: V = Version ID, D = depended-on Version ID.

(1) Begin
(2)   // Part 1
(3)   Define two lists: PotentiallyApproved and Rejected
(4)   Add V to PotentiallyApproved
(5)   Add D to Rejected
(6)   For each version V* of the cell, starting from D, and excluding V
(7)    and D
(8)       Get the 'Depended-on Version ID' (D*) of V*
(9)       If PotentiallyApproved contains D*
(10)                Add V* to PotentiallyApproved
(11)      Else if Rejected contains D*
(12)                Add V* to Rejected
(13)  Remove D from Rejected
(14)  // Part 2
(15)  Define list Approved and add V to it
(16)  Define version nextVersion = V
(17)  Define flag stopCheck = false
(18)  While (!stopCheck)
(19)      Define Count = 0
(20)      Define version temp = null
(21)      For each version V' in PotentiallyApproved
(22)         If 'Depended-on Version ID' of V' is equal to nextVersion
(23)                If Count is equal to 0
(24)                         temp = V'
(25)                         Increment Count by 1;
(26)      If (Count is equal to 0) OR (Count > 1)
(27)         stopCheck = true
(28)      If (Count is equal to 1)
(29)         nextVersion = temp
(30)         If nextVersion has no 'Dependency Function Name' field
(31)                Add nextVersion to Approved
(32)         Else get 'Dependency Function Name' (D_F) of nextVersion
(33)                Get 'IAA' of D_F
(34)                If 'IAA' is equal to 1
(35)                         Add nextVersion to Approved
(36)                Else stopCheck = true
(37) End
```

and Rejected Versions' to determine which nodes should be automatically approved according to AUDIT Rule 5. Each version *V* in *PotentiallyApproved* is examined to determine whether it should be automatically approved. If yes, it is added to an *Approved* list. *ScHistory* then approves each version in *Approved* by setting the 'Status' of each version in *Approved* to "approved". On the other hand, the *Rejected* list is passed to another algorithm which determines for each version in *Rejected* whether it should be automatically rejected or not, depending on the 'IAR' field in its dependency function. Figure 10 illustrates an example of the execution of the '*Approved and Rejected Versions*' algorithm.

The algorithms used when rejecting versions are similar to those in the '*Approved and Rejected Versions*' algorithm, with the exception that only a *Rejected* list is generated when rejecting a specific versions, i.e., the system only automatically rejects the versions within the subtree rooted at the version to be rejected, according to AUDIT Rule 6.

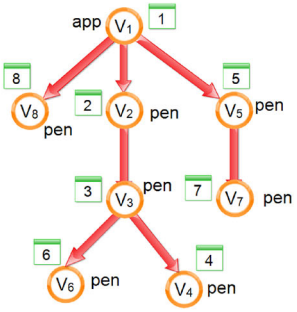


Fig. 10 Example of executing Part 1 and Part 2 of the *Approved and Rejected Versions* algorithm on a sample version that is to be approved

Algorithm: (*Approved and Rejected Versions*).
Inputs: *V = 2, D = 1.*

// Part 1
Define *PotentiallyApproved* and *Rejected*
Add 2 to *PotentiallyApproved* and 1 to *Rejected*
For each version *V**, starting from 1, and excluding 2 and 1
1 excluded
2 excluded
3 derived from 2 and *PotentiallyApproved* contains 2
=> add 3 to *PotentiallyApproved*
4 derived from 3 and *PotentiallyApproved* contains 3
=> add 4 to *PotentiallyApproved*
5 derived from 1, *PotentiallyApproved* doesn't contain 1
Rejected contains 1 => add 5 to Rejected
6 derived from 3 and *PotentiallyApproved* contains 3
=> add 6 to *PotentiallyApproved*
7 derived from 5, *PotentiallyApproved* doesn't contain 5
Rejected contains 5 => add 7 to Rejected
8 derived from 1, *PotentiallyApproved* doesn't contain 1
Rejected contains 1 => add 8 to Rejected
Remove 1 from Rejected
Finally: *PotentiallyApproved* => {2, 3, 4, 6}, Rejected => {5, 7, 8}

// Part 2
PotentiallyApproved => {2, 3, 4, 6}
Define *Approved*
Add 2 to *Approved*
Define nextVersion = 2
Define Count = 0
For each cell in *PotentiallyApproved*:
2 not derived from 2 => Count remains 0
3 derived from 2 => count = 1; temp = 3
4 not derived from 2 => count remains 1
6 not derived from 2 => count remains 1
At the end: Count = 1 => check dependency between 3 and 2
Suppose dependency 'IAA' = 1 => add 3 to Approved
nextVersion = 3;
For each cell in *PotentiallyApproved*:
2 not derived from 3 => Count remains 0
3 not derived from 3 => Count remains 0
4 derived from 3 => count = 1; temp = 4
6 derived from 3 => count = 2
At the end: count = 2 => stopCheck = true
Finally: *Approved* = {2, 3}

6 AUDIT operations

The AUDIT update approval and history tracking system, implemented using HBase, is composed of five main operations: *ScPut*, *ScDelete*, *ScHistory*, *ScGet*, and *ScGetHistory*. Each of these operations is divided into two basic components; a client API that runs at the client-side, obtains the operation inputs from the client, and formats these inputs into a Remote Procedure Call (RPC) message that is sent to the second component. Note that HBase uses Protobuf [31] to send rpc messages to the Region Servers. The second component of each operation is an HBase *Endpoint* that runs at the regions that contain the data on which the operation will execute. The correct *Endpoint* is invoked by an HBase call from the client API. For example, the *ScPut* client API invokes the *ScPutEndpoint* at the corresponding Region Servers. In case of *ScGet* and *ScGetHistory*, the results of querying the original and History tables are returned from the regions to the client API, where they are displayed. In this section, we describe the tasks performed by each of the five operations.

6.1 ScPut

ScPut inserts new data or updates existing data. It takes as input the table name and the row in which data is to be inserted, and a list of cells. Each cell is specified by its family, qualifier, and value, and optionally by its depended-on Version ID, and dependency function name. For each cell in the list, *ScPut* checks whether the cell exists in the table (by checking whether a *mode* is assigned to the cell). If the cell does not exist, *ScPut* inserts the value into the original table and inserts an initial history record in the History table (each cell starts, by default, in AUDIT Mode 1). If the cell exists, *ScPut* checks the History table to see if the most recent history record of the cell has 'Operation' equal to "delete" and 'Status' equal to "pending". If yes, the cell has been deleted and no updates to the cell are allowed until the delete is approved or rejected. Hence, the update to the cell is rejected. If the cell has not been deleted, *ScPut* executes the '*Check New Branch*' algorithm (Sect. 5.3). Based on the results, *ScPut* either inserts a single new history record into the History table, or inserts two history records which constitute a new branch as explained in Sect. 5.3. *ScPut* then checks the *mode* of the cell. If *mode* is 1 or 3, *ScPut* updates the cell in the original table with the new value.

6.2 ScDelete

ScDelete marks the data as deleted, pending PI approval. Similar to *ScPut*, *ScDelete* takes as input a table name, a row name, and a list of cells (family and qualifier). The initial tasks performed by *ScDelete* are similar to those of *ScPut*: It checks whether the cell exists. If the cell does not exist, *ScDelete* continues to the next cell. If the cell exists, then if the cell has already been deleted, *ScDelete* continues to the next cell. If the cell has not been deleted, *ScDelete* inserts a new history record into the History table and checks the *mode* of the cell. If *mode* is 1 or 3, *ScDelete* marks the value of the cell in the original table as "deleted".

6.3 ScHistory

ScHistory is used by PIs to approve or reject pending updates. The inputs to *ScHistory* are the operation ("approve" or "reject"), a list of input cells (Table, Row, Family, Qualifier), and a list of timestamps (which represent the versions that are to be approved or rejected). For each cell, *ScHistory* obtains the timestamp of the first history record of the cell (t_{min}) from the History table , and the *mode* of the cell from the original table. If the operation is "approve", then, for each timestamp, t, in the input cell list, *ScHistory* attempts to approve the record of t according to the following rule: if t is equal to t_{min} (i.e., t is the first record), *ScHistory* writes "approved" to the record of t in the History table, and updates the value of the cell in the original table if *mode* is 2 or 3. *ScHistory* then executes the '*Approved and Rejected Versions*' algorithm to determine the records that should be automatically approved, and approves these records. If t is greater than t_{min}, *ScHistory* checks that the 'Status' of the 'Depended-on Version ID' of t is "approved". Then *ScHistory* checks if the operation in t's record is "delete". If yes, ScHistory moves the cell history records to an Archive table, deletes all cell related qualifiers in the original table, and deletes the cell search most recently inserted and search most recently approved rates and its *mode* variable. If the operation is not "delete", *ScHistory* executes the '*Approved and Rejected Versions*' algorithm. After obtaining the result, *ScHistory* approves the records that should be automatically approved and calls the Rejection algorithm in order to reject the records that should be automatically rejected. Next, *ScHistory* writes the last approved value to the original table if *mode* is equal to 2 or 3. If the input operation is "reject", *ScHistory* rejects the history record in a similar way to that of an "approve".

6.4 ScGet

ScGet retrieves either the most recently inserted, the most recently approved, or both values of one or more cells. *ScGet* takes as input a table name, a list of cells, and a search type (most recently inserted, most recently approved, or both). *ScGet* obtains the *mode* of each cell in the list. Then, according to the search type, *ScGet* obtains the most recently inserted, the most recently approved, or both values of the cell from the original and/or the History table, according to the *mode* of the cell.

6.5 ScGetHistory

ScGetHistory looks up the history of one or more cells in the History table and returns the values of the history records. If the user specifies a list of cells, *ScGetHistory* defines a scan on the History table and adds the list of cells to the scan. Otherwise, *ScGetHistory* scans the History table over the complete list of cells from the original table. *ScGetHistory* uses one of three HBase custom-built filters for the scan, according to whether the user requires only approved values, only pending values, or both approved and pending values.

7 Experimental study

7.1 Dataset and testing environment

We realize and test our system within a virtual HBase Cluster that consists of twenty five virtual machines (VMs) that are divided into two sub-clusters: the first sub-cluster contains fifteen VMs that store the HBase database on which the AUDIT operations are tested, while the other sub-cluster consists of ten VMs, each of which hosts a client who executes the AUDIT operations on the database. The testing procedures is divided into two main parts. In the first part, we test each of the six AUDIT operations (which are Insert, Delete, Approve, Reject, Search-last-inserted and Search-last-approved) alone. In this part, each of the ten clients executes a workload of 10,000 queries of the tested AUDIT operation. In the second part, each of the ten clients runs a mixed workload of the six AUDIT operations. Details about how AUDIT operations are distributed within the mixed workload will be presented shortly.

Among the fifteen VM sub-cluster that holds the database, one is used as the Hadoop master node and the HBase master, while the other VMs act as Hadoop slaves. The data used to create the database and to test the AUDIT operations was extracted from dumps provided by Wikipedia [32], which gives the history of updates performed by Wikipedia users on Wikipedia pages. We consider each Webpage as a data cell and each update on the page as a new version of the data cell that is written as a new record to the History table. The Wikipedia data is formatted in XML files. We used the SAX XML parser to extract, for each webpage revision, the pieces of data that are inserted in the HBase database, e.g., Revision ID, size, md5, user ID, timestamp, and comment. into a csv file. The data in the csv files are read by the clients and inserted in the HBase database. In our experiments, we tested four database sizes: 10, 100, 500 GB and 1 TB. For each of these four size, we create an HBase database of the corresponding size, then we test all six AUDIT operations separately as well as using a mixed workload as previously explained.

When running a mixed workload scenario, the 10,000 queries will be a mixture of the six AUDIT operations. The percentage of Delete queries to Insert queries is set using a parameter p_1, while the percentage of (Insert + Delete) queries with respect to (Search-last-inserted + Search-last-approved) queries is set using a parameter p_2. The number of (Approve + Reject) queries is set to be equal to that of (Insert + Delete) queries (since each Insert and Delete operation will be either approved or rejected). Also, the number of Search-last-inserted queries is selected to be equal to that of Search-last-approved queries. We calculated from the Wikipedia dataset the values of p_1 and p_2, which were found to be equal to approximately 0.01 and 0.07% respectively. The four database sizes were tested in all three AUDIT modes, i.e., we tested on each database size, a separate scenario for each of the three AUDIT modes. We also tested, for each database size, a dynamic-mode scenario in which the workload parameters were varied over time, such that data cells will change their AUDIT modes while the scenario is running. This allows us to evaluate the effectiveness of dynamic mode adaptation.

In the dynamic mode, we divide the time into three equal parts: In the first part, the ten clients target 90% of their search queries to the most recently inserted values,

and the other 10% to the most recently approved values. In this case, the best AUDIT mode to use is AUDIT Mode 1. In the second part, the ten clients target 90% of their queries to the most recently approved values, and the other 10% to the most recently inserted values. The best AUDIT mode for this case is AUDIT Mode 2. In the third part, the search queries target both the most recently inserted values and the most recently approved values equally. Hence, the best AUDIT mode for the third part is AUDIT Mode 3. A continuously running *ScMode* thread calculates the search-last-inserted and search-last-approved cost rates for each cell, and dynamically changes the AUDIT mode of the cell to the most suitable one, according to the values of the two rates.

The results of the experiments are presented in the next sections. First, we illustrate the results that were obtained from executing workloads that contain a single operation each, then we present the results of mixed AUDIT operations workloads. Finally, we compare AUDIT results with those of OrpheusDB.

7.2 Results of standalone AUDIT operations

Figures 11, 12 and 13 show the total query delay for each of the six AUDIT operations: Insert, Delete, Approve, Reject, Search-most-recently-inserted, and Search-most-recently-approved. Each point in the graphs of these three figures is the average of running the standalone AUDIT operation workload on the ten clients, where the number of queries per client workload was set to 10,000. Note that the Wikipedia dataset does not include any dependency information between Webpages. Hence, dependencies between different versions of each Webpage were generated randomly as follows: when inserting a new version, the system reads the Version IDs of all existing versions of the cell, then it selects a random version as the depended-on version.

Figure 11a shows that AUDIT Mode 2 has the least insertion delay, which is expected because it inserts data to the original table only when approved, while AUDIT Mode 1 inserts data to the original table when data is inserted, and also when a history record is rejected. In contrast, AUDIT Mode 3 inserts data to the original table at insertion, approval, and rejection and hence yields the highest delay. For all three modes, the delay decreases as the input data size increases. This behavior is also observed for the Delete, Approve, and Reject operations (Figs. 11b, 12a, b), indicating that

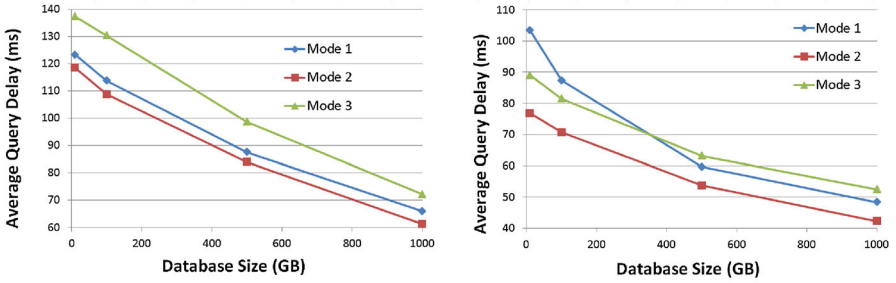


Fig. 11 Results of standalone operations: the effect of varying the data size on the query delay of **a** Insert and **b** Delete

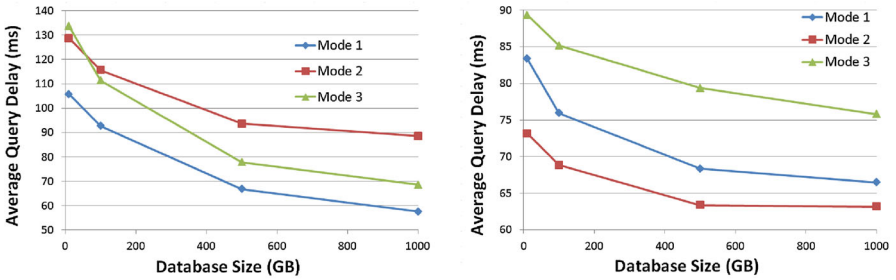


Fig. 12 Results of standalone operations: the effect of varying the data size on the query delay of **a** Approve and **b** Reject

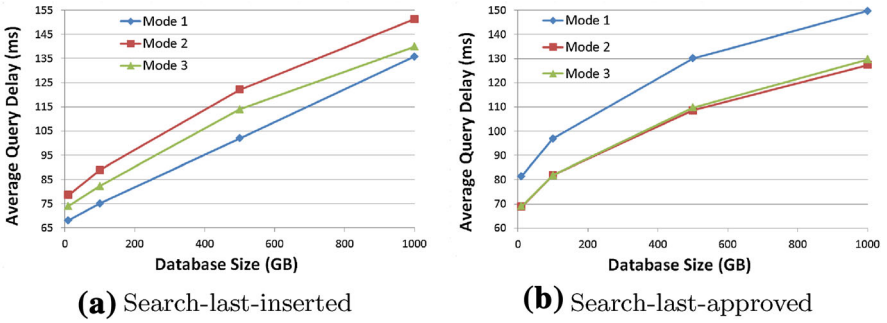


Fig. 13 Results of standalone operations: the effect of varying the data size on the query delay of **a** Search-last-inserted and **b** Search-last-approved

AUDIT performance, for these operations, improves as the database size increases. Since HBase is designed for massive data sets, it generally performs better with large data sizes than with small sizes.

Figure 11b shows that the Delete operation has a similar behavior to the Insert delay. Generally, the Delete algorithm has only minor differences in the three AUDIT modes. As for Approve, AUDIT Mode 1 has significantly lower Approve delay, because it approves the cell only in the History table, while AUDIT Mode 2 and AUDIT Mode 3 require writing to the original table when approving. With respect to Reject, Fig. 12b illustrates that AUDIT Mode 2 has significantly lower delay, because it does not need to write to the original table since the last approved value already exists in the original table. On the other hand, AUDIT Modes 1 and 3 require restoring the previous value to the original table when the current value is rejected, causing an additional delay. AUDIT Mode 3 shows higher delay than AUDIT Mode 1, because it requires restoring both the most recently inserted and the most recently approved qualifiers of the cell in the original table.

For searching, Fig. 13a, b illustrate the importance of saving the most needed values in a separate table. AUDIT Mode 1 (Fig. 13a) has a significantly lower delay than AUDIT Mode 2 when searching for the most recently inserted value, because it saves the most recently inserted value in the original table, while in AUDIT Mode 2

this value is retrieved from the History table. On the other hand, (Fig. 13b) illustrates that AUDIT Modes 2 and 3 have significantly lower delay than AUDIT Mode 1 when searching for the most recently approved value for the same reason. The delays in Figs. 13a, b increase as the size of data increases, because the number of regions increases, and the regions are distributed on different VMs. In this situation, there is a higher probability that the region that contains the target data is on another VM. With smaller amounts of data, data locality is better preserved.

From the results, we notice that each of the three AUDIT modes has its own advantages and disadvantages. AUDIT Mode 3 performs best when searching for both the most recently inserted and approved values. However, AUDIT Mode 3 has more delays when inserting, approving, or rejecting. In contrast, AUDIT Mode 1 is best when searching for the most recently inserted values, but has higher delays over AUDIT Mode 2 when inserting or rejecting. AUDIT Mode 2 is best in searching for most recently approved value, Insert, and Reject operations, but leads to higher delay for the Approve operation. In order to exploit the advantages of each AUDIT mode, we proposed the dynamic mode in which the AUDIT mode is dynamically changed according to the AUDIT operations that are dominating the workload of a data cell. In the next section, we present the results of workload scenarios that contain mixture of AUDIT operations, and we illustrate how varying the AUDIT modes based on the AUDIT operations in the workload will improve the general performance.

7.3 Results of mixed workloads

The results of the previous section reflect the behavior of each of the operations of the system alone. In this section, we test the performance of the system while running a workload that contains a mix of the six operations. The experiment in this section was repeated for each of the three AUDIT modes. In addition, we add a fourth dynamic mode scenario in which the AUDIT mode is changed dynamically according to the AUDIT operations in the workload (as explained in Sect. 4.2.2).

Figures 14, 15 and 16 show the results for the six operations analyzed in the previous section when these operations are running together in the same workload. If we compare the results of these three figures with those of Figs. 11, 12 and 13, we notice that the delays of the six operations increases slightly when running in a workload that contains other operations as compared to when the operation is running alone. This increase is noticed in the delay of all six operations. However, this increase is limited and reaches a maximum of 11 ms for the Approve operation in AUDIT Mode 2. In general, the behavior of the six AUDIT operations is similar when running alone or within a mixed workload.

From Fig. 14a, we notice that the data insertion delay of the dynamic mode scenario is close to that of AUDIT Mode 1 when the data size is small, and almost equal to that of AUDIT Mode 1 when data size is large. This reflects that the Insert delay of the dynamic-mode becomes near to that of the best delay When the data size increases. For all data sizes, the Insert delay of the dynamic mode is less than that of AUDIT Modes 2 or 3. Also, the Delete delay of the dynamic mode is similar to that of the lowest Delete delay (AUDIT Mode 2) for all data sizes, as Fig. 14b illustrates. For

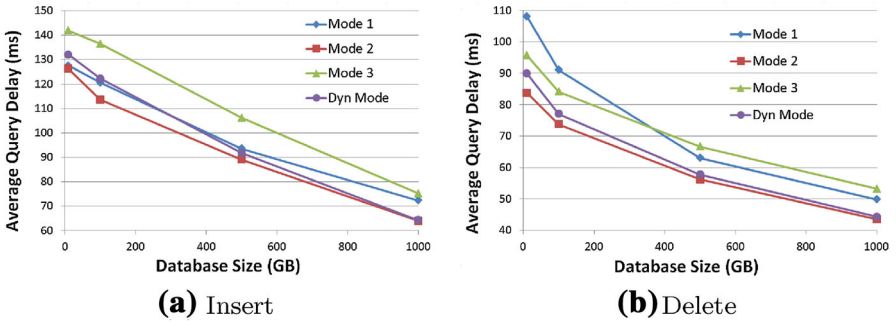


Fig. 14 Results of workloads with mixed operations: the effect of varying the data size on the query delay of **a** Insert and **b** Delete

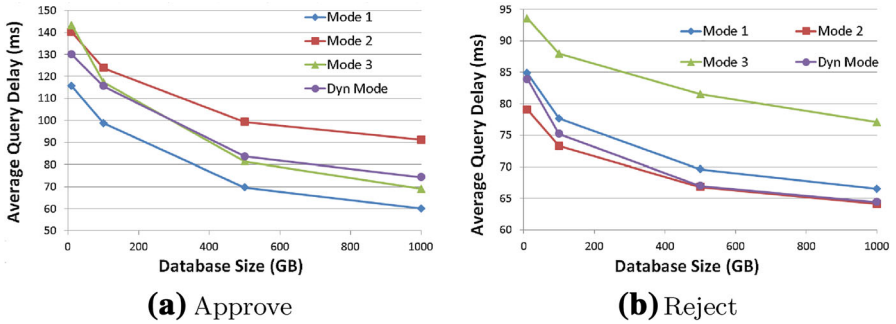


Fig. 15 Results of workloads with mixed operations: the effect of varying the data size on the query delay of **a** Approve and **b** Reject

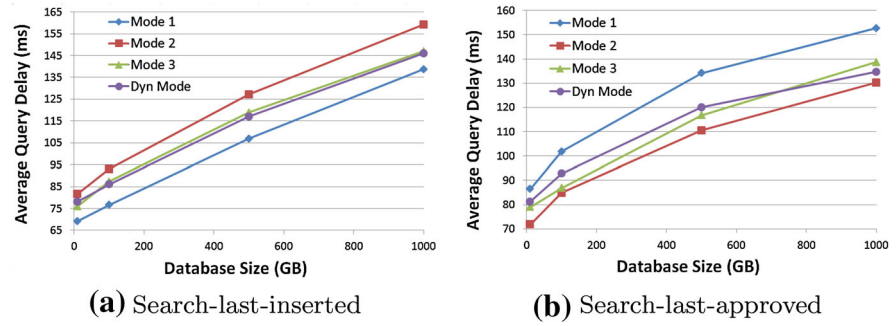


Fig. 16 Results of workloads with mixed operations: the effect of varying the data size on the query delay of **a** Search-last-inserted and **b** Search-last-approved

Approve, Fig. 15a shows that the Approve delay of the dynamic mode is somehow in the middle between the delays of the three modes, and is less than the delay of AUDIT Mode 2 (highest delay) for all data sizes. Similar to Approve, the Reject delay of the dynamic mode scenario is less than the delays of AUDIT Modes 1 and 3 for all data sizes, also approximately equal to that of the lowest delay of AUDIT Mode 2 when the data size is large. As for searching, the Search-last-inserted and Search-last-approved delays of the dynamic mode scenario (Fig. 16a, b) are close to the best possible search delay, and less than the highest search delay for both operations.

In general, the results in this section show that, for all the six operations, the dynamic mode scenario exhibits a delay which is equal or slightly larger than the best delay, but much less than the worst delay. Hence, the dynamic mode scenario is the best scenario that guarantees good performance for all operations. This makes the dynamic mode suitable for both database users and PIs. On the other hand, non-dynamic mode scenarios can have good performance for some operations but weak performance for others.

7.4 Comparing with OrpheusDB

To the best of our knowledge, AUDIT is the first system that maintains version history and version dependency per data cell, and allows collaborators to view pending versions and update them, in addition to maintaining dependency rules for Insert, Approve, and Reject operations. Other systems in the literature (discussed in Sect. 2) contain similar functionalities to AUDIT, but on the whole dataset scale. In this section, we experimentally compare between one of these systems, OrpheusDB, and AUDIT. First, we note that OrpheusDB operations (Checkout and Commit) are performed on the whole dataset, while AUDIT operations are performed on a single data cell (i.e., single value). In order to be able to compare the two systems, we classify their similar operations. Checkout in OrpheusDB is similar to Insert in AUDIT, since both operations create a new version in the database. The difference between OrpheusDB checkout and AUDIT Insert is that the version created by checkout is available only to its owner, while the version created by Insert is available to all users as a *pending* version. Also, Commit in OrpheusDB is similar to approve in AUDIT, as both declare a certain version as approved by the DB administrator; with the difference that OrpheusDB Commit makes the version available to all users, while AUDIT Insert changes the version *status* to "approved", and also approves all dependent versions that should be automatically approved.

The simulation environment for this section is as follows: we use the same Wikipedia data from [32]. We create one table in the database, called Wiki, which contains six attributes that represent the Webpage {Revision ID, size, md5, user ID, timestamp, and comment}. Only one CVD is considered, which is the Wiki table. Each revision in the Wikipedia file will be executed as a checkout operation in OrpheusDB, where the -t argument of checkout is chosen randomly as one of the previous revisions of the same webpage. In each scenario, we keep reading webpage revisions from the Wikipedia file and executing each revision as a checkout until we reach a certain total dataset size. Similar to the previous section, we test four dataset sizes: 10, 100, 500 GB and 1 TB. Periodically, an algorithm is run that selects a random number of checkout versions and commits them. The average time between two consecutive commits was set to 10 s. We implemented the Split-by-rlist model of OrpheusDB, which was supported by the LyreSplit partitioning algorithm to enhance its performance. The δ input parameter to LyreSplit was set to 0.5 (middle trade-off between storage and latency). We compare the results of Checkout and Commit with those of Insert and Approve that were obtained by running the dynamic Mode of AUDIT under the same simulation environment. The results are shown in Fig. 17.

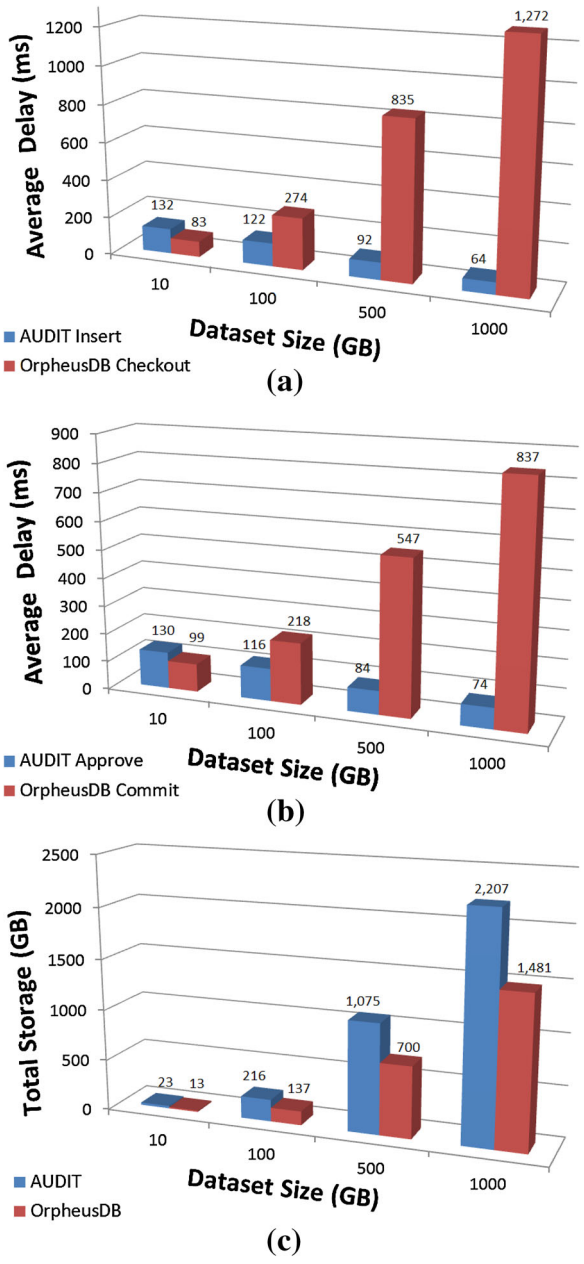


Fig. 17 Comparing AUDIT with OrpheusDB: **a** AUDIT Insert versus OrpheusDB Checkout, **b** AUDIT Approve versus OrpheusDB Commit, and **c** total storage space required by AUDIT and OrpheusDB for different dataset sizes

From Fig. 17, we notice that OrpheusDB performs well for a small size dataset, but the latency of its operations increases rapidly for large datasets. In fact, OrpheusDB performs better than AUDIT when the dataset is less than 30 GB. However, the delays of Checkout and Commit become much higher than those of Insert and Approve for large datasets (more than 100 GB). The reason for this degraded performance of

OrpheusDB is that as the dataset increases to a very large size, the number of versions per row id will also increase. In our simulations, as the number of revisions of the same webpage increases to tens or hundreds of thousands, the number of rows per version in the versioning table also increases causing the checkout and commit to search within a huge number of rows. This problem is eliminated in AUDIT since the search is within the data cell only, and not within the whole table, since versions are per data cell. For this reason, systems like DataHub and OrpheusDB will perform well as long as the average number of versions per row is less than a certain limit. In scientific applications, the number of versions per row can increase to a very high value, especially when a new organism or feature is being experimented by thousands of scientists, each of which will generate new results and update the data cell with new values very frequently. In such scenarios AUDIT will give better performance, as it will allow collaborators to access the version history of the specific cell that they want in an efficient manner.

This supremacy in AUDIT performance from the latency perspective comes at the expense of storage. From Fig. 17c, we notice that OrpheusDB requires an overhead between 29 and 48% of the dataset size, while AUDIT results in a storage overhead between 115 and 134%. This large storage overhead in AUDIT is due to the fact that each new version that is added to the original table will result in six new attributes that are added to the History table, as we explain in Sect. 5. While in OrpheusDB the versioning table will contain only a single attribute that combines information about many versions in a single array. This approach reduces the storage overhead, but still requires large delay for inserting and approving new versions, especially when the dataset size is very large, as Fig. 17a, b illustrate. In big data applications, additional storage overhead is not a big issue, since the cluster can be extended to support new nodes. Hence, it is often the case that storage is sacrificed to obtain better response time in this type of applications.

8 System usability

An important factor that helps in determining the overall strength of a system is its usability. The latter is defined as "the degree to which a software can be used by its consumers to achieve the required objectives with effectiveness, efficiency, and satisfaction" [33]. No matter how much a system is important for a certain application, its overall quality is diminished if users of the system find major difficulties in working with the system, such as the inability to understand and use the system functionalities, finding it hard to work with system interface, or not being able to deal with system feedback. Such factors decrease the system usability and make its consumers less willing to use it.

For these reasons, we tested the usability of the proposed AUDIT system by conducting a user study that involved two experts in Bioengineering and Bioinformatics. First, in order to enable the participants in the user study to interact with AUDIT, we developed a user-friendly user interface that enables an ordinary user who has no experience with databases to use the system. Next, we gave each of the two experts a laptop that has AUDIT installed, and asked them to perform a testing experiment that

includes creating a scientific database inside AUDIT, extracting data manually from the GenBank online Database, and inserting this data into AUDIT database using our developed user interface, then performing approve and reject operations on the data also using AUDIT user interface. At the end, we conducted a brief survey in which the two experts answered several questions regarding the system usability. In the next sections, we provide a brief description of the user interface of AUDIT, then we describe the list of operations that were performed by the experts. Finally, we present the results of the user study.

8.1 AUDIT graphical user interface

A user-friendly graphical interface was developed on top of AUDIT. The GUI displays any cell in the database as a dependency tree and enables the user to perform various operations on the cell by interacting directly with the tree. If the user wants to insert a new cell, the interface displays a form in which the user inserts the new cell parameters, which are the Table, Row (key), Family, Qualifier, and value of the first version of the cell. When the user wants to perform other operations such as update the cell (i.e., insert a new version) or approve/reject a version or set of versions, the interface displays the cell dependency tree (Fig. 18). The user selects the operation from the "Operations" menu, clicks on the version(s) in the dependency tree that he wants to include in the operation, and enters the required values of the operation inputs in their corresponding textboxes (if any). When the user executes the operation, the system updates the cell versions in the database, then updates the dependency tree in the interface. Figure 18 illustrates how an AUDIT approve operation is performed on multiple versions of a sample cell. Note that each node in the dependency tree contains three elements separated by commas: The version ID, *status* (P for *pending*, A for *approved*, and R for *rejected*), and value.

8.2 AUDIT user study

We selected two experts in the field of Bioinformatics and explained to them the concept of UPA and the details of the AUDIT operations. We ran a sample scenario similar to the one illustrated in the previous section to make them familiar with the AUDIT user interface. Then we asked them to perform a test on real scientific data that they should extract from the GenBank database. GenBank [34] is a collaborative genetic sequence database offering an annotated collection of all publicly available nucleotide sequences and their protein translations. GenBank is maintained by the National Center for Biotechnology Information (NCBI) as a part of the International Nucleotide Sequence Database Collaboration (INSDC). GenBank data is produced in laboratories around the world from more than 300,000 organisms. GenBank accepts only original sequences submissions, which are vetted for originality by the GenBank staff (PI in our system). Each of the experts performed the test study alone, in which he was required to extract the data of chromosome 21 sequence in the human genome, and enter this data to AUDIT database to create a dependency tree for chromosome 21 similar to the one shown in Fig. 19. All inserted values are

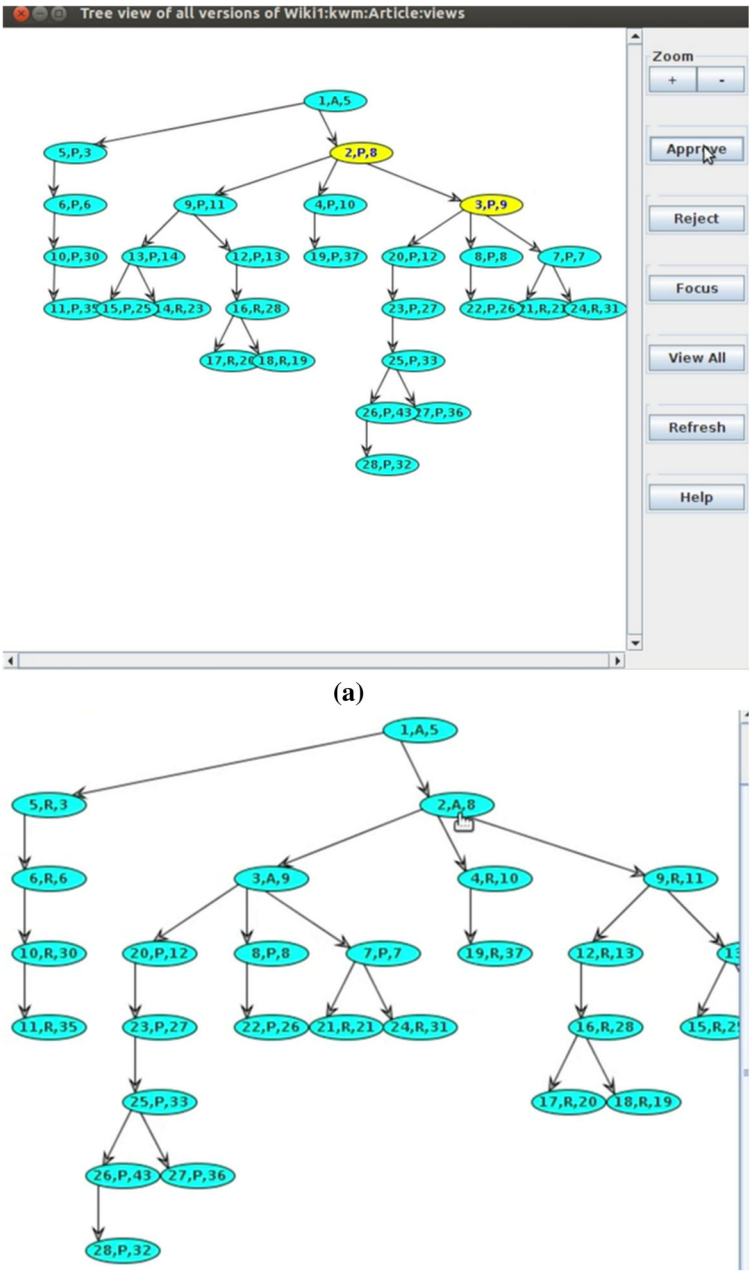


Fig. 18 Example of approving multiple versions: **a** The user selects versions 2 and 3 and clicks on the "Approve" button, **b** the system changes the status of the selected versions to *approved*. Also, version 5, which is a sibling of 2, and all versions within the subtree of 5, are automatically rejected by the system, and the same for versions 4 and 9, which are siblings of version 3 (the 'IAA' and 'IAR' fields were set to 1 in this example)

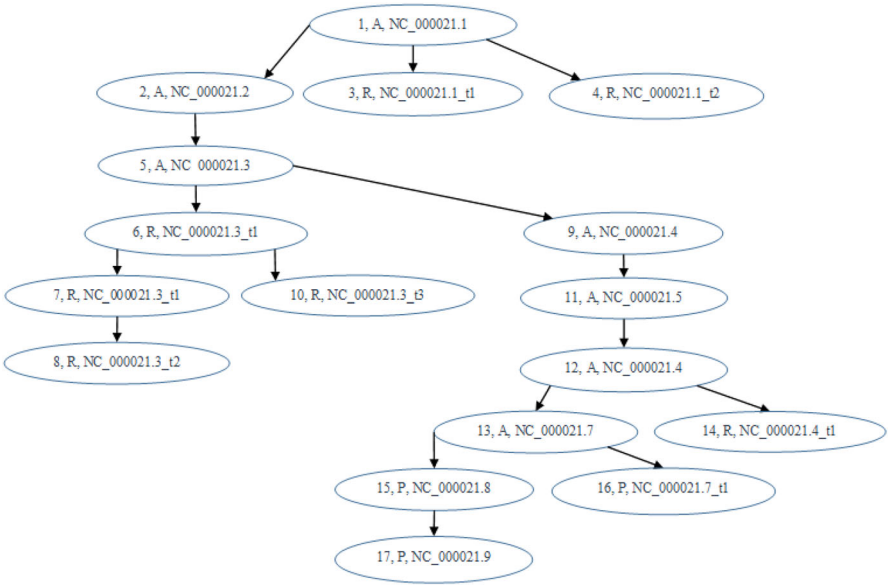


Fig. 19 The dependency tree for chromosome 21 sequence versions from the GenBank database

first saved as *pending* versions. After that, the expert was required to test the various operation of AUDIT by performing the following operations on the dependency tree of Fig. 19 (note that the 'IAA' and 'IAR' fields were always set to 1 in this example):

1. Approve version 1.
2. Approve version 2, make sure that versions 3 and 4 are automatically rejected.
3. Approve version 5.
4. Approve version 9, and make sure that the system automatically rejects version 6 and all versions within its subtree. And also make sure that versions 11 and 12 are automatically accepted.
5. Insert versions 13 and 14 as children (depend on) version 9. The system should automatically move them under the last approved version, which is version 12.
6. Insert versions 15 and 16 as children of version 13. Insert version 17 as child of version 15.
7. Approve version 13, the system should automatically reject version 14.

After performing these operation the expert should produce a dependency tree same as that in Fig. 19. After finishing the experiment, each expert was asked to fill an evaluation report, which we discuss in the next section.

8.3 User study results

The user study evaluation report that was filled by the experts at the end of the experiment contained ten questions that were designed to test the experts' satisfaction with the AUDIT system. The questions are shown in the second column of Table 1, while

Table 1 Experts' answers for AUDIT usability user study questions

Question no	Question	Expert 1 answer	Expert 2 answer
1	To what extent you consider the software important for the scientific community?	9	8
2	Were you able to easily understand the key features and functionalities of the system?	9	7
3	To what extent is a new user with no previous experience with the software able to successfully perform the system operations?	7	7
4	To what level do you consider the graphical interface user-friendly?	9	9
5	How do you rate the system average response time for various operations?	10	8
6	How do you rate the clarity of system outputs and system feedback?	9	7
7	To what extent do you think the system features are complete?	8	7
8	What is the level of help offered by the system to the user?	7	7
9	How many errors did you encounter during the experiment?	0	1
10	How many times were you stuck and needed help to continue?	0	0

the third and fourth column of the table show the experts answers to the questions. Questions 1–8 were answered on a scale of 10, while questions 9 and 10 required a regular numeric answer. Among the ten questions, Question 1 was asked to test the importance of the system, Questions 2 and 10 were asked to test the understandability of the system, Questions 3 and 8 test the easiness of the system, Questions 4 examines the system user-friendliness, Question 5 was used to test the system efficiency, Question 6 reflects the clarity of the system, Questions 7 is related to the system completeness, and Question 9 demonstrates the effectiveness of the system.

Each expert was required to provide an answer between 1 and 10 to Questions 1 to 8, and to state the number of errors that he encountered and the number of times he was stuck in Questions 9 and 10. In order to calculate an average score of the system from the experts' answers, we reverse the answers of question 9 and 10 on a scale of 10. In other words, an answer of 0 to these questions means a grade of 10, an answer of 1 means a grade of 9, and so on. In Table 1, expert 1 stated he encountered zero errors, which means a grade of 10 on this question, while expert 2 encountered one error, which means a grade of 9. Hence, if we calculate the average grade for the ten questions, the system score will be 8.8 from expert 1 answers and 7.9 from expert 2 answers, with a total average of 8.35/10.

In general, the user study reflects that both experts were able to carry out the experiment and perform the AUDIT operations smoothly. The most important lesson that we conclude from the user study is that the system didn't produce any serious

errors while both experts were performing the experiment. Both experts were able to generate the exact dependency tree of Fig. 19 by successfully executing the steps in Sect. 8.2. Both experts stated that the operations, outputs, and feedback were clear, the response time was very fast, and the interface is user-friendly. The experts commented that the system help should be slightly extended to contain more details about how each operation work, which will help a new user understand the operations better and assert the correctness of the outcomes of each operation.

9 Conclusion

In many existing applications, the correctness of inserted data is based not only on the identities of the users who insert the data, but also on the values of the data itself. It is crucial for these applications that the underlying DBMS enables the review and approval of data, and permits users to view its status and history. In this paper, we presented a mechanism for collaborative databases, implemented on a cloud platform, to handle data dependency between different versions of a data item. In our system, data is classified based on its status: approved, pending approval, or rejected. The system identifies dependencies between different versions of data items and proposes mechanisms for handling dependencies. Our results illustrate the advantages and disadvantages of each of the system modes. One of the important features of our design is its ability to dynamically switch between different modes, thereby adapting to changes in the workload in order to achieve the best performance.

Acknowledgements This publication was made possible by the support of an NPRP Grant 4-1534-1-247 from the the Qatar National Research Fund (a member of Qatar Foundation) and the National Science Foundation under Grants IIS-1117766 and IIS-0964639. The statements made herein are solely the responsibility of the authors.

References

1. BLAST. http://blast.ncbi.nlm.nih.gov/Blast.cgi/
2. Fagin, R.: On an authorization mechanism. ACM Trans. Database Syst. **3**(3), 310–319 (1978)
3. Griffiths, P.P., Wade, B.W.: An authorization mechanism for a relational database system. ACM TODS **1**(3), 242–255 (1976)
4. Apache hbase. https://hbase.apache.org/
5. Mershad, K., Malluhi, Q., Quzzani, M., Tang, M., Aref, A.: Approving updates in collaborative databases. In: Proceedings of the 3rd IEEE International Conference on Cloud Engineering. IC2E 15 (2015)
6. Dayal, U., Hsu, M., Ladin, R.: Organizing long-running activities with triggers and transactions. SIGMOD Rec. **19**(2), 204–214 (1990)
7. Garcia-Molina, H., Salem, K.: Sagas. SIGMOD Rec. **16**(3), 249–259 (1987)
8. Aiken, A., Hellerstein, J., Widom, J.: Behavior of database production rules: termination, confluence, and observable determinism. In: SIGMOD (1992)
9. Paton, N.W., Daz, O.: Active database systems. ACM Comput. Surv. **31**(1), 63–103 (1999)
10. Lomet, D., Barga, R., Mokbel, M., Shegalov, G.: Transaction time support inside a database engine. In: Proceedings of the 22nd International Conference on Data Engineering (ICDE 06) (2006)
11. Oracle flashback. http://www.oracle.com/technetwork/issue-archive/2008/08-jul/o48totalrecall-092147.html/

12. Shankar, S., Kini, A., DeWitt, D.J., Naughton, J.: Integrating databases and workflow systems. SIGMOD Rec. **34**(3), 5–11 (2005)
13. Apache subversion. http://subversion.apache.org/
14. Git. http://git-scm.com/
15. Bhardwaj, A., Deshpande, A., Elmore, A.J., Karger, D., Madden, S., Parameswaran, A., Subramanyam, H., Wu, E., Zhang, R.: Collaborative data analytics with DataHub. Proc. VLDB Endow. **8**(12), 1916–1919 (2015)
16. Xu, L., Huang, S., Hui, S., Elmore, A.J., Parameswaran, A.: ORPHEUSDB: a lightweight approach to relational dataset versioning. In: Proceedings of the ACM International Conference on Management of Data, ACM 2017, pp. 1655–1658 (2017)
17. Bhagwat, D., Chiticariu, L., Tan, W.C., Vijayvargiya, G.: An annotation management system for relational databases. VLDB J. **14**(4), 373–396 (2005)
18. Buneman, P., Chapman, A., Cheney, J.: Provenance man-agement in curated databases. In: SIGMOD 06. ACM, pp. 539–550 (2006)
19. Buneman, P., Khanna, S., Wang-Chiew, T.: Why and where: a characterization of data provenance. In: Database Theory ICDT 2001, Ser. Lecture Notes in Computer Science, vol. 1973, pp. 316–330. Springer, Heidelberg (2001)
20. Davidson, S.B., Freire, J.: Provenance and scientific workflows: Challenges and opportunities. In: SIGMOD 08. ACM, pp. 1345–1350 (2008)
21. Amsterdamer, Y., Davidson, S.B., Deutch, D., Milo, T., Stoyanovich, J., Tannen, V.: Putting lipstick on pig: enabling database-style workflow provenance. Proc. VLDB Endow. **5**(4), 346–357 (2011)
22. Wikipedia Page history. https://en.wikipedia.org/wiki/Help:Page_history/
23. XWIKI Homepage. http://www.xwiki.org/xwiki/bin/view/Main/
24. Eltabakh, M.Y., Ouzzani, M., Aref, W.G.: DBMS—a database management system for biological data. CIDR **2007**, 196–206 (2007)
25. Eltabakh, M., Aref, W.G., Elmagarmid, A., Ouzzani, M.: Handson db: managing data dependencies involving human actions. In: IEEE TKDE, no. PrePrints, p. 1 (2013)
26. Apache hadoop. http://hadoop.apache.org/
27. Hadoop distributed file system. http://hadoop.apache.org/docs/r2.3.0/hadoop-project-dist/hadoop-hdfs/HdfsUserGuide.html
28. Apache zookeeper. http://zookeeper.apache.org/
29. Bernstein, P.A., Goodman, N.: Concurrency control in distributed database systems. ACM Comput. Surv. **13**(2), 185–221 (1981)
30. Apache hbase coprocessors. https://blogs.apache.org/hbase/entry/coprocessor_introduction
31. Protocol buffers. https://developers.google.com/protocol-buffers/
32. Wikimedia Downloads. https://dumps.wikimedia.org/
33. Wikipedia: Usability. https://en.wikipedia.org/wiki/Usability
34. Benson, D.A., Cavanaugh, M., Clark, K., Karsch-Mizrachi, I., Lipman, D.J., Ostell, J., Sayers, E.W.: GenBank. Nucleic Acids Res. **41**(D1), D36–D42 (2012)