Record Locking Methods in the System for Managing Academic Records and Transcripts

Lucas Kushner

December 5, 2013

Abstract

ToDo

- Change all timestamp to lock_time

- Change all recordId to key

- Change all userId to locker

- Update section 3 to reflect the Unified lock table

- Future of SMART

Acknowledgements

First and foremost I would like to thank Dr. Alyce Brady for all of her support and efforts of the SMART project. it is only through her vision and work that this project is able to exist and progress, and I am extremely grateful to have the opportunity to work toward such a meaningful and profound goal. Additionally, I would like to thank Dr. Pam Cutter for her assistance and feedback throughout the various stages of writing this paper was greatly appreciated.

Also, I would like to thank Kyle Sunden and Ashton Galloway, who had previously worked on the RAMP and SMART projects, and whose assistance helped me through various difficulties during setup and installation of SMART and its associated tools.

Contents

1 Introduction

1.1 Background

After a destructive 11-year civil war that ended in 2001, the nation of Sierra Leone remains in the process of rebuilding its governmental and educational infrastructure. The University of Sierra Leone and Njala University, the country's two major universities, play an important role in providing education, supporting research, and producing alumni who will be instrumental in rebuilding their nation. As both universities continue to develop and expand, using accessible academic and administrative records will be a critical component in institutional capacity building. Currently, these institutions maintain their records purely on paper. By managing computerized records, they will be able to further establish and communicate educational requirements, monitor and support students' educational progress, forecast institutional needs, and assess and document progress toward institutional goals.

Although electronic records are a high priority for these universities, currently no free or Open Source project exists which is suitably equipped to meet their needs. To address this issue, we have proposed combining two existing projects, the Record and Activity Management Program (RAMP), and System for Managing Academic Records and Transcripts (SMART) to produce a dynamic and adaptable system for electronic record keeping.

RAMP provides a framework which makes the creation and implementa-

tion of dynamic websites much more flexible, by working off of a dynamically defined database structure. In essence, RAMP allows the creation and use of tables and activities within a web application, without explicitly identifying the tables in the code itself. This is instead done by defining the desired structure of data to be used in a series of configuration files. RAMP allows users to view these tables (or a selected set of columns), search for entries based on certain field values, modify, add and delete records, all in a fashion which is designed to be generic and easily adaptable to different database schemas.

SMART was originally a system used by the Registrar's Office at Kalamazoo College for managing academic records and electronic course registration and placement. It consists of a database structure which has been tuned to the needs of an academic institution, with facilities for organizing curriculum records (including courses of study, courses or modules, and individual offerings), instructor records, and student records.

Each of these two projects is meritable in its own right, however by combining the two we hope to create a new kind of record management system that will be fully functional for academic institutions, yet flexible enough to adapt to the different internal structuring of data within any given college or university. Currently, these projects are being combined to create a new SMART system which runs using RAMP as its framework for managing and displaying data. By taking this approach, we can take advantage of the deeply defined database structure utilized in the previous SMART system,

while running atop the flexibly stable RAMP system for managing the data. In doing so, we aim to create a free, robust, and dynamic records management and administration infrastructure for these two universities. Ideally, the final product will also be one which could be easily adapted to different schools and systems with minimal changes to code, providing a solution that any variety of universities would be able to implement in the future.

1.2 Structure of SMART[1]

Several of the most popular web technologies are integrated throughout SMART. At the most basic level, SMART operates on top of a combination of an Apache web server, MySQL database, and the PHP server-side scripting language. This is a very common web-development setup typically referred to as AMP. However, on top of these basic tools, the core functionality and implementation of SMART is made possible through the use of a PHP library known as Zend Framework.

Zend is a popular, modular implementation of a Model-View-Controller (MVC) framework used for powering web applications written in PHP. The MVC application development paradigm outlines a flexible yet powerful structure for the creation of complex applications, and so was perfectly suited for the SMART project. The basic principle is concerned with the separation of responsibilities within the program. The Model component is responsible for

[1]From this point onward, the term "SMART" will be used in reference to the newer system which is a combination of the RAMP and SMART systems as described previously.

managing all direct interactions with the data, outlining its structure, and providing the necessary interfaces and functions to interact with it. On the opposite end of the spectrum, the View components are purely responsible for what the user sees from the application, and handling all logic related to the presentation of data. The Controller is the bridge which connects the data and the user interface, defining and mediating interactions between the Model and View. Whenever data is updated in the View, the Model needs to know how to update the database, and contrastingly, when an update is made to the data the View needs to be notified to present the updated data. Managing these interactions is the core function of a Controller. By splitting up responsibilities in this fashion, the MVC framework keeps the structure of a project neatly organized while still fulfilling the requirements of a complex and dynamic application.

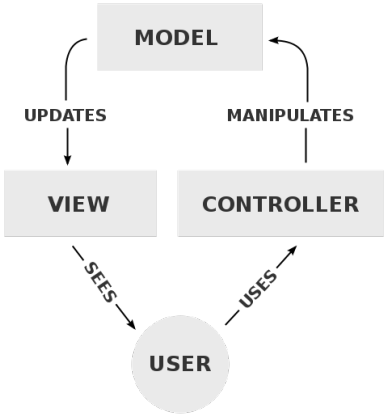

Figure 1: A simple diagram of a Model-View-Controller layout[2]

[2]Image retrieved from `http://en.wikipedia.org/wiki/File:MVC-Process.svg`

Along with the expansive and powerful set of application development tools provided by Zend Framework and the MVC paradigm, the robust flexibility of SMART is possible primarily due to two of its key features: Activities and Tables[3].

In SMART, Activities can be thought of as a way for the user to navigate through the program and find the data they are looking for. In essence, an Activity is a webpage which provides a submenu presenting a certain set of Tables that a user can view, gathered under a common heading. However, like much of SMART these Activities are defined not explicitly in code, but in separate configuration files, making them easy to rearrange, modify, and expand. In addition to redirecting users to specific Tables, there are a variety of other Activities that can be used to display information or fire specific actions.

Tables are at the heart of SMART's functionality. Without a clean and secure way of accessing and manipulating data, users would not be able to effectively and efficiently achieve their record management goals. In SMART all Tables are defined in configuration files known as Table Settings. These files outline which tables from the database are to be used, which columns should be shown, and specify the labels and formatting of each field. Additionally, Table Settings can outline connections between different database tables, and consolidate data from multiple sources into one Table that is

[3]Throughout this paper, we will be using the capitalized word "Table" in reference to Tables that are defined in SMART by Table Settings, whereas the lowercase word "table" will be used in reference to tables in the database.

displayed to the user, allowing for the construction of easy-to-use structures that abstract away the underlying complexity of the database.

The formatting requirements of Table Setting files are designed to be flexible, such that complicated data structuring is possible. The relaionship between a database table and Table Setting file is actually defined such that multiple Table Settings can access a particular table, and multiple tables can be consolidated and referenced in one particular Table Setting. This means that this relationship is both one-to-many, and many-to-one, functionally creating a many-to-many relation which can be applied in any number of ways, depending on the specific needs of the system.

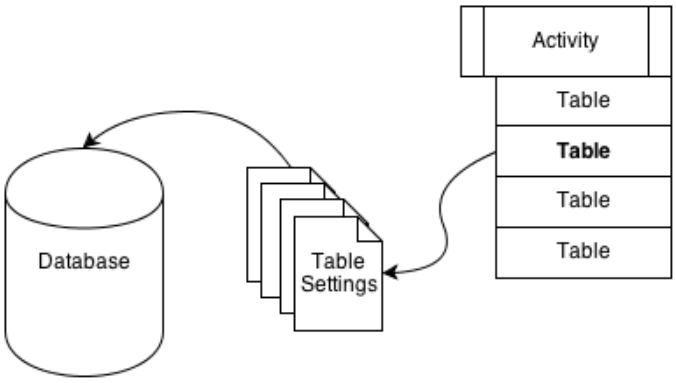

Figure 2: The flow of control in accessing the database in SMART. Activities define a set of tables with which the user can interact. Each Table corresponds to a specific Table Setting file which outlines the relationships to database tables. All definition of database structure is done in the Table Settings, abstracting it away from the program itself.

