Research Prospectus

for

ScaDaVer – A Framework for Schema and Data Versioning in OLTP Databases

Robert L. Wall
Department of Computer Science
Montana State University – Bozeman

May 10, 2011

Abstract

Managing and accommodating the evolution of database schema (the metadata describing the structure of a database) poses a number of interesting problems. Over time, the characteristics of the data stored in a database inevitably change. The database schema must be adjusted to capture these changes, and it may be necessary to transform the data already stored in the database to reflect these changes. Some of these problems are particularly acute in Online Transaction Processing (OLTP) databases that serve as the data store for large, extremely active data processing systems, especially in systems that use a Software as a Service (SaaS) delivery model. We propose a framework to be incorporated into such databases to efficiently manage the inevitable schema evolution. This framework is based on common practices used to manage source code changes in software development. It allows administrators and users of the database to create sandboxes in which changes to the database are isolated from the main database and from other sandboxes. Schema versioning techniques are used to isolate schema changes made within sandboxes and to allow queries executed in a sandbox to retrieve data from the main database without transforming all data in the system to conform to the sandbox's schema. The framework also includes data versioning mechanisms to maintain isolation of data that is added, updated, or deleted in the sandbox. A merge facility allows changes made in a sandbox to be integrated into the main database while it is online, with minimal disruption to ongoing transaction processing. These two versioning techniques together with the merge facility create a database infrastructure that will significantly reduce the time, manpower, opportunity for errors, storage capacity, and infrastructure required to perform development, maintenance, and testing of schema and data changes against a high-volume online database.

Table of Contents

Table of Tables

Table of Figures

1. Introduction

The database schema – the metadata defining the organization of a database's contents – is a key component of any software system that uses a database. And in any such system of even moderate complexity, this database schema will need to be modified after the system has been developed and an instance of the database has been created using the initial schema. There has been significant research on mechanisms for handling the evolution of database schemas over time, including techniques for simplifying the transition between versions, as well as methods for actually versioning the schema so that queries against a particular version of a database schema can retrieve data that was created using other versions. However, most of this research has focused on data warehouses and OLAP (online analytical processing) databases, temporal relational databases, XML data stores, and object-oriented databases. There is very little published work on handling schema evolution in a relational OLTP (Online Transaction Processing) database.

Many modern Web-based Information Systems are built around a relational OLTP database. These systems characteristically store and retrieve huge volumes of data; consider, for example, systems like Wikipedia, Facebook, and Twitter. Also consider how rapidly some of these systems must change to accommodate new user expectations and customer trends. A common aspect of many of these changes is new data that must be captured and used by the system and modifications to existing data maintained by the system – changes that require modifications to the database schema.

Complicated software systems such as customer relationship management (CRM), enterprise resource planning (ERP), and e-commerce systems are also often implemented using an OLTP database as a persistent store of the system state. This data store is a critical part of the system's operation, and for these complicated systems, the database is often extremely large and has a complicated schema. Unlike many Web-based Information Systems, these systems typically have significantly more complicated data processing demands than simple insertion and retrieval; complex business processes may transform the data, it may be involved in integrations to and from other data processing systems, and complicated analytics calculations might be performed. These complex data flows often result in tighter coupling between system software and the database. Because of this coupling, adding application functionality frequently requires schema changes, and schema changes can affect many different parts of the software.

With such large systems, enhancement requests for additional system functionality or performance improvements are inevitable. The volume of demands for system changes rises and system complexity correspondingly increases with the expanding availability of data interchange mechanisms between systems and with increasingly complicated business models that the systems must support. The issue is further exacerbated in systems that are delivered using a Software as a Service (SaaS) model; in this model, a service provider is usually supporting multiple instances of the system for different customers, each of whom may have a different set of business requirements. This arrangement is typically referred to as a *multi-tenant* model [**1**], [**2**]. To compound the problem, customers typically have an expectation of more responsive incorporation of change requests and of easier customization in Web-based applications [**3**]. Some requested changes will likely be incorporated into the base software system as features available to all SaaS clients, while other changes will be customizations of the system for individual customers.

1.1. Expected Contributions

We have worked to characterize the schema evolution in a large commercial software system implemented using a SaaS model and we have compared this with the evolution of another large open source system, MediaWiki, the web portal software built to run Wikipedia. We have identified problems that schema evolution causes in this system for which there are currently no suitable resolutions, including excessive overhead to manage testing and deployment of software upgrades and difficulties in developing and testing customizations on systems in operation. We plan to research, design, and implement a framework to address these problems. This framework should significantly reduce the amount of effort required to modify the schema of a large OLTP database, and it will provide the ability to work in a sandbox where schema and data changes can be tested against the data in the database without interfering with the normal operations of the production system. It will also address the severe operational and performance problems created when making several types of schema changes to an online MySQL database.

In order to evaluate design and implementation alternatives in a structured, reproducible way, and to allow for comparison to other systems that might address some of these problems in different ways, we will create a pair of benchmark data sets; the first will be built on an abstract schema that will provide a clean separation of data types and will allow the creation of straight-forward examples, while the second will be representative of the kind of large database-centric systems in which high-volume and complex data transactions occur. These data sets will be made publicly available and will be used for all evaluations and measurements of the framework.

1.2. Motivation

We have been allowed to perform an evaluation of a commercial software system which is provided to its customers using a Software as a Service (SaaS) model, where the software vendor hosts the application and the database for each customer on servers that the vendor owns and operates. (In the remainder of this paper, we refer to this commercial system as the "SaaS system".) In the introduction to their paper on techniques for providing flexible schemas for SaaS systems, Aulbach et al. [**2**] describe a number of different database architectures that can be used to support the multi-tenant SaaS requirements. They divide these techniques into those in which the database "owns" the schema, meaning that each customer's database structure is defined explicitly in Data Definition Language (DDL), and those in which the application "owns" the schema, meaning that the database structure is mapped onto a set of generic storage structures and the application manages the mapping. The multi-tenant model used by the SaaS system we studied utilizes the *Private Table* approach, a technique in which the database owns the schema, each customer has its own instance of the database schema, and the schema is customized as required for that customer. This approach offers good performance, allows for easy interpretation of the data stored in a customer's schema, and provides isolation of individual tenants' data, but it poses some additional challenges in handling schema evolution.

The SaaS software vendor has several thousand customers, and thus manages thousands of instances of the product's database schema. In a recent study of this schema, we found over 300 tables, containing over 3,000 columns [**4**]. The table count and column count had increased by nearly 150% over a four year period. Compare this to the database schema that is used by Wikipedia; as of 2008, it included 34 tables with 242 columns [**3**]. Customers are able to customize their individual instances of the system in different ways, including by adding new columns to tables that are provided as a standard part of the product, and by adding new tables to their schemas. These customized columns and tables can be subsequently modified or removed, but the standard database schema provided with the product cannot be altered by the customer. In a recent study of nearly 2,000 of the vendor's customer instances, we found that 75% of the schemas had columns added to the product tables, and almost 15% had added tables.

Customers can also customize the application, adding new scripts, reports, and user interfaces that interact with the database. This creates a complex interaction between the product, the database schema, and the customer's modifications to the product.

The currently accepted industry practice for handling database and application upgrades is to make a copy of the database, update the application software, apply any schema changes, test the application, resolve any problems with the database and applications, and when everything has been updated and confirmed to be functional, disable the production system, repeat the schema modifications on the main database, deploy the new and updated application software to the production environment, and re-enable the system (e.g. see [**5**], [**6**], [**7**]). Obviously, this introduces a significant amount of overhead if it must be done for hundreds or thousands of separate instances of the application and schema, with varying levels of complexity and different schedules for installing the upgrades. This approach has several drawbacks, including the following:

- *Prohibitive cost.* The cost in resources (data storage, network bandwidth) and time (system administrator setup, data transmission and processing time) to create a copy of the database can be significant. This copy will most likely reside on a separate database server, so the data must be transported between servers; there may be issues with obtaining a read-consistent copy of the database without disrupting the operation of the production database. This database server must be sized appropriately to handle a full copy of the production data.
- *No forward path from production database.* If data is changed in the production database after the copy is made, the changes are not made on the copy in the development system. Depending on the scope and nature of the changes, this could impact development and testing. For instance, suppose the database is used to store report definitions and user interface (UI) layouts; if a new report is added to the system after the database is copied and this report references database schema that is changed, test suites on the development system will not detect the problem.
- *No easy reverse path back to production database.* Some data and schema changes performed on the copy must be migrated back to the production database after testing is complete. This can be a very complicated process, depending on the changes; some data will inevitably be created specifically for testing purposes and cannot be copied back wholesale. The data that must be migrated back must be extracted from the test system and applied to the production database; this is a very error-prone process.
- *Complicated merge process.* There can be complications with data that must be copied back – tables with auto-incrementing key fields will have conflicting key values if rows were added to the table in both production and the development copy. Other constraints might be violated when trying to merge rows from two instances of a table. Other changes may require a more localized merge with the production data – suppose, for example, that updates were made to different columns of the same row on the development system and the production system; resolving inconsistencies, removing duplication, etc. is time-consuming and complicated.

In addition to problems installing, testing, and deploying new releases of the software, similar difficulties are encountered in providing the ability for customers to make changes to customize their instances of the system. The same procedure of making a copy of the application and database can be used, allowing the customer to make changes to the schema, metadata, and application scripts, test, and to then migrate the changes back to the production database. As customers start making more customizations, this has the potential of doubling the number of copies of the systems that must be created and managed.

In spite of these difficulties, it is very beneficial to be able to test changes against production data to ensure correct operation. In addition to creating a platform for developing and testing upgrades to the base system, it would be beneficial to provide an environment in which customers can easily make customizations of individual instances of the service. This could be used by a SaaS customer, or a consulting teams operating on the customer's behalf, to develop, test, and deploy changes to its own instance of the service. Empirical studies of the SaaS system we examined show that customization of the base database schema is a common occurrence, and most of these schema changes will also be accompanied by other system customizations such as additional user interface components, new reports, additional integrations with other systems, etc. Making these changes on the production system is not feasible if the changes are even moderately complex.

We discovered that very little attention has been given to this problem. As we discuss in the next section, much of the research on handling schema changes in databases has focused on *Online Analytical Processing* (*OLAP*) and *temporal* databases. An OLAP database is a multidimensional database commonly used in data warehousing and data mining (i.e. one based on hyper dimensional *data cubes* rather than related tables). A temporal database is a relational database in which each record or fact is characterized by a time period over which the fact is true (e.g. databases of traffic flow, weather, credit card transactions). In both database types, records typically represent a snapshot in time. Once a snapshot has been taken, the structure of the data contained in that snapshot will not change. So as the characteristics of the data to be captured change over time, it is sufficient to define a new schema for the database and capture snapshots that conform to that schema (a process referred to as *schema versioning* [**8**]). The issue is thus one of evaluating queries that span a time period over which snapshot data was captured using more than one schema.

In contrast, an *Online Transaction Processing* (OLTP) database, often referred to as an *operational* database, is typically used to represent the current state of physical entities. Transactional data with temporal aspects may be stored in the database as well (e.g. purchase information or customer interaction logs), but the database itself is not predominantly temporal. Historically, the structure of this information has not been very dynamic; that is, the characteristics of the data stored in the operational database did not change quickly over time. When schema changes were necessary, the entire database would be transformed to conform to the new schema (a process known as *schema evolution* [**8**]); there was thus no need to handle multiple schema versions. Also, companies were typically managing one of these databases for their internal use, and had control over scheduling changes to the database and the application software. The process of upgrading the schema and the applications might be time and labor intensive, but it could be scheduled for off times to minimize disruption to users.