1.3 Record Locking

In a system as expansive and dynamic as SMART many complexities can lead to issues. One such problem that lies at the core of the system is dealing with multiple users attempting to edit data simultaneously, often resulting in an overwrite. The issue of such an overwrite occurs when several users try to access the same piece of data at once. For example, imagine that at the beginning of a new academic year, an employee of the registrar is editing the the address information of a particular student. At the same time, another employee attempts to alter the advising information of that same student. With both employees editing the same data, only the last change made will be saved, overwriting the previous change, meaning that either the address or advisor will be saved, but not both. Such corruption of data could have dire consequences depending on the circumstances.

One standard solution to deal with this issue is known as "record locking". Record locking is a way of managing data to assure its integrity, and prevent data from accidentally being overwritten. Several techniques have been developed for "locking" the data, all aiming to prevent multiple users from simultaneously altering it. While a variety of different database and application level implementations exist, the realm of possible solutions fall into two main categories: optimistic and pessimistic locking.

With optimistic locking, multiple users are allowed to continue editing the data simultaneously. The data is only locked when a specific user actually writes their changes to the database. In this situation, whenever a user

attempts to write their data, a version number is checked to verify that no new data is present. Only then will the user be notified that a conflict has occurred, at which point they must edit their data again and retry. This method allows several users to continue editing data while still locking the data when it is written. However, the last minute notification of data changes could prove frustrating or confusing to the user.

On the other hand, the pessimistic locking strategy uses what is called a "lock table" to lock specific records, and keep track of who is editing a given record. This table includes the unique ID of the record, the user that is editing it, and the time at which the user gained access to the record. Whenever a user attempts to edit a record, the lock table is checked to see if the record in question is already being edited. In that case, the user is then immediately notified that a change is in progress, and must wait until that change is submitted before attempting to make their edits. Often, the user will receive a notification once the data has been unlocked, and the page will automatically display the editable data. Although this system is much stricter in its protection of the data, this can lead to an additional problem. In the case known as "deadlock", two locked processes are waiting for each other to finish before they are able to unlock. Because of this possibility, careful database and software design is necessary when choosing to lock data in a pessimistic fashion.

2 Record Locking in SMART

2.1 Challenges and Considerations

In approaching the issue of record locking, SMART presents an interesting challenge to finding an appropriate implementation. One significant complexity in the problem, is that a "Table" in SMART often consists of data that is pulled from several different tables in the database. For example, when a user attempts to edit the data for a specific course offering, they may see information about the course itself, about the professor of the course, and about the current enrollment. While it makes sense to show all of this information on one page in SMART, the offering's information comes from a Courses table, while the professor and student information comes from a separate Person table. Although the application conveniently abstracts away the underlying database structure, the appropriate entries in both tables must be locked in order to ensure that all applicable data is protected from overwrites.

Some of these tables, such as the Person and Courses, have a high-frequency usage throughout various Activities in SMART, and therefore strict locking of that data is critical to maintaining its integrity. However, other tables may be accessed much less frequently, and therefore implementing a full-featured record locking system could contribute significant complexity to the structure of data and program execution, with little benefit in terms of realized record management. For example, as student information

and course enrollment changes, clearly the Student and Courses tables will be used frequently, and need locking mechanisms in place. However, a Terms table containing information about the start and end of academic terms will likely only be edited once or twice each year, and any mistaken overwrites would be non-detrimental and easily fixed. Therefore, creating a lock table and setting up the locking infrastructure for Terms is too much effort with not enough payback. Thinking this way about which tables to lock is necessary for producing an efficient and sensible implementation of record locking throughout SMART.

2.2 Design of Specific Implementation

Taking these various factors into account, we decided to implement a pessimistic locking system at the record level. In making this decision there were several considerations that were addressed. Firstly, the information that is most frequently edited in an academic records system generally has a direct relationship to a student or professor's work, and so any mistaken changes to this data could have a significant and stressful impact. Additionally, this information is also used to generate legal or official documents, and so all steps must be taken to protect data from mistaken overwrites. Therefore, locking the data in a pessimistic fashion makes the most sense to minimize such mistakes.

Moreover, particularly for an academic institution, we cannot assume that an individual employee would contain enough information to appropriatley

handle conflicting data were we to implement an optimistic system. Since the chance of two people editting the same record at the same time is fairly low, we also face a small penalty for collissions in a pessimistic system. Considering the highly-sensitive nature of data, the difficulty of handling any conflicts that could arise, and the low penalty cost, a pessmistic approach for record locking was chosen.

In general, pessimistic systems tend to have some sort of mechanism for notifying a locked-out user when the desired resource has become available. While implementing this into SMART is definitely feasible, we have decided not to implement this feature, and instead simply notify a locked-out user to try again later. The reasoning behind this has to do with the working environment of a Registrar's office. Typically, a particular employee may have a long list of updates that need to be made to a certain set of records, involving many different students, faculty, and courses. If they were to run into a record that was locked, they would likely not wait around, and instead continue on to the next item on their list, retrying the locked record later on. With a workflow such as this, it would not be pragmatic to expend resources checking for records to unlock when the majority of the time the user will simply brush over locked data and move on, rendering the checking mechanism unnecessary.

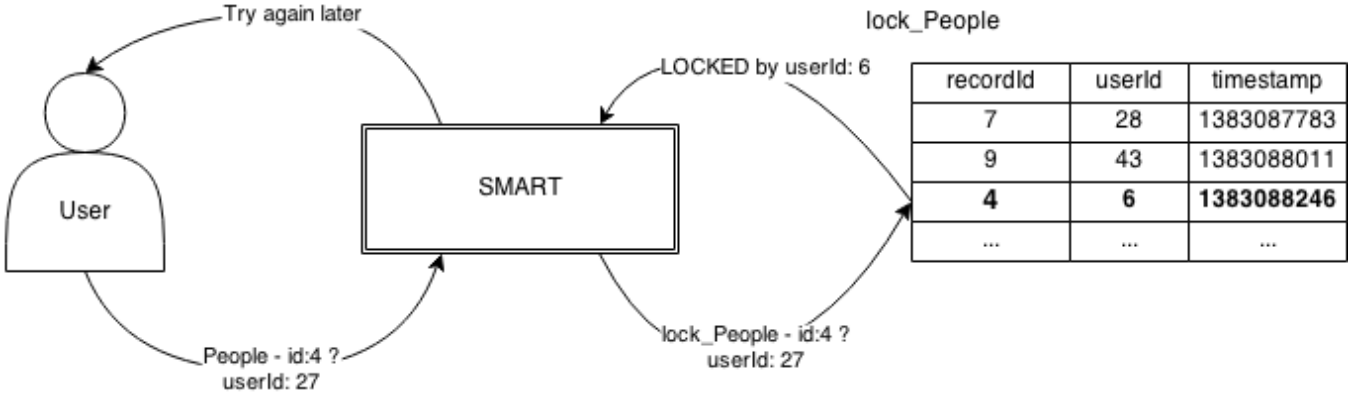

Figure 3: Design of initially proposed locking system. When the user requests a particular record, the record and user IDs are sent to SMART, which then checks the appropriate lock table and notifies the user if there is a lock. In this model, SMART is programmed with knowledge of the lock table structure in the database.

2.3 Definition of Locking Relationships

After coming up with an initial design we set out to begin implementation, and immediately ran into the issue of defining which lock tables in the database would correspond to different Tables in SMART. Originally, our design lent itself to explicitly including logic within the code for determining which lock table corresponded to a certain set of data. However, the power of SMART is that table relationships are not explicitly defined, but instead can be easily configured through the use of Table Settings. To follow this paradigm, it is necessary to define locking relationships in a dynamic fashion, such that SMART could be implemented on any database structure without having to modify the code itself.

One approach would be to define specific locks for each field of a Table in the Table Setting file. This is the approach used by SMART for mapping

specific fields that the user sees to the corresponding database table. The issue arises when considering the possibility of an incorrectly defined Table Setting. Imagine that a user creates a new table setting but neglects to set up the locking mechanisms. If such mechanisms were purely defined by the files, the possibility exists for a user to create a Table without any locking, exposing a significant vulnerability in a fundamental part of the system.

Our solution to this problem is to instead create a new database table which maintains the information about mapping specific data to its appropriate lock table. Currently, SMART assumes the existence of only two specific tables: `ramp_auth_auths`, and `ramp_auth_users`. These tables are responsible for logic related to user authentication and permissions, while all other aspects of the application are flexibly defined in files. By adding another program-specified table (ie. `ramp_locks`) and using it to manage locking information, we can force specific locking relationships on the database level, as opposed to the settings level. Doing so means that locks are enforced completely regardless of any specific Table Setting, thereby making it impossible for a user to accidentally create unlocked access to the data. The downside to this approach is that it creates more work for the database administrator during the initial setup of the database. However, the extra effort is necessary to ensure a secure and stable record locking mechanism

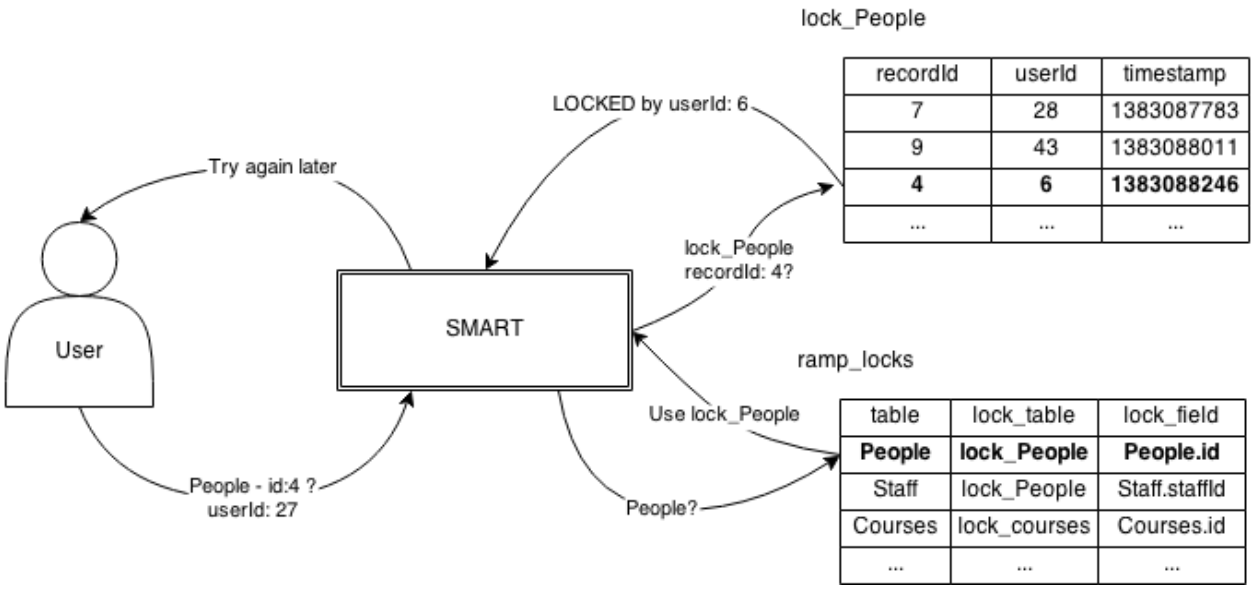

Figure 4: Flow of control for the updated locking solution in SMART. We see that the program checks the `ramp_locks` table for the defition of locking relatoinships, and uses those values to enforce locking at the database level.

As shown in the figure above, the proposed `ramp_locks` table contains three columns: `table`, `lock_table`, and `lock_field`. As one might expect, `table` and `lock_table` specify the name of the table in the database and the corresponding `lock_table` which will manage its locks. Beyond these basic record locking components, the `lock_field` column gives this particular locking mechanism enhanced flexibility. Ultimately, the entry in `lock_field` will correspond to the `recordId` column of the `lock_table`. Because any column in the table can be used in `lock_field`, this allows for both primary and foreign keys to be used as a locking trigger, which in turn creates the possibility for powerful locking schemes on tables that pull data from multiple sources. In the example shown in the figure, when a user begins

editing a particular entry in the Staff table, SMART will use the value of `Staff.staffId` as the key in `lock_person`. Because of this, if another user tries to edit the Person with that ID from any other part of SMART, they will be notified that the resource is locked. In this way, record locks are able to be persistent regardless of what particular Activity they originate from. This feature of our record locking system is the critical component in establishing a technique that can handle the requirements of SMART.

2.4 A Unified Lock Table

The above management of locking relationships allows locking to be forced on the database level. However, this system still involves many necessary alterations to the database for any particular system, specifically, adding lock tables for each of the appropriate database tables. If at all possible, SMART should operate in such a way that minimal changes are necessary when adapting to a new system. To address this issue, we will continue using the same method for defining locking relationships in the `ramp_locks` table, but unify the lock tables into one more required table (ie. `ramp_lock_locks`), which will store all existing locks.

This master lock table will contain four columns: `recordId`, `userId`, `table` and `timestamp`. The `recordId`, `userId` and `timestamp` will operate in the same fashion that has been described previously. The `table` field is the critical component which makes unifying the lock tables feasible. This will specify which table of the database holds the locked record, making

it possible to differentiate different records from different tables that might happen to have the same `recordId`. By being able to distringuish between records in this way the system of lock tables that was previously required can be condensed down to one.

Additionally, because SMART will assume the existence of `ramp_lock_locks`, less configuration is needed by an institution trying to adopt the SMART system. All that is required is to define the connections in the `ramp_locks` table, and then sturdy, enforced, record-level locking will be enforced throughout the application.

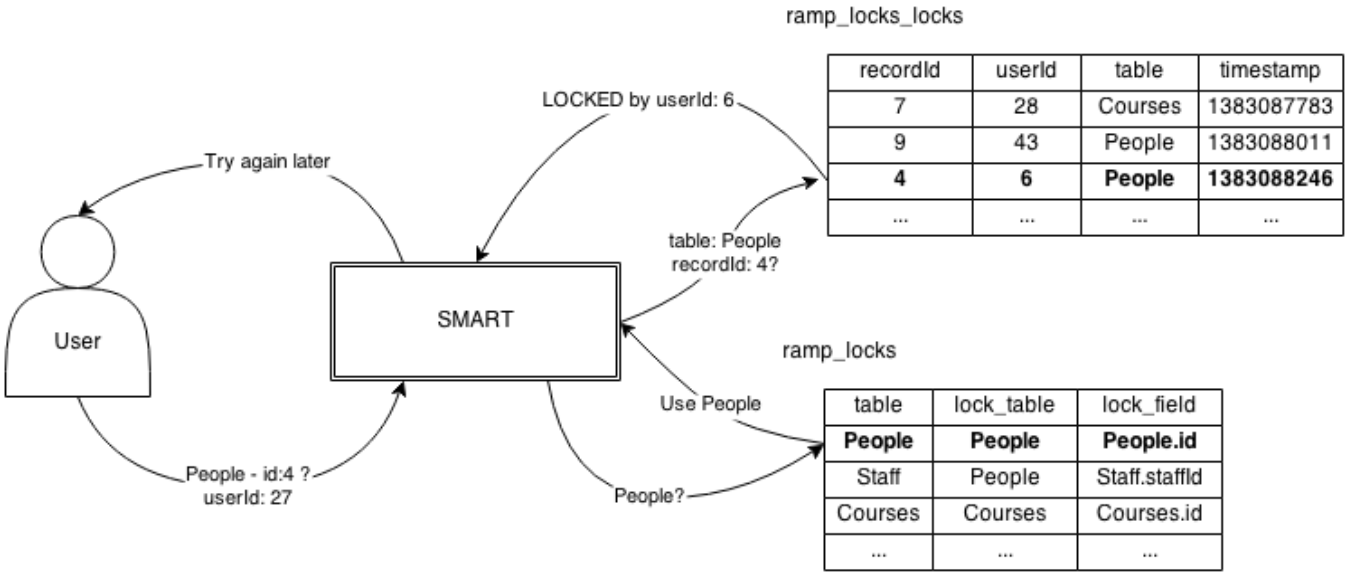

Figure 5: Record locking in SMART using one, unified lock table for managing all locks. The `ramp_lock_locks` table will use the `lock_table` value from `ramp_locks` as the value of the `table` column, making it possible to distinguish records which belong to different tables in the database.

3 Alternative Implementations and Solutions

Throughout this paper, the majority of our exploration of record locking has been specifically attuned to the needs and nature of SMART. However, the development of our implementation and its various features and components involved in-depth analysis of this specific system, and the consideration of a variety of possibilities. Next, we will discuss different systems for record locking, including potential features for other pessimistic systems, and optimistic alternatives.

3.1 Pessimistic Systems

One such additional feature for a pessimistic system would be a mechanism for notifying a locked-out user when the desired resource has become available, as described previously. Such a notification typically involves an Asynchronous Javascript and XML (AJAX) function call that pings the server at regular intervals to check if the data is still locked. Once the record is unlocked, the user is then redirected to the editing page, which has been populated with the recently updated information. In such a system as this, it is also necessary to implement some form of access queue, to handle the scenario in which several users attempt to access the same page while a record is being edited. This access queue could be implemented relatively easily by performing additional logic when checking a record's availability, as demonstrated in the code fragment below.

```
function checkLock($recordId, $userId)
    // Access queue logic is implemented by ordering
    // by time and selecting only the most recent lock
    $lockRow = SELECT * FROM locks
               WHERE recordId=$recordId
               ORDER BY lock_time ASC LIMIT 1;
    if ($lockRow['userId'] == $userId)
        return true;
    else
        return false;
```

As opposed to our method of managing lock relationships using a separate `ramp_locks` table in addition to a master lock table, it would be possible for the database to be setup in such a way that every table had its own lock table. In this case, the software would simply need knowledge of which tables were being accessed to immediately know which lock tables to use. For systems where there are numerous tables, many of which contain sensitive and very frequently accessed information, a solution such as this may be the most appropriate. While this layout contains more overhead during setup, and explicit programmatic knowledge of the database schema, it could be cleaner and more efficient, based on the particular system's needs.

3.2 Optimistic Systems

Alternatively, depending on the structure and purpose of the application an optimistic locking system may make more sense. In SMART, the sensitivity of data and workflow of academic records management implied that the use of pessimistic record locking was better suited for the project. Scenarios in which an optimistic solution would be better suited include cases where it is unviable to make all users wait to edit a certain record, or where many unrelated users are accessing data from different locations. A perfect example of this kind of system is a large, Wiki-style website, such as Wikipedia. Also, optimistic solutions are preferrable in cases when database reads are much more frequent than writes, because it allows users to read data frequently, even if another user is editing it.

In optimistic systems such as these, typically the program must manage version numbers on all edits to records. When a user attempts to write data, the original version number is included with the submitted data and checked against the version number stored in the lock table. If there is a versioning conflict (ie. if another user has submitted new data during the interval that the initial user was editing), the user is then notified and must handle any conflicting data, as demonstrated in the following snippet.

```
function submitData($recordId, $userId, $data)
    $dataRow = SELECT * FROM data_table
               WHERE recordId=$recordId;
    if ($dataRow['version'] == $data['version'])
        UPDATE data_table
        SET version=$data['version'], ...
        WHERE recordId=$recordId
    else
        // If there was a conflict, send them to a
        // page to sort out conflicting data
        redirectToConflictPage($recordId, $data)
```

One of the primary advantages present in using an optimistic system as opposed to a pessimistic one is that all users are able to edit data simultaneously, only being locked out at the time of submission. If good mechanisms are in place for managing conflicts, optimistic systems have the potential to allow for much more efficient editing by many users. However, this method puts more responsibility on the user to make the appropriate decisions when dealing with conflicts and merging data. Furthermore, if the conflicts are significant, than it may not be clear which version is correct. While it creates the opportunity for concurrent data modification, optimistic record locking also opens up the possibility of significant frustrations for the end user.

4 Conclusion

4.1 Summary

Throughout this paper, we have explored various potential solutions to the problem of record locking. In the two major categories of solutions - pessimistic and optimistic locking - there are a variety of different approaches that may be suited to different kinds of software systems. Additionally, for any given record locking mechanism there are a several features and alterations that can be made to fine-tune the design pattern to a systems needs, as well as increase the usability and maintainability of whatever locking system is to be implemented.

By considering the merits and disadvantages of both pessimistic and optimistic systems, it was concluded that a pessimistic approach was the most appropriate for meeting the needs of SMART. In addition to the traditional methodology of pessimistic record locking, it was necessary for us to include an additional layer of logic to preserve the dynamically defined table structure that is characteristic to SMART. To address this, we propose defining database locking relations in a required table called `ramp_locks`, thereby enforcing record locking on a database level. While such a system adds slightly more overhead for an institution setting up SMART, it creates secure locking relationships that can maintain the integrity of data throughout the application.

4.2 The Future of SMART

By outlining a robust record locking system, we hope to provide the framework for a vital data management component of the SMART system. However, much still remains to be done before SMART is ready to be used in full production. Included among the remaining key functionality requriements are database audit trails, more advanced reporting, and support for searching within user-specified date ranges. Although in its current form SMART can be used by an institution to begin the arduous process of data entry, it cannot be used as a full-featured record management system until these and more components are complete.

As the functionality and shape of SMART evolves throughout its development, the aim of creating a robust, Open Source academic records management solution remains unchanged. Ultimately, we hope that SMART will be a resource that any institution can adopt and adapt to provide reliable electronic records management and administration. While one may consider such tools a basic need of any college or university, in reality paper-based insitutions still exist. At places like Njala University and The University of Sierra Leone, access to a system such as SMART not only frees a signifcant amount of resources, but allows these universities to better educate the next generation of innovators, researchers, and leaders of their recovering nation.

Appendix A System Specifications

For development purposes, our backend for the SMART system used a Cross-platform Apache, MySQL and PHP distribution known as XAMPP. This package allows for the easy installation of these various components, as well as providing default configurations for easy use. Additionally, we used Zend Framework 1.12 library for the backbone of the application. Although Zend is currently in version 2, we continued using 1.12 because much of SMART had been written in years prior to the new version, and a variety of significant changes were made to the framework that could not be easily updated within the existing project.

Further investigation is still needed to determine the minimum possible hardware and software system requirements for SMART. Most tests and development discussed in this paper have taken place on a Linux system, with software specifications as outlined below:

Machine Specifications
Linux Mint 14 Nadia
Intel Core i5 M430 @ 2.27 GHz

Development Software
XAMPP for Linux (1.8.1)
Apache (2.4.3)
MySQL (5.5.27)
PHP (5.4.7)
phpMyAdmin (3.5.2.2)
Zend Framework (1.12.3)
VIM - Vi IMproved (7.4)