There are other significant differences between OLAP and temporal databases, which are primarily used for decision support and reporting, and OLTP databases, which are primarily used for system operations. The problem of handling schema changes in OLTP databases is in some ways more difficult. Reporting databases typically handle only queries (read access) after the data has been added to them, while OLTP databases are handling queries and also doing concurrent inserts and updates. Response time, throughput, and database availability requirements are typically higher for operational databases. The nature of the data in an operational database in many cases makes timely access to it more mission-critical. These differences may have also discouraged research on schema evolution in OLTP databases.

In recent years, there have been some changes in the demands placed on applications and databases in an OLTP environment. Systems must adapt more quickly now to changing demands for information. Database schemas must be much more dynamic, and handling these schema changes quickly and efficiently is crucial for many businesses. This is particularly true of SaaS providers; they are handling the data demands for hundreds or thousands of customers, and most of those customers cannot tolerate having the system taken offline for hours or days to implement an upgrade. The motivation to address the problem for OLTP-based systems has definitely increased.

In a SaaS environment built on a relational OLTP database, it would be advantageous to provide a *sandbox* environment for the service provider's customers. The term sandbox has the same meaning that it does when used in the context of software development systems – a separate environment created for an individual or group in which changes are isolated from the production system and from other sandboxes [**9**]. This environment must provide functionality similar enough to the production system that accurate tests can be run on the sandbox system. In a sandbox, a customer could implement schema changes, populate the modified schema with data, make additional customizations based on the new schema and data, and test the entire package of changes using a data set representative of the production data, without disrupting the production system. Once testing is complete, the set of changes could be moved from the sandbox into production as a package, increasing system stability and reducing errors in the production system following any customization efforts. Ideally, the system would provide multiple independent sandboxes, so more than one customization effort could be in progress concurrently.

MySQL is a very commonly used database engine for Web-based applications, including operations as large as Facebook and Google. A number of the schema modifications cannot be performed on a MySQL database while the database is online. This is primarily due to the fact that many of the modifications to be performed on existing tables, including adding or dropping columns or indices, require the table to be locked and copied to a new table, making the required changes on each row as it is migrated. For tables with hundreds of thousands or millions of rows, this can take a significant amount of time, and the only practical alternative is to take the system offline while the schema is being updated. A mechanism to perform online updates of the schema during this migration of schema changes to the production database would also be a significant benefit.

2. Background

The relational data model which is the foundation on which most modern relational databases are built was proposed by E. F. Codd in 1970 [**10**]. Codd et al. introduced the concept of Online Analytical Processing (OLAP) in 1981 [**11**], [**12**]. The issues of handling time in databases was an active research topic in the late 1970s and early 1980s; the term *temporal database* was introduced by Snodgrass and Ahn in 1985 [**13**] [**14**].

There has been a significant amount of research on the problem of dealing with schema changes over time (referred to as *schema evolution*) in the literature. Roddick presented an excellent survey of issues pertaining to schema versioning, schema evolution, and related topics in 1995 [**8**]; this included the following definitions, to which we alluded in the previous section:

> *Schema Evolution*. Schema evolution is accommodated when a database system facilitates the modification of the database schema without loss of existing data.
>
> *Schema Versioning*. Schema Versioning is accommodated when a database system allows the accessing of all data, both retrospectively and prospectively, through user definable version interfaces..

In the case of schema evolution, no support for previous schemata is required.

Rahm and Bernstein present an online bibliography of schema evolution issues in [**15**]; as of Apr. 19, 2011, this bibliography includes 619 publications, cross-referenced by a number of categories. Most of the following works that we cite were published within the last 5 years.

Most of the research has focused on data warehouses using OLAP (online analytical processing) databases and on temporal databases (e.g. [**16**], [**17**], [**18**]), XML data stores (e.g. [**19**]), and object-oriented databases (OODB) (e.g. [**20**], [**21**], [**22**], [**23**], [**24**], [**25**]). Ra et al. presented an early approach using views to manage schema evolution in an OODB [**26**]. Roddick et al. described a technique for schema selection in spatio-temporal databases, where in addition to the time dimension associated with data, there is a spatial dimension [**27**]. Wang and Zaniolo have researched the idea of presenting the history of schema changes in a relational database using an XML representation [**28**].

There has also been research on model management, matching schemas, and mapping between schemas that are not necessarily from the same type of data store (such as database schema to XML schema to Electronic Data Interchange (EDI) formats – e.g. [**29**]). There is very little published work on handling schema evolution in a relational OLTP (Online Transaction Processing) database.

Curino, Moon, Zaniolo, and others have investigated the handling of schema evolution in temporal relational databases at length. See [**18**], [**30**], [**31**], [**32**] for work published in the last three years by this group on mapping queries across schema versions in temporal databases. Independent groups have also studied this aspect of the schema versioning problem; e.g. De Castro et al. [**33**] and Wei and Elmasri [**34**].

In addition to their work on evolution and versioning in temporal databases, this group also developed more general tools to enable database administrators (DBAs) and developers to deal with schema evolution. These tools help to predict the impact of schema changes, rewrite queries for new versions of the schema, migrate data to a new schema version, and document the history of schema changes. See [**35**], [**36**], [**37**], and [**38**] for the work by Curino, Moon, Zaniolo, et al. on their PRISM and PRISM++ workbenches.

In 1993, Sjøberg conducted a studio of the evolution of the schema for a database that was used in a commercial health management system [**39**]. He studied the evolution of the schema over an eighteen-month period and attempted to quantify the changes. In 2008, Curino, Moon, and Zaniolo published the results of an extensive evaluation of the evolution of the schema of a large database, the one underlying the Wikipedia system [**3**]. They isolated a set of *Schema Modification Operators* (SMOs) that they encountered and used these operators in subsequent analysis. We have adopted the use of these operators for this research.

Aulbach et al. investigated alternative organizations for handling the database schema specifically for multi-tenant SaaS architectures in order to facilitate schema modifications in 2008 [**1**] and 2009 [**2**].

A valuable focus of research has been on handling schema mappings; that is, on techniques to specify how to map data from a source schema to a target schema. Of particular interest to us are the work by Bernstein [**40**] [**41**] and Fagin et al. [**42**], and the work on inverse schema mappings by Fagin [**43**] and Arenas et al. [**44**].

We are also very interested in the research on alternatives to the traditional row-based table storage used by most relational database engines. These techniques might prove useful for handling the data versioning issues. Stonebraker et al. proposed a column-based storage scheme that could offer much higher performance for certain read-dominant query mixes [**45**]. Abadi et al. compared the column-stores and traditional row-based data stores on a mix of query types [**46**]. Liu et al. specifically examined how using a column-oriented data store (CODS) could simplify schema evolution problems [**47**]. Plattner explored the use of column stores for combined OLAP and OLTP applications, examining the impact of modern processor and memory architectures on making a CODS practical for OLTP applications [**48**]. Grund et al. propose an interesting hybrid approach that attempts to dynamically partition the columns of a table into column-oriented stores containing multiple columns, based on the locality of reference of different columns in queries [**49**].

Haapasalo et al. have worked on database structures to allow efficient access to multiversion data in relational databases and to control concurrency across versions [**50**], [**51**], [**52**]. Much of their work is based on an earlier multi-version B+-tree proposed by Becker et al. [**53**]. These multiversion indices may also be useful to resolve data versioning issues between the trunk and branch schemas when indexed columns are updated in a branch.

Dumitraş and Narasimhan have done extensive research on handling schema modifications in online databases; they specifically examine the effects on system uptime of upgrading the Wikipedia database in [**54**] and [**55**] (both published in 2009).

Chatterjee et al. have proposed a system for loading a database schema with known test data on which to execute integration tests in [**56**]. This system utilizes long-running transactions to provide isolation of the test data. Unfortunately, long-running transaction support is not available in MySQL, and this technique only addresses problems of testing. Also, this method does not provide isolation for schema changes, and it does not provide any mechanism for merging any data changes back to the main schema; all data modifications would need to be either committed or rolled back as a group.

The current state of the art for performing schema updates to large tables in an online MySQL database is to build custom scripts to make a copy of the table, possibly on a replicated copy of the database, perform the modifications on that table, apply any changes made to the original table after the copy was taken (using some complicated set of triggers or other techniques to track the changes), then rename or drop the original table and rename the shadow table to the original name. See the report by Mark Callaghan at Facebook [**57**] (from 2010) and the online documentation for the *oak* toolkit [**58**] (from 2008) for two popular implementations. We anticipate using these as the starting point in our schema merge facility.

One aspect of this research will include estimating the impact of changes to the source code for the MySQL/MariaDB database engine. There is a large body of research on change impact analysis, but most of it is focused on object-oriented software systems, including papers by Lee et al. [**59**], [**60**], Ryder, Ren, and others [**61**], [**62**], and approaches using decomposition slices by Dor et al. [**63**] and Tonella [**64**]. A approach described by Herzig in 2010 using Transition Dependency Graphs (TDGs) looks promising [**65**]. The literature includes a paper by Xiao et al. from 2007 specifically about change impact analysis in service oriented architectures [**66**] and an interesting paper from 2009 on using Bayesian networks to estimate change impact by Abdi et al. [**67**].

3. Proposed Solution

Given the inevitable need for evolution of the database schemas in these systems, and the volume of changes that will be required of a SaaS provider, it is important to have processes which allow this evolution to be handled smoothly. We have found no existing solutions to help in this regard.

It is common practice for software engineers to follow a fairly standard development cycle when maintaining and enhancing a software code base. The code is maintained in a version control system, and when bug fixes or enhancements must be made, a branch of the code repository is created, a sandbox environment is built using that code, development and testing are done, changes to the code are applied to the code branch in the repository, and when the cycle is complete, the updated code is merged back into the main code base. Version control systems typically support multiple branches, so independent development projects can be undertaken concurrently. There are usually tools to help resolve conflicts when merging code. (See the history of version control by Ruparelia for a summary of this process [**9**].) In operational databases, particularly those used by SaaS providers, it would be useful to treat the database schema and data in a similar fashion, where a branch could be created for the schema and data, a sandbox could be created and associated with that branch, and all changes could be made and tested in the sandbox before being merged into the main production database

We propose a system to significantly streamline the feature deployment and customization processes, focusing on the goals and requirements in Table 1 (from [**4**]). Note that unlike much of the previous research on handling schema evolution, we are not concerned with handling the mapping of queries against the current version of a database schema over an arbitrary number of past versions of the schema (see [**16**] and [**32**] for example). Those are OLAP and decision support applications, and have been studied. We are instead focusing on OLTP systems and the difficulties of handling upcoming changes to the schema until they have been completed, tested, and finally integrated into the production database.

Table 1 – Versioning System Design Requirements

#	**Description**
1.	Allow the creation of sandboxes for development.
2.	Support the existence of multiple sandboxes concurrently.
3.	Eliminate the need to create copies of the database to create these sandboxes.
4.	Allow changes to the production data to be immediately visible in each sandbox.
5.	Isolate schema and data changes in sandboxes from each other and from the production database.
6.	Provide a reliable way to migrate schema and data changes made in a sandbox back to the production database.
7.	Allow other data modifications to be abandoned during this migration (e.g. rows added for testing).
8.	Reduce the downtime required to integrate the changes into the production system.

We will use some standard terminology borrowed from configuration management and version control systems to describe components of our versioning process. (See the history of version control by Ruparelia for a good summary of the terms and concepts related to version control systems used in software development [**9**]). The *trunk* version of a database will be the base version of the database schema and data against which production code operates. A *branch* is a distinct version of the schema and data that includes some set of *deltas*, or differences, from its parent version. A branch is created for each sandbox. We will discuss both *major* and *minor* versions of the schema; major versions will be identified using consecutive positive integer numbers, and minor versions will be identified using the major version from which they were created, a decimal point, and a positive integer number.

In order to limit complexity, the system will allow at most one major version to be created from the trunk. A new major version that is created from the trunk is referred to as the *head* version; this head version is intended for use by the SaaS provider, to install and test the next version of the system software. Zero or more minor versions can be created from the trunk, and if there is a head version, zero or more minor versions can be created from it. In order to limit the complexity of the system, branches cannot be created from a minor version. There are two differences between the head and other branches: additional branches can be created from the head, but not from minor version branches, and when the head is merged into the trunk, the head version becomes the trunk.

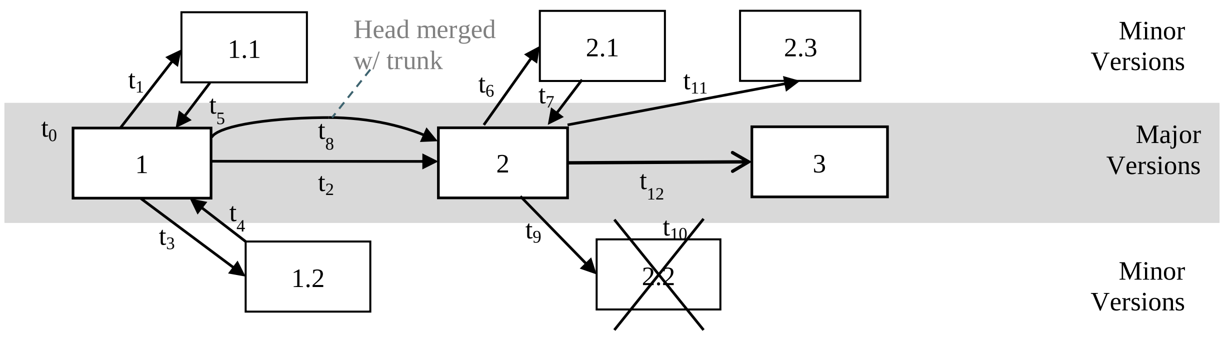

Figure 1 – Example of Version Management

Consider the example shown in Figure 1. The database schema is originally created at time t_0; this is the initial *trunk* version, 1. At a later time t_1, a branch with minor version 1.1 is created from the trunk. Subsequently, at time t_2, a new head version, 2, is created. At time t_3, a second branch version 1.2 is created from the trunk. At time t_4, branch 1.2 is merged back into the trunk. At time t_5, branch 1.1 is merged back into the trunk. At time t_6, branch 2.1 is created from the head version. At time t_7, branch 2.1 is merged back into the head version. At time t_8, the head is merged and becomes the trunk (i.e. the trunk version is now 2). At time t_9, branch 2.2 is created. At time t_{10}, that branch is discarded. At time t_{11}, branch 2.3 is created. At time t_{12}, a new head version, 3, is created.

In order to accommodate all the stated goals, we propose a system whereby the branch for every sandbox resides within the same logical database schema. The remainder of this section will discuss the components of the versioning system:

- Managing versions
- Versioning the database schema for a branch
- Versioning the data for a branch and storing data modifications in the branch
- Merging schema and data changes in a branch back to its source version

3.1. Managing Versions

Database client applications (hereafter referred to as *clients*) that connect to the database schema can specify a version; this version is an attribute of the session, so we plan to implement it as an extension to the standard SQL statement for setting session attributes; the syntax is given in Figure 9 in the appendix. Each database session always initially connects to the trunk version; this provides backward compatibility for applications that are not version-aware. Changing the version for a session affects any outstanding transaction using the same semantics as disconnecting the session. That is, the transaction will either be automatically committed or it will be rolled back, just as if the session was ended.

Clients are able to create new versions in conjunction with the creation of new sandboxes. As explained in the preceding section, in order to limit complexity, the system allows at most one major version, the *head* version, to be created from the trunk. Branches can be created from either the trunk or the head, but branches cannot be created from other minor versions. Again, this will be implemented as an SQL extension, with the syntax shown in Figure 10 in the appendix.

Every version except 1 (the initial database schema) has a *source version*. For minor versions, this is the corresponding major version; for instance, the source version for 2.1 is 2. For major versions, this is the previous major version; for instance, the source version for 2 is 1. Since branches cannot be created from minor versions, the source version will always be a major version.

When work in a sandbox is complete and the changes it contains are tested and ready to be incorporated into the system, the branch can be *merged* back into its source version. When a minor version is merged, if the source version is the trunk, this involves actually modifying the trunk tables to reflect the changes made in the branch. If the source version is the head, this involves combining changes made to the minor version with changes made to the head. A key aspect of the merge process is the resolution of conflicts between the branch and its source version. The merge process is described in more detail in section 5.7. After the version is merged, it is consolidated into its source version and is no longer available. When the head is merged, the trunk tables are similarly modified, and the trunk is moved forward to the new version.

Versions other than the trunk version can also be abandoned. In this case, any changes made in the branch are discarded, and it is no longer available. If the head version is dropped and there are branches created from it, those branches are also discarded. When minor versions are dropped, their version numbers are not reused. However, since there can only be one head version created from the trunk, the major version number is reused if the head version is dropped. This makes it simpler to identify the head version that will follow the trunk. The syntax of this SQL extension is given in Figure 11 in the appendix.

3.2. Schema Versioning

One of the primary constraints on the versioning system is that it must have minimal impact on the performance and stability of the production system, which is using the trunk version of the database. To facilitate this, the trunk version uses the standard database, with little or no modification. Database structures are unchanged, and changes to code paths in the database server are minimized when accessing the default trunk version. The impact will be measured using two different metrics, performance and stability. Measuring performance is relatively straightforward; a mix of queries will be designed and executed against our benchmark schemas on an unmodified version of the database engine, and the same set of queries will be executed on the same schemas in the trunk version of the modified database engine. Times will be measured over multiple runs. Differences in measured time between the unmodified database engine and the modified engine must be statistically insignificant.

Measuring stability will require measuring the potential impact of the code changes that are encountered in the code paths that are used while connected to the trunk version of the schema. We will first enumerate the code changes in these code paths and then find methods to measure impact. We plan to investigate some of the techniques described in the change impact literature listed at the end of section 2.

In their 2008 study of schema evolution in the MediaWiki database, Curino et al. introduced a set of schema modifications operators (SMOs). We have adopted this list and adjusted it to reflect the SMOs that we encountered in our study of the schema evolution of the Saas system. This resulted in omitting some operators and adding others. The initial list of SMOs we are considering is shown in Table 2.

Any SMOs are applied to a version of the schema in the form of Data Definition Language (DDL) statements in SQL. SMOs that are applied to the trunk version are processed normally. SMOs applied to a non-trunk version are stored separately, as *deltas* from the source version. That is, they are stored as the changes that must be applied to the source version to create the new version. These deltas will likely be stored as a parsed version of the DDL statements that were applied to change the schema.

Table 2 – Supported Schema Modification Operators (SMOs) with Mappings

SMO	**Mapping Operation**
CREATE TABLE*	Treat table as if it were empty in the trunk schema – queries return no rows
RENAME TABLE*	Replace new table name with old name
DROP TABLE*	No mapping – references to removed table in branch are errors
ADD COLUMN*	If column can be NULL, treat it as if all values are NULL in trunk schema. If column is NOT NULL in branch, must provide a default value.
RENAME COLUMN*	Replace new column name with old name
DROP COLUMN*	No mapping – references to removed column in branch are errors
ALTER COLUMN	Depends on the alteration
ADD INDEX	No mapping, although adding a unique index affects inserts, updates, and deletes on the branch
DROP INDEX	No mapping
ADD PK	Add primary key - same as adding a unique index
DROP PK	Drop primary key - same as dropping a unique index
ADD FK	Add foreign key - no mapping, although adding a foreign key introduces a referential integrity constraint that may affect inserts, updates, and deletes on the branch
DROP FK	Drop foreign key - no mapping

* denotes SMOs originally identified in [**3**].

In order to satisfy requirements 3, 4, and 5 in Table 1, most data is stored in the standard database tables associated with the trunk schema. When a SELECT query is submitted on a branch other than the trunk, it is first validated against the schema definitions associated with the branch. If the query is valid, it is mapped back to the trunk schema; that is, it is rewritten to be valid in the trunk by applying the inverse of the branch's SMOs and possibly the inverse of its source branch's SMOs (if it is a minor version created from the head branch). Again, unlike the techniques described by Moon et al. in [**18**], [**32**], and similar works, we are not concerned with mappings over multiple temporal snapshots of the data. We are focused on supporting up to two sets of changes from the trunk schema, and isolating these schema changes. These query mapping techniques only need to provide access to the trunk version of the tables.

Once the query has been executed, any deltas that affect the result set are applied to the data before it is returned. (Data that has been added or modified in the branch is also merged into the result set, as described in the next section). Because only one major version branch can be created from the trunk, and minor version branches can only be created from major versions, the versioning system will never need to apply more than two levels of mapping to resolve a query on a branch.

In order to support this mapping, the set of SMOs must be constrained so that they are invertible. Figure 2 shows the supported SMOs along with the mapping operation required to map a SELECT query back to the source version. The only operators that have any complications are adding a NOT NULL column and altering a column. The additions are handled by requiring that any NOT NULL column added in a branch must have a default value. This value is used to create data for the column when mapping query results forward to a branch. We have empirically assembled the following possible column alterations:

Change data type. Whether the change is to a less restrictive type (i.e. from a number or date field to a string) or to a more restrictive type, it is necessary to provide a mechanism to convert from the old type to the new type and vice versa. (This allows the data to be mapped from the source version to the branch for queries against the branch, and to map the data from the branch to back to the source version when the branch is merged.) This is supported by requiring a pair of stored functions that take one value and map it to the other. For instance, if a column *bar* in table *foo* is changed from an integer to a string in branch version 1.2, a pair of stored functions *1_2@foo@bar@to* and *1_2@foo@bar@from* must be created that do the data conversion in each direction.

Change nullability. Making a column NOT NULL has the same requirement as adding a NOT NULL column; the column must have a default value and this value replaces all occurrences of NULL values in the column. Making a column nullable requires no mapping operations.

Change constraints (length, range). This is handled the same way as changes to data type. A pair of mapping function must be defined to convert values going from source to branch and from branch to source version. One of these functions will just return the input value, while the other must handle values that fall outside the range when going from the less restrictive to the more restrictive constraint.

Change default value. No mapping operations required.

Change auto-increment. Make the primary key column auto-increment, or remove the auto-increment from the primary key column. No mapping operations required.

Note: the '@' character is used as a special delimiter in some table and function names that the versioning system creates or uses. Although this character is valid in MySQL names, its use is disallowed in our modified database engine.

If DDL is applied to a major version from which other versions (major or minor) have been created, the schema modifications are reconciled against any branches. If the change made to the source version is the same as a change that has been made in a branch, the delta is marked as disabled in the branch metadata. (The change is marked as disabled rather than being removed so that if the change is undone in the source version, it has not disappeared in the branch). For instance, consider the example shown Figure 1. Suppose that at time $t_{4.5}$, a column is added to the trunk schema. Branches 1.1 and 2 will be checked; if their deltas have added the same column, that addition is disabled in the branch. If at time $t_{4.6}$, that column is removed from the trunk again, the delta is re-enabled in the branch.

Suppose that at time $t_{6.5}$, a column is renamed in the trunk. This change must be checked against the schema changes for branches 2 and 2.1. If either of them has the same rename operation, that delta is disabled. If the new name conflicts with a column that was added in the branch, the column addition is disabled and an exception log is generated; it may be necessary to adjust queries and possibly change the name of the conflicting column that was added in the branch.

3.3. Data Versioning

As mentioned previously, in order to provide isolation between branches, it is necessary to version not only the database schema changes but also data changes made in a branch. Some techniques have been investigated for handling data versioning in an unchanging schema, such as the approach introduced by Chatterjee in [**56**] that uses long-running transactions to provide isolation for the data and to allow it to be discarded when it is no longer needed. However, this approach is limited in scope to setting up a predefined test data set; it does not provide the capability to isolate the branch and still allow for selective migration of some data back to the trunk when the branch is merged.

To reiterate design requirements, updates to trunk data should be visible in all branches, but updates to branch data should only be visible in that branch. Another design goal is to avoid maintaining multiple copies of the same data in different branches; when possible, data should be maintained in the standard database tables in the trunk schema. The rationale for this is that the likelihood of maintaining (adding, updating, or deleting) a large amount of data in the branch is low. Most data that must be merged back to trunk will be in tables with a relatively small number of rows – lookup, configuration, and metadata tables versus large tables of transactional data.

To meet these requirements, a branch will store only added, updated, and deleted data. This data is stored in tables separate from the main tables; a set of tables is maintained for each branch. There are many different ways to store this data. Deleted rows in the branch are tracked using a special table that contains the primary key values of the deleted rows. For updated data, our initial approach utilizes a Column-Oriented Data Store (CODS) similar to the one analyzed by Liu et al. in [**47**]. For any updates to a table in a non-trunk branch, any column whose data is changed will create a separate table containing the primary key of the row that was affected and the new value for the column. This approach requires that every table have a primary key; it would be very difficult to use a row ID construct because of the difficulty of coordinating between the rows in the trunk table and new rows in different branches.

For example, suppose a table *foo* in the trunk schema has an auto-increment primary key column *ID* and an integer column *bar*. Now suppose the following SQL statements are executed in branch 1.2:

```
ALTER TABLE foo ADD bar2 INTEGER;

UPDATE foo SET bar = bar + 1 WHERE ID < 4;
UPDATE foo SET bar2 = bar * 3 WHERE ID IN (1,4,5,6);
DELETE FROM foo WHERE ID = 5;
INSERT INTO foo (bar, bar2) VALUES (5, 23);
```

Figure 2 shows the tables that will be present in the database following the execution of these statements. Tables associated with a version are prefixed by the version number, and the '@' character is used to delimit components of the table name. The table *foo* is the table present in the trunk, so it does not have the version number prefix. For the *deleted* table, we use "@@" to avoid possible conflicts with tables having a column named *deleted.*

ID	bar
1	3
2	6
3	9
4	12
5	15
6	18

foo

ID	bar
1	4
2	7
3	10

1_2@foo@bar

ID	bar2
1	12
4	36
6	54

1_2@foo@bar2

ID	bar	bar2
7	5	23

1_2@foo

ID
5

1_2@foo@@deleted

Figure 2 – Example Tables for Data Versioning

As mentioned in the previous section, when a SELECT query is performed in a non-trunk branch, the query is rewritten to the trunk schema and executed, and then the data is mapped forward to the branch schema definition if necessary. Any modified data from the column stores then overwrites the corresponding columns in the data set.

For data added to the branch, we plan to create a *shadow table* that conforms to the branch version's schema. INSERT queries on the branch add rows to this shadow table, and subsequent updates will directly update the rows in this table. This approach is utilized rather than using the column oriented storage for both updated and added data in order to improve performance of queries on test data added in a branch. SELECT queries on the branch will perform the mappings described to fetch results from the trunk schema, merge data from the head branch (if the target branch is a minor branch of the head), update the data using any modifications, and query the local shadow tables and union the results of that query with the other data set to produce the final results.

To simplify subsequent merging of branch data into the trunk tables, we anticipate making a change such that a trunk table is affected by operations in a branch. In order to prevent conflicts, the values assigned to auto-increment columns share the same pool of values in the trunk and all branches. Consider the previous example of table *foo* with auto-increment primary key *ID*. As shown in Figure 2, the maximum value of *foo.ID* was 10. When a row was added to *foo* in branch 1.2 and no value is specified for *ID*, it was assigned the value 7. A subsequent insert into *foo* in the trunk would use the next value, 8, even though a row with *ID* 7 is not present in the trunk table. Using this approach avoids the necessity of correcting values when data is merged from the branch to the trunk; this correction could be very complex if the value was stored in a foreign key in other tables that were modified in the branch.

In order to facilitate scenarios in which a branch is created and a set of scripted unit and integration tests are to be run to identify regression issues, the versioning framework will provide a mechanism to override the standard mapping of data from the trunk tables into each branch. Executing a TRUNCATE TABLE statement in a branch will break the backward mapping to the table in the source version. That is, following the truncation, queries will only be executed against the shadow table associated with the branch. A deleted row table and any deleted column data tables associated with the table will be removed. This can be used, for instance, to empty a table in the branch so it can be populated with a predefined set of test data without affecting the data in the trunk. If it is actually desired to delete all rows in the table and to have that propagate back to the source version on a merge, DELETE FROM can be used.

While we plan to use column oriented storage to handle updates and deletes in a branch, we will also evaluate an alternate storage strategy in which the framework would always create the shadow table. This table would include the deleted flag as a column, and it would provide a mechanism whereby each element of a row tuple could have a new special NOT SPECIFIED value. This value would be similar to NULL, but would indicate that the element has not been affected in the branch.

3.4. Merging

The final component of the framework is a facility to merge schema and data changes made in a branch back into its source version; that is, to *commit* the changes. Changes in a branch are always merged with the source version; considering the example in Figure 1, changes in versions 1.1 and 1.2 would be merged into the trunk version, changes in version 2.1 would be merged into the head version, changes to version 2 would be merged into version 1 (at which time version 2 would become the trunk), and changes to version 2.3 would be merged into the trunk. (Details on the command to merge a branch are shown in Figure 12 in the appendix.)

Merging changes from a branch to the head version, like the merge of version 2.1 in the example, requires migration of the deltas from the minor version to the head version. This involves merging the schema deltas, possibly modifying the shadow tables in the head version, and merging the data in the branch column stores, deleted row table, and shadow tables into the tables for the head version.

Merging changes from a branch to the trunk version requires applying the schema deltas to the trunk database tables, then applying any data changes (deletions, additions, updates) that are to be merged. Applying DDL to tables in the trunk can cause significant disruption of service in a MySQL database, as described in [**54**] and [**55**]. High-volume production systems currently use techniques to perform the changes on a replicated database to minimize the impact, but there is still some downtime required to perform some of the modifications. See the methods described by Callaghan [**57**] and Noach [**58**] for the current state of the art. We plan to integrate an improved solution into the merge facility to perform schema modifications to online tables. This should address the performance penalties that are inherent in most ALTER TABLE commands in MySQL; the modified schema update process will minimize the amount of time when a table is locked and unavailable for queries. We plan to use a shadow table approach similar to the one described in [**57**] and [**58**]; the storage for this table can utilize the shadow table that was created to store any added data in the branch. We anticipate creating a more stable, less error prone mechanism to migrate the data from the original table to the shadow table and to then perform the renames to make the new table available.

Merges of a branch into the head or trunk require that the deltas in the branch be reconciled against any other branches created from the same source version. This is very similar to the work that must be done if DDL is applied to the trunk version after branches are created, as described at the end of section 3.2. For instance, in the example in Figure 1, when version 1.2 is merged into the trunk, its deltas must be compared against the current deltas in versions 1.1 and 2, just as if the modifications had been performed directly on the trunk after versions 1.1 and 2 were created.

As was mentioned earlier, some data that was added to the branch will be test data, and should be discarded while merging the branch back into the trunk. Other data should be retained and merged. The framework tracks information about whether data should be merged or discarded on a per-table basis. The default behavior is to maintain and merge data changes. Any table on which a TRUNCATE TABLE has been executed will have the changes discarded. There is an additional SQL command that can be used to modify a table so that changes are not merged. (Details on the syntax of this command are given in Figure 13 in the appendix.)

In the scenario where data changes are merged, it is possible that conflicts will be encountered where the data in the branch was changed and the data in the trunk was changed after creating the branch. Possible methods of handling these collisions are being researched. The initial system will use a very simple algorithm – modified data in the branch always wins and will overwrite the data in the trunk during the merge. A more complicated mechanism would be last writer wins; however, this would require timestamp on the rows, or an independent log of updates to be used to determine the priority data.

As part of the versioning framework, we plan to develop tools to display the schema changes and data differences between a branch and its source branch. In its initial implementation, the need to resolve conflicting changes in the branch and the source version is avoided, but if the semantics of schema modifications or of data updates are expanded, we anticipate enhancing these tools so they can be used to resolve merge conflicts.

Another initial simplification involves dealing with minor branches if their source branch is merged. For example, consider the timeline in Figure 1. If branch 1.1 had not been merged before branch 2 was merged, how would it be handled? Initially, we plan to disallow the merge of the head into the trunk if there are outstanding minor branches from the trunk. In the future, we will examine whether branches from the trunk can be re-parented to the head version when head and trunk are merged.

4. Preliminary Work

We have submitted a paper to the 2011 Very Large Database (VLDB) conference [**4**]. This paper describes our proposed framework and includes an analysis of the evolution of a commercial SaaS system's database schema as compared to the evolution of the MediaWiki schema.

In unrelated research, we also published a paper in 2005 on creating concept hierarchies using user queries against an information retrieval system [**68**]. We were invited to significantly expand this work into a chapter of a book on data mining advances that was published in 2008 [**69**].

4.1. Characterizing Schema Evolution

Studies have been conducted on schema evolution in large-scale Web-based Information Systems; an often referenced study by Curino et al. details the evolution of the MediaWiki database [**3**]. This database had 171 released schema versions in 4.5 years – Table 3 shows a breakdown of the different *Schema Modification Operators (SMOs)* that were applied to the schema over that time period, with the frequency of each operation in the second column (also from [**3**]).

To contrast the evolution of a Web-based Information System such as MediaWiki to a SaaS system, we have studied the schema evolution of a commercial SaaS system over a four year period. Over the latest two years, during which there were eight official product releases, the number of tables in the baseline database (one that includes only standard product features, with no customizations for any customer) increased by nearly 75 tables, to over 300. These tables contained over 3,000 columns. In contrast, as of the study published in 2008, the MediaWiki database included a total of 34 tables, with a total of 242 columns. The third column of Table 3 shows more details of the mix of schema modifications that were made on the two database schemas; the MediaWiki analysis spanned 4.5 years, and the SaaS system analysis spanned a four year period. The MediaWiki evaluation did not include any statistics on the number of times a column was altered – a change of data type, length, or constraints such as nullability. This was a fairly common modification to the SaaS database schema. While the number of removals of tables and columns and rename operations were roughly similar, the number of tables and columns that were added to the SaaS system was far greater. This emphasizes the heightened demand for new features in the SaaS system.

In addition to the schema associated with the product, we analyzed the schema customizations that were made for individual customers. This analysis was conducted over a sample of 1,850 individual customer schemas.

Table 3 – Occurrences of Schema Modification Operators

Schema Modification Operator	**# occurrences MediaWiki** [**3**]	**# occurrences SaaS System** [**4**]
CREATE TABLE	24	178
DROP TABLE	9	7
RENAME TABLE	3	12
UNION TABLE	4	0
COPY TABLE	6	0
ADD COLUMN	104	301
DROP COLUMN	71	41
RENAME COLUMN	43	24
MOVE COLUMN	1	15
COPY COLUMN	4	0
ALTER COLUMN	*	39

Table 4 shows statistics about the number of customer schemas that have been augmented by the addition of custom columns to standard product tables in the schema. The statistics include both the five-number summary (minimum, first quartile Q1, median M, third quartile Q3, maximum) and the mean μ and standard deviation σ. There are three sets of statistics – the first summarizes the number of custom columns per schema, irrespective of table, while the second summarizes the number of custom columns per table, irrespective of schema. The third is also the number of custom columns per table, irrespective of schema, but only considers instances of the table most often customized.

Table 4 – Customer Schemas with Custom Columns

# schemas with custom columns	1,392 (75%)
Total custom columns	99,551
(Min; Q1; M; Q3; Max) per schema	(1; 9; 27; 79; 1,117)
(μ; σ) per schema	(71.6; 117.5)
# table types w/ custom columns	9
(Min; Q1; M; Q3; Max) per table	(1; 2; 6; 20; 503)
(μ; σ) per table	(22.9; 47.7)
(Min; Q1; M; Q3; Max) per table, for most customized table	(1; 4; 12; 37; 503)
(μ; σ) per table	(33.7; 57.1)

Table 5 – Customer Schemas with Custom Tables

# schemas with custom tables	249 (13.5%)
Total custom tables	1,037
(Min; Q1; M; Q3; Max) per schema	(1; 1; 2; 4; 36)
(μ; σ) per schema	(4.2; 5.8)
Total columns in custom tables	8,833
(Min; Q1; M; Q3; Max) per table	(1; 3; 7; 9; 117)
(μ; σ) per table	(8.5; 9.3)
Total rows in custom tables	659,925,926
(Min; Q1; M; Q3; Max) per table	(1; 10; 198; 10,750; 60,878,797)
(μ; σ) per table	(636,379; 3,587,857)

These numbers clearly show the customer demand for extending the standard product solution with schema customizations. 75% of the SaaS customers extended the tables in their database schemas with custom columns; they added a median of six and an average of nearly 23 fields per customizable table, with the most often customized table having a median of 12 and an average of nearly 34 custom columns added.

Table 5 shows statistics about the number of customer schemas that had been augmented with custom tables. These tables have a variety of uses; some are small tables that store lookup information (lists of sales regions, product codes, etc.) or configuration information, while others are very large tables of transactional data. Again, data is presented using the five-number summary as well as the mean and standard deviation. There are statistics for the number of custom tables per schema, the number of columns per custom table, and the number of rows per table. These numbers show the demand for customizations beyond just adding custom attributes to predefined tables in the schema. 13.5% of the SaaS customers added custom tables to their schema. The number of custom tables per customer was relatively small - a median of two and average of four tables per customer schema. This was likely influenced by the significantly greater complexity of adding custom tables compared to adding custom columns. The statistics on the number of rows per table show what a range of uses these custom tables have, from small lookup tables to tables with tens of millions of rows.

4.2. Creating a Benchmark Data Set

We have started to design a reference schema and data set that can be used to evaluate the versioning framework. We want a benchmark data set that is more representative of a high-volume transaction processing system than the MediaWiki database, and we want to create a benchmark that will be made available online. It will be less complex than the schema of the SaaS system we evaluated, but should have a number of the same characteristics. We will use this benchmark data and the MediaWiki database to perform evaluation of any experiments; this will allow the framework to be evaluated in a well-understood and public environment and will allow researchers to compare various performance metrics for similar systems. We will also evaluate the framework on the SaaS system database that we have evaluated.

The benchmark data set consists of a sequence of scripts containing schema modifications (in the form of MySQL DDL statements), meant to simulate a sequence of system upgrades and separate system customizations. These scripts will use the SQL extensions we are implementing to create branches, select branches, and merge branches back into their source versions, but these commands will be easily modified so the scripts can be used on standard SQL databases. The data set will also include an accompanying set of scripts to populate the database schema with test data. There must be a sufficient volume of test data to allow meaningful performance measurements to be made, particularly when evaluating alternative designs and implementations of parts of the framework.

In addition, we are adopting a test-driven approach to the design and development of the framework; included in the benchmark data set will be a set of scripts to emulate the applications. These scripts will be comprised of a set of different queries along with logic to check the results. These scripts will help to document the expected behavior of the framework. In addition to serving as a load generator to be used to measure performance, the test scripts will provide integration and regression testing of the database.

The entity-relationship (ER) diagram in Figure 3 shows the initial database schema (version 1). The schema is designed to simulate a very simple CRM system. This is somewhat similar to the test schema presented by Aulbach et al. in [**1**] and [**2**]; however, this database is more helpdesk and customer service oriented, while the one used by Aulbach was geared toward sales automation.

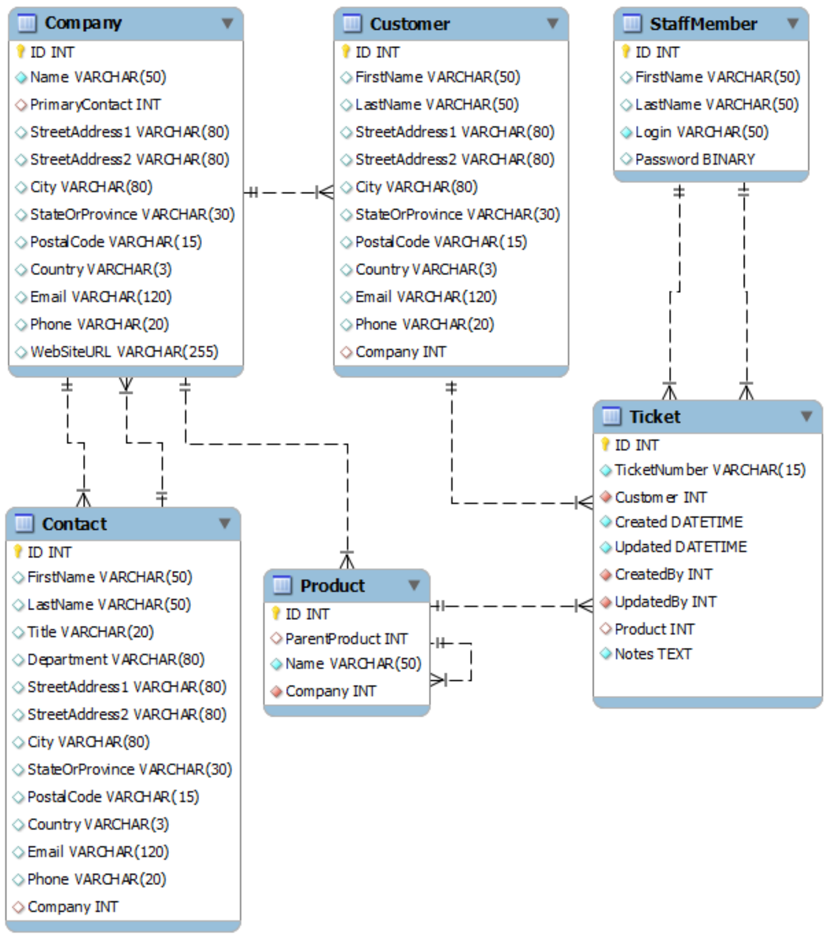

Figure 3 – ER Diagram for Benchmark Schema, Version 1

In this diagram, columns with solid squares are required (NOT NULL), while columns with hollow squares are not required. Columns with blue squares are simple fields; columns with red squares are foreign key fields. The primary key column is indicated by a key icon.

The first upgrade, version 2, includes the following SMOs and associated data modifications. The ER diagram for the version 2 of the schema is shown in Figure 4.

- ADD COLUMN *ParentCompany* to the *Company* table, to allow the creation of hierarchies.
- CREATE TABLE *TicketThread*. Instead of a single *Notes* field to contain all of the information associated with a ticket, create a new *TicketThread* table.
- Run a script to populate the *TicketThread* table with any notes contained in already-created tickets.
- DROP COLUMN *Notes* from the *Ticket* table.

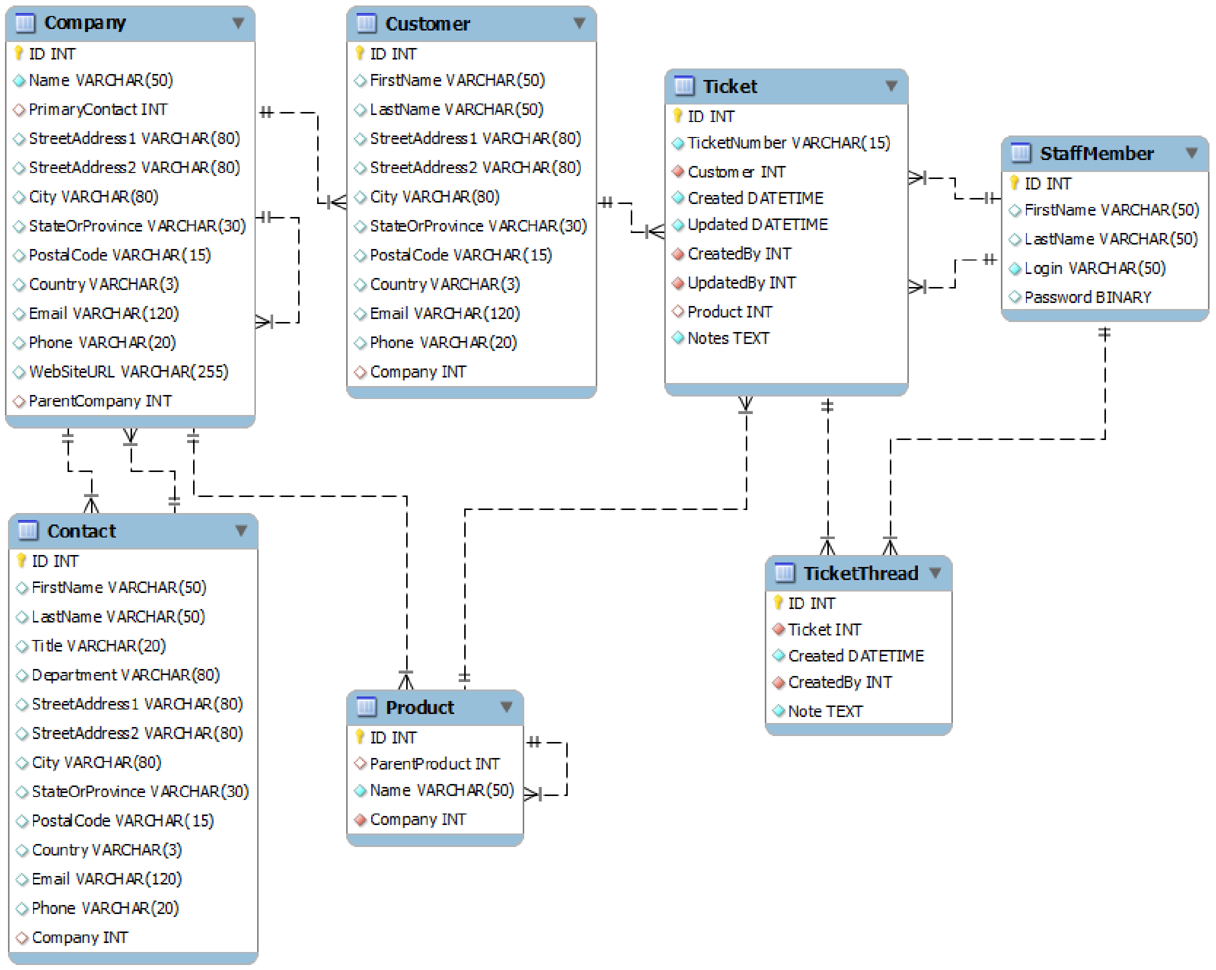

Figure 4 – ER Diagram for Benchmark Schema, Version 2

The next upgrade to version 3 of the schema includes the following changes, as shown in the ER diagram in Figure 5.

- CREATE TABLE *CompanyAddress*. Multiple addresses must be stored for a company (e.g. mailing, shipping, headquarters). Instead of prefixing the five address columns with a common name and creating multiple instances of the five columns (e.g. *MailingCity*, *ShippingCity*, *HQCity*), create a new table that includes an address type field and the individual address fields.
- RENAME COLUMN *Phone* in the *Customer* table to *PhoneHome*.
- ADD COLUMN *PhoneMobile* and *PhoneWork* in the *Customer* table.
- RENAME COLUMN *Phone* in the *Contact* table to *PhoneWork*.
- ADD COLUMN *PhoneHome* and *PhoneMobile* columns.
- ADD INDEX *Customer*.*Email* – unique index.
- Run a script to populate the *CompanyAddress* table with any addresses contained in already-created companies – use the *Mailing* value for *AddressType*.
- DROP COLUMN *StreetAddress1*, *StreetAddress2*, *City*, *StateOrProvince*, *PostalCode*, *Country* from *Company*.

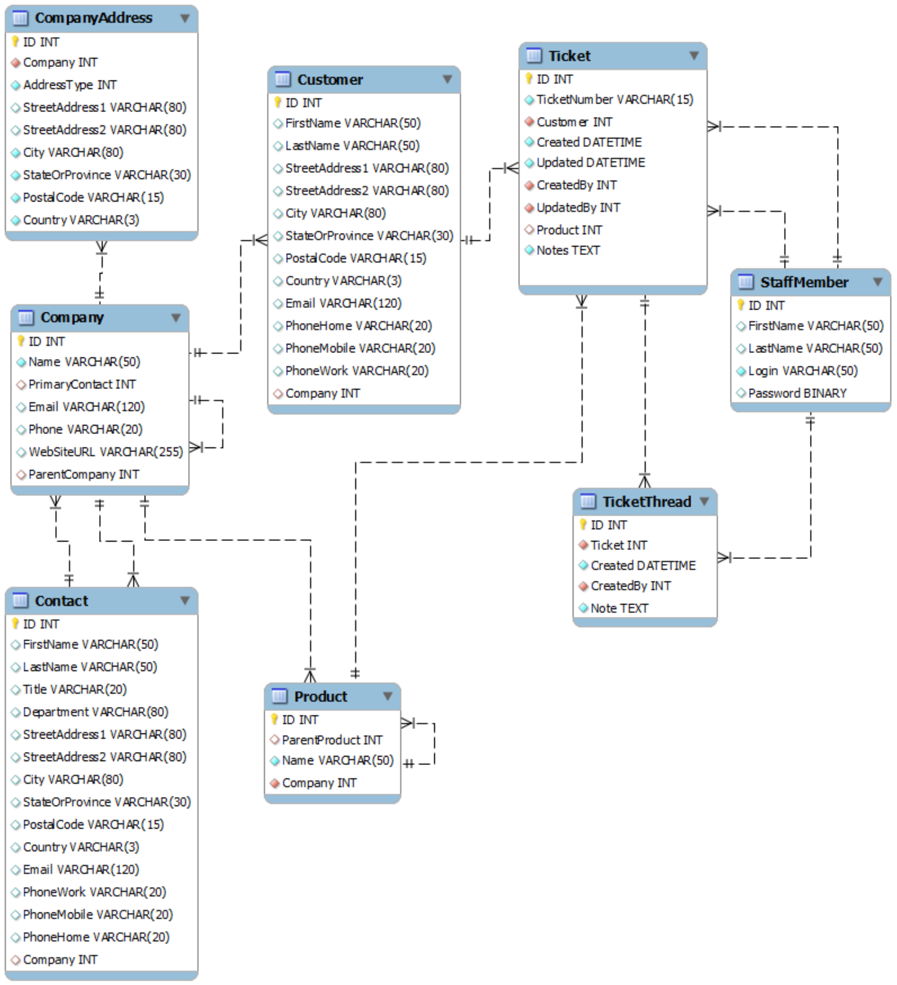

Figure 5 – ER Diagram for Benchmark Schema, Version 3

The next upgrade to version 4 includes the following changes, as shown in Figure **6**.

- CREATE TABLE *CustomerToCompany* – this mapping table will allow a customer to be associated with multiple companies.
- Run a script to populate the *CustomerToCompany* table with any company values contained in already-created customers.
- CREATE TABLE *TicketToCustomer* - this mapping table will allow a ticket to be associated with multiple customers.
- Run a script to populate the *CustomerToCompany* table with any company values contained in already-created customers.
- DROP COLUMN *Customer.Company*
- DROP COLUMN *Ticket.Customer*.

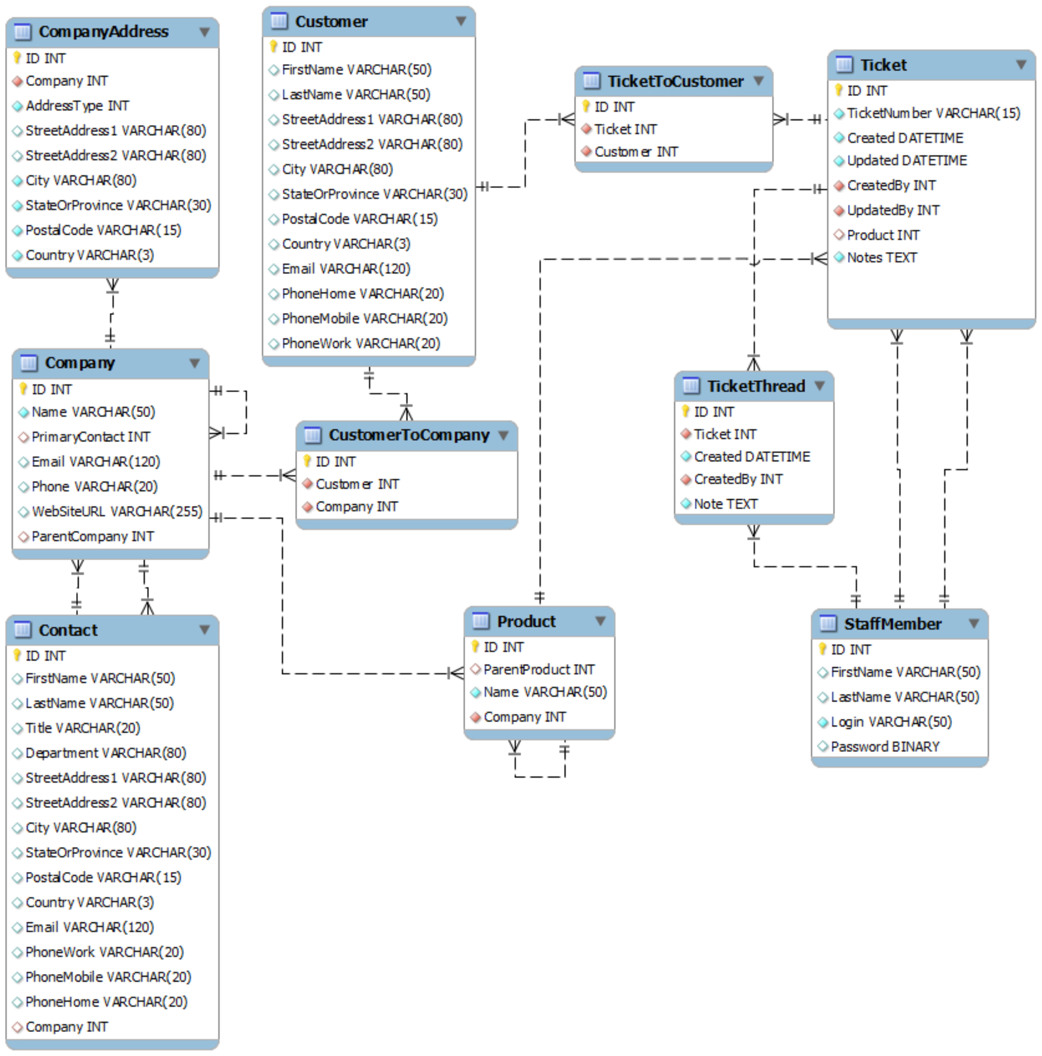

Figure 6 **– ER Diagram for Benchmark Schema, Version 4**

5. Research Plan

This section contains an outline of the plan to complete the research for the dissertation. It is arranged in roughly chronological order.

Our plan is to modify the MySQL database server to incorporate the versioning framework. The different research tasks enumerated below include metrics for evaluating alternatives and for determining the success of different approaches. One common metric across all of the tasks is a measure of how the impact the code changes have on the standard MySQL database execution. Ideally, the operation of the database would be completely unaffected by the addition of the framework if a client connects to the trunk version of the database. Some instances have already been identified where this cannot be the case; there must be some code paths in the database engine that require the insertion of conditional logic to handle versioning. Changes to the code to add the framework will be measured in terms of their impact on the execution paths through the default original database server code. As mentioned in section 2, Background, most of the research published recently has focused on object-oriented systems, but the work by Herzig using Transition Dependency Graphs (TDGs) can be applied to procedural code as well ([**65**]).

We plan to use version 5.3 of MariaDB, which should be released for Beta in mid-May, as the database server code base on which to implement the framework. MariaDB is a fork of the MySQL database that was created by Michael “Monty” Widenius, the original author of MySQL, in the wake of MySQL’s acquisition by Sun and Sun’s subsequent acquisition by Oracle. Mr. Widenius’ company, Monty Program AB, is actively developing MariaDB. See http://www.mariadb.org for more information.

5.1. Benchmark Data Sets

The next step in the research is to flesh out the schema for the benchmark data set, and then generate test data with which to populate it. This will be done concurrently with designing and implementing the test scripts that will be used for integration and regression testing.

In addition to the sequence of schemas depicted in section 4.2, we will create a separate data set containing a more abstract data model. This data model will be less realistic, but it will provide a complete set of data types and relationships. The goal of this data set is to provide a second framework for making measurements and evaluating alternatives.

The reference data sets will be packaged as sets of Linux shell scripts that will create and initialize a MySQL database, create the initial schema, populate it with the initial data set, and run a set of tests to validate the starting configuration. The master script will then run through a sequence of simulated version upgrade and customization scenarios, each of which will perform the following steps:

- create a branch
- perform a set of schema and data modifications in the branch
- run a set of test scripts that query the branch, the trunk, and possibly other branches to confirm proper operation and to measure performance
- merge the branch and measure the performance of the merge
- run additional scripts to verify the correctness of the merge

The suite will include scenarios that discard a branch and that create two or more branches that exist concurrently.

In addition to these data sets, we will package the set of MediaWiki schema updates, along with a set of test data, in the same packaging, so that we can use this as another benchmark and perform the same measurements and evaluations against that schema.

5.1.1. Simple Example

For the remainder of the proposal, please consider the following simple scenario. This will be used to illustrate some of the concepts with concrete examples. The scenario is presented as a sequence of SQL commands, with the expected output of commands shown in red.

Suppose the trunk schema starts out empty, and then a table *test1* is defined and populated as follows:

```
CREATE TABLE test1
(   id INT NOT NULL AUTO_INCREMENT,
    int_1 INT NOT NULL,
    int_2 INT,
    str1  VARCHAR(50) NOT NULL,
    str2  VARCHAR(50),
    dt1   DATE NOT NULL,
    dt2   DATE,
    PRIMARY KEY (id)
);
INSERT INTO test1 (int_1, str_1, dt_1)
VALUES (1, ‘A1’, ‘2011-01-01 00:00:00’);
INSERT INTO test1 (int_1, int_2, str_1, str_2, dt_1, dt_2)
VALUES (2, 4, ‘A2’, ‘Red’, ‘2011-02-01’, ‘1970-01-01’);
INSERT INTO test1 (int_1, int_2, str_1, str_2, dt_1, dt_2)
VALUES (3, 6, ‘A3’, ‘Blue’, ‘2011-03-01’, ‘1980-01-01’);
```

Then suppose the following SQL is executed to create a new head version:

```
CREATE VERSION NEXT;
SHOW VARIABLES LIKE 'm%version';
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| major_version | 2     |
| minor_version | NULL  |
+---------------+-------+
2 rows in set (0.00 sec)

ALTER TABLE test1 ADD COLUMN new_int1 INT NOT NULL DEFAULT 32,
                  ADD COLUMN new_int2 INT,
                  ADD COLUMN new_str1 VARCHAR(50) NOT NULL DEFAULT ‘Black’,
                  ADD COLUMN new_str2 VARCHAR(50),
                  CHANGE COLUMN int1 int1a INT NOT NULL;
DESCRIBE test1;
+----------+-------------+------+-----+---------+----------------+
| Field    | Type        | Null | Key | Default | Extra          |
+----------+-------------+------+-----+---------+----------------+
| id       | int(11)     | NO   | PRI | NULL    | auto_increment |
| int_1a   | int(11)     | NO   |     | NULL    |                |
| int_2    | int(11)     | YES  |     | NULL    |                |
| str1     | varchar(50) | NO   |     | NULL    |                |
| str2     | varchar(50) | YES  |     | NULL    |                |
| dt1      | date        | NO   |     | NULL    |                |
| dt2      | date        | YES  |     | NULL    |                |
| new_int1 | int(11)     | NO   |     | NULL    |                |
| new_int2 | int(11)     | YES  |     | NULL    |                |
| new_str1 | varchar(50) | NO   |     | NULL    |                |
| new_str2 | varchar(50) | YES  |     | NULL    |                |
+----------+-------------+------+-----+---------+----------------+
11 rows in set (0.00 sec)
```

```
INSERT INTO test1 (int_1a, str1, dt1, new_int1, new_str1)
VALUES (23, ‘B1’, ’2011-01-23’, 5, ‘Alpha’);
INSERT INTO test1 (int_1a, int_2, str1, str2, dt1, dt2, new_int1, new_int2, new_str1, new_str2)
VALUES (9, 16, ‘B2’, ‘Laptop’, ’2011-01-24’, ‘2000-01-01’, 6, 7, ‘Beta’, ‘Canary’);
SELECT * FROM test1;
UPDATE test1 SET int_2 = 6, str2 = ‘Green’ WHERE id = 2;
UPDATE test1 SET dt2 = ‘2010-04-20’ WHERE id = 1;
DELETE FROM test1 WHERE int_1a = 3;

SELECT id, int_1a a, int_2 b, str1 c, str2 d, dt1 e, dt2 f, new_int1 g, new_int2 h,
       new_str1 i, new_str2 j
  FROM test1;

+----+----+------+----+--------+------------+------------+----+------+-------+--------+
| id | a  | b    | c  | d      | e          | f          | g  | h    | i     | j      |
+----+----+------+----+--------+------------+------------+----+------+-------+--------+
|  1 |  1 | NULL | A1 | NULL   | 2011-01-01 | 2010-04-20 | 32 | NULL | Black | NULL   |
|  2 |  2 |    6 | A2 | Green  | 2011-02-01 | 1970-01-01 | 32 | NULL | Black | NULL   |
|  4 | 23 | NULL | B1 | NULL   | 2011-01-23 | NULL       | 5  | NULL | Alpha | NULL   |
|  5 |  9 |   16 | B2 | Laptop | 2011-01-24 | 2000-01-01 | 6  |    7 | Beta  | Canary |
+----+----+------+----+--------+------------+------------+----+------+-------+--------+
4 rows in set (0.03 sec)

SELECT new_int1 FROM test1;
ERROR 1054 (42S22): Unknown column 'new_int1' in 'field list'
```

At this point, the scenario switches back to the trunk version to verify the isolation of data and schema changes in the head version:

```
SET VERSION DEFAULT;
SHOW VARIABLES LIKE 'm%version';
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| major_version | 1     |
| minor_version | NULL  |
+---------------+-------+
2 rows in set (0.00 sec)

INSERT INTO test1 (int_1, int_2, str1, str2, dt1)
VALUES (6, 25, ‘A3’, ‘Brown’, ‘2012-03-19’)
SELECT id, int_1 a, int_2 b, str1 c, str2 d, dt1 e, dt2 f FROM test1;
+----+----+------+----+--------+------------+------------+
| id | a  | b    | c  | d      | e          | f          |
+----+----+------+----+--------+------------+------------+
|  1 |  1 | NULL | A1 | NULL   | 2011-01-01 | NULL       |
|  2 |  2 |    4 | A2 | Red    | 2011-02-01 | 1970-01-01 |
|  3 |  3 |    6 | A3 | Blue   | 2011-03-01 | 1980-01-01 |
|  6 |  6 |   25 | A3 | Brown  | 2012-03-19 | NULL       |
+----+----+------+----+--------+------------+------------+
3 rows in set (0.03 sec)

SELECT new_int1a FROM test1;
ERROR 1054 (42S22): Unknown column 'new_int1a' in 'field list'
```

The scenario then switches back to the head version and merges it into the trunk, advancing the trunk to version 2. There are some queries to confirm that the merge was executed correctly.

```
SET VERSION 2;
MERGE VERSION;
SET VERSION DEFAULT;
SHOW VARIABLES LIKE 'm%version';
```

```
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| major_version | 2     |
| minor_version | NULL  |
+---------------+-------+
2 rows in set (0.00 sec)

DESCRIBE test1;
+----------+-------------+------+-----+---------+----------------+
| Field    | Type        | Null | Key | Default | Extra          |
+----------+-------------+------+-----+---------+----------------+
| id       | int(11)     | NO   | PRI | NULL    | auto_increment |
| int_1a   | int(11)     | NO   |     | NULL    |                |
| int_2    | int(11)     | YES  |     | NULL    |                |
| str1     | varchar(50) | NO   |     | NULL    |                |
| str2     | varchar(50) | YES  |     | NULL    |                |
| dt1      | date        | NO   |     | NULL    |                |
| dt2      | date        | YES  |     | NULL    |                |
| new_int1 | int(11)     | NO   |     | NULL    |                |
| new_int2 | int(11)     | YES  |     | NULL    |                |
| new_str1 | varchar(50) | NO   |     | NULL    |                |
| new_str2 | varchar(50) | YES  |     | NULL    |                |
+----------+-------------+------+-----+---------+----------------+
11 rows in set (0.00 sec)

SELECT id, int_1a a FROM test1 ORDER BY id;
+----+----+
| id | a  |
+----+----+
|  1 |  1 |
|  2 |  2 |
|  4 | 23 |
|  5 |  9 |
|  6 |  6 |
+----+----+
5 rows in set (0.00 sec)
```

5.2. Version Management Changes

The first step in actually working on the framework will be to design and implement the changes described in the Appendix to handle version management. This will include adding variables to MySQL for the major and minor version and augmenting the parser and command processor to handle the additional syntax. This will be mostly an implementation task, but it will be an opportunity to begin getting familiar with the MySQL/MariaDB code base, and to start investigating the code change impact metrics and how they can be used to guide design decisions.

This task will be evaluated for correctness; there is no significant performance impact expected. We will develop a set of test scripts to confirm proper operation of all the extensions. We will also explore the code impact metrics and how we can use them to evaluate alternatives.

5.3. Data Storage for Branch Data

The next research task will be to investigate methods for storing data associated with a branch and maintaining the required isolation from other branches and the trunk, while still allowing queries to access data in the trunk and the branch. As mentioned, we plan to explore at least two different options for storing the updated data in a branch – the column-oriented data store (CODS) and shadow tables augmented by a special *NOT SPECIFIED* indicator. We plan to review the research on CODS (Stonebraker [**45**], Abadi [**46**], Liu [**47**], Plattner [**48**], and Grund [**49**]) for the best alternatives. The research by Grund on combining multiple columns per vertical partition of the table appears to be particularly appropriate.

The metrics used to evaluate these techniques will include the following:

- number of disk blocks accessed
- the spread of accesses across disk (in as far as we can measure this), to evaluate the impact of losing locality of reference in the CODS
- mean, median, maximum query time (CPU time and wall clock time)

The storage will be evaluated using a branching scenario that includes new tables, new columns, modified columns, and removed columns, with new rows, deleted rows, and updated rows in several modified tables. A mix of different queries will be created against this branch schema to cover the different scenarios where data must be fetched from the production tables and merged with data in the branch storage. Initially, the mapping of these queries from the branch back to the trunk and the mapping of the result data back to the branch will be done by hand.

Consider the example given in section 5.1.1. Before the branch is merged, the table structures in the schema will look like those shown in Figure 7, using the CODS approach. Since table *test1* is in the trunk schema, it does not have a version prefix. Three CODS tables are added for the updated data in existing rows. The table *2@test1@@deleted* is added to hold the primary key values of any deleted rows.

id	int_1	int_2	str1	str2	dt1	dt2
1	1	NULL	A1	NULL	2011-01-01	NULL
2	2	4	A2	Red	2011-02-01	1970-01-01
6	6	25	A3	Brown	2012-03-19	NULL

test1

id	int_2
2	6

2@test1@int_2

id	str2
2	Green

2@test1@str2

id	dt2
1	2010-04-20

2@test1@dt2

id
3

2@test1@@deleted

id	int_1a	int_2	str1	str2	dt1	dt2	new_int1	new_int2	new_str1	new_str2
4	23	NULL	B1	NULL	2011-01-23	NULL	5	NULL	Alpha	NULL
5	9	16	B2	Laptop	2011-01-24	2000-01-01	6	7	Beta	Canary

2@test1

Figure 7 – Branch Data Storage using Column-Oriented Data Store

The table structures will look like those shown in Figure 8, using the shadow table approach. Again, *test1* is the normal table in the trunk schema; it is unchanged regardless of the approach for storing branch data. All other modifications are stored in the shadow table, including the deleted row flag, all new rows added to the table, and all modifications to existing rows.

id	int_1	int_2	str1	str2	dt1	dt2
1	1	NULL	A1	NULL	2011-01-01	NULL
2	2	4	A2	Red	2011-02-01	1970-01-01
6	6	25	A3	Brown	2012-03-19	NULL

test1

id	@deleted	int_1a	int_2	str1	str2	dt1	dt2	new_int1	new_int2	new_str1	new_str2
1	0	***	***	***	***	***	2010-04-20	***	***	***	***
2	0	***	6	***	Green	***	***	***	***	***	***
3	1	***	***	***	***	***	***	***	***	***	***
4	0	23	NULL	B1	NULL	2011-01-23	NULL	5	NULL	Alpha	NULL
5	0	9	16	B2	Laptop	2011-01-24	2000-01-01	6	7	Beta	Canary

2@test1 *** - column value NOT SPECIFIED

Figure 8 – Branch Data Storage using Shadow Table

Another additional research opportunity for the shadow table storage approach is an evaluation of ways to handle the *NOT SPECIFIED* marker for columns and the *deleted* specifier for each row. We may want to avoid adding too much additional non-standard information to each row of the shadow table, so that it can be reused as the table into which to migrate the data in the trunk table during the merge (see the following section on merging for details). This will require exploring alternatives for storing this extra data for each row, including a separate CODS. We will measure these alternatives in terms of query performance using the same metrics we use to evaluate the CODS against shadow tables for the main data storage in the branch.

This is just a simple illustrative example; the actual benchmark will include operations on significantly larger and more complicated data sets, in order to obtain useful performance metrics to use in evaluating the data storage options. Performance will be measured by executing the queries directly on a dedicated database server machine with background load minimized and equalized across all tests as much as possible, to minimize the effects of network latency, process switching, other access to disk storage, etc.

Note that the performance of the branch data storage compared to queries directly against the trunk branch is a lesser concern. As long as performance does not degrade to an unusable level in a sandbox, the branch storage technique will be considered acceptable. Average query performance of 25% of the performance of queries against the standard schema (a four times slow-down) will be considered usable.

5.4. Schema Deltas

The next step will be to design and implement handling of the schema deltas. This will require changes to the MySQL command processor to isolate the DDL statements that perform the schema modifications. The validation portion of the command processor must also be modified to take into account the schema deltas when validating DDL and DML (data manipulation language) statements.

Once we are able to identify SMOs while processing SQL statements, we will evaluate options for storing the schema deltas. One option is to have a set of metadata tables similar to *information_schema.tables*, *information_schema.columns*, *information_schema.table_constraints*, etc. that are augmented with major and minor version numbers and additional information to facilitate mapping back to the corresponding elements in the source version. Another option is to just create a set of metadata tables that contain the mapping information and to dynamically apply these as needed to materialize the effective schema for a particular branch. We plan to examine the schema mapping techniques described by Bernstein [**40**], [**41**] and by Fagin [**43**], [**42**] as a starting point.

The options will be evaluated based on the impact of changing the source code to accommodate them, the relative complexity of the code, and the performance of the database engine in parsing and processing queries in the branch (in terms of the time required to parse the query and prepare it for processing, independent of time required to actually evaluate the query). It is not anticipated that the performance differences will be significant, but we will run tests to confirm this. The benchmarks will include integration tests that include a mix of queries against a branch designed to measure the average time and range of times across types of queries.

The subsequent actions that will be taken depend on the SMOs that were identified:

Table 6 – Changes to Branch Data Storage for Schema Modification Operators

SMO	**Additional Actions**
CREATE TABLE	Create the table, with the appropriate version number prefix in the name.
RENAME TABLE	Rename shadow table if it exists. Rename any CODS tables if they exist.
DROP TABLE	Drop shadow table if it exists. Drop any CODS tables if they exist.
ADD COLUMN	Create or modify shadow table. Note that CODS table is not automatically created until the column is referenced when updating a row in the existing table.
RENAME COLUMN	Create or modify shadow table. Rename CODS table if it exists.
DROP COLUMN	Create or modify shadow table. Rename CODS table if it exists.
ALTER COLUMN	Create or modify shadow table. Modify CODS table if it exists.
ADD INDEX	Create multi-version index
DROP INDEX	Drop version from multi-version index
ADD PK	Same as adding a unique index
DROP PK	Drop version from multi-version index
ADD FK	To be determined
DROP FK	To be determined

Handling indices, particularly unique indices, in a branch will be complicated. We will initially investigate the multi-version B+ trees introduced by Becker [**53**] and the refinements proposed by Haapasalo et al. [**50**], [**51**], [**52**].

The benchmarks will include integration tests designed to verify the correctness of the schema definition handling in a branch, and in concurrent branches.

5.5. Query Mapping for SELECT Queries

We will next design and implement query mapping for SELECT queries against the branch database. We plan to examine the research on schema mapping for efficient techniques [**40**], [**41**], [**42**], [**43**]. Since we have constrained the versioning system to require at most two levels of mapping (from branch to head to trunk), we should have less performance penalties than if we allowed an arbitrarily long sequence of schema changes to coexist.

The following are some kinds of queries that we must handle; each shows an example using the schema from 5.1.1.

Simple single-table queries: handle any renames of table or columns. Ignore columns that were added in branch. Consider the following query:

```
SELECT ID, int_1a, new_int1 FROM test1;
```

This would be mapped to the following queries:

```
SELECT ID, int_1a, new_int1 FROM
(SELECT ID, int_1 int_1a, 32 new_int1 FROM test1
 ORDERED UNION
 SELECT ID, int_1a, new_int1 FROM 2@test1) A
LEFT JOIN 2@test1@@deleted B ON A.ID = B.ID
WHERE B.ID IS NULL;
```

This uses the left outer join with the deleted row table to implement a set difference operation. It also introduces a new concept of an ORDERED UNION, where we guarantee that rows from the second select clause take precedence over rows from the first select clause in the event of a matching primary key value. This obviously requires that both tables have the same primary key, which will be the case.

Multi-table joins, no grouping or aggregate functions: similar to a single-table query, performing the same handling of renaming, additions, and deletions. Obviously, this might be more complicated if the columns used for the join condition are columns that have been modified.

ORDER BY clauses may also require special handling. It may be necessary to materialize the query data (create a temporary table with the results of the query mapping, apply any updates with data from the branch) then sort the final results.

Evaluation of the mapping will be done first on the basis of correctness. The benchmark data sets will include a set of queries to confirm correct functionality of the query mapping against the three possible branch types in the schema: a branch from the trunk, the head branch, and a branch from the head. The scripts will create the full range of possible data conditions: data only present in trunk, data added in branch, data updated in branch, data in trunk deleted in branch, etc.

Measuring alternatives will be done on the basis of performance, using the metrics enumerated in section 5.3.

Handling of grouping will be dealt with if time permits after the basic query mapping and merge tasks have been completed.

5.6. Handling Inserts and Updates

Handling SQL INSERT statements in a branch should be relatively simple – the data should just be inserted into the appropriate shadow table. However, we will need to handle complications including the following: primary/unique keys, foreign keys (referential integrity), and auto-increment primary key values.

SQL UPDATE statements will also require simple inserts or updates to the CODS tables or the shadow table. However, we will also need to handle referential integrity for these statements as well. For both INSERT and UPDATE, we may need to perform query mapping to resolve the WHERE clause in the query.

These modifications will be evaluated in the same way as the SELECT statements; correctness will be confirmed by comprehensive set of integration tests, and performance will be measured using the metrics from section 5.3.

5.7. Branch Merge

When a branch is merged, the schema of the source version must be updated; if the source is the head, this will involve combining the CODS and/or shadow tables for the branch and the head, then updating the schema deltas for the head to include the deltas for the branch. If the source is the trunk, then the trunk schema must be updated, and the data changes must be merged.

An issue to which we have not had much opportunity to investigate yet is conflict identification and resolution during the merge. This will include both schema merge conflicts (e.g. a column was modified in a branch then dropped in the source version prior to the merge) and data conflicts (e.g. a row was updated in the branch then deleted in the source version prior to the merge). The initial approach will be to minimize conflicts as much as possible; for example, our first implementation will be "branch wins" on most conflicts.

Merge modifications will be made to the database engine to allow schema changes to be merged into the production tables while the database remains online and the tables are available. That is, the merge should eliminate or drastically reduce the amount of time in which access to database tables is prevented (for example, by a lock on the tables).

The initial strategy is to employ a two-phase merge process, as follows:

- Lock the version when the merge starts. It will be unavailable for connections until the merge completes. In addition, the MERGE VERSION command will be disabled for other branches as well, to eliminate conflicts with between changes in different versions.
- Construct shadow tables for any tables with schema modifications, if they do not already exist.
- For each table to be updated, enable capture of modifications (insert, update, delete).
- Start a background task to copy rows from trunk table to shadow table, a block of 100,000 rows at a time.
- After all rows copied, replay the redo information on the shadow table, mapping as necessary.
- When redo information has all been replayed, lock the trunk table.
- Apply any last-minute redo information.
- Rename or delete the trunk table.
- Rename the shadow table to the trunk table name.

The most complicated part of this process is capturing the modifications. There are three mechanisms mentioned in [**57**]: use statement-based replication to capture any SQL statements that modify the table or tables being copied, use row-based replication to capture the row changes for those tables, and create triggers that fire on insert, update, and delete of those tables and update the shadow tables directly. Each of these approaches has problems; a solution that is more tightly integrated with the database would be less brittle and require less complicated setup. Our initial idea is to integrate this into the trigger mechanism, without requiring the explicit creation of triggers. The different alternatives will be compared based on their potential for errors, performance impact on the database engine, and the code impact of the changes.

The merge process will be evaluated against the standard database schema update process using the following metrics:

- wall clock time elapsed from the time the process starts to the time that all tables involved in the merge are available for use
- wall clock time elapsed from the time the tables involved in the merge are locked or otherwise made unavailable for queries until the time the modified tables are available for use again
- amount of additional (temporary) disk storage required to accommodate the schema updates

The experiments will be to modify the schema of a table containing at least ten million rows, with at least 25 columns per row. The modifications will include adding columns, removing columns, altering columns, and adding unique and non-unique indices. Executing any of these modifications in MySQL currently requires the table to be locked and a new table to be constructed. While the merge process is executing, a mix of inserts, updates, and deletes against the trunk version of the table will run concurrently to verify the correctness of the modification tracking mechanism.

5.8. Additional Research Tasks

The following tasks will be included in the dissertation if time permits.

Handling unique indices in branches. Methods of creating indices on data that might exist in two different tables must be investigated; ideally, these methods would avoid creating an index that duplicates the one that is present in the trunk schema.

Handling multi-column indices in branches. This is an extension of the previous issue – particularly if the framework uses the CODS storage for modified data in the branch, maintaining multi-column indices that include one or more of these modified columns is a complicated problem.

Handling foreign keys in branches. This is similar to the problem with indices – if a table includes foreign key references to other tables, and that table is modified in a branch, how should the referential integrity of the data be ensured?

Handling grouping and aggregation operators. Queries that include GROUP BY clauses and/or aggregate operators like SUM() and AVERAGE() will require special handling when the data on which they operate exists in both the branch and the source version. It may be necessary to materialize temporary tables to evaluate some aggregate functions.

Handling sub-selects. The initial modifications will not deal with nested sub-selects like the following:

```
SELECT int_1a FROM test WHERE ID IN (SELECT ID FROM test WHERE new_int1 > 7);
```

6. References

[1] S. Aulbach, T. Grust, D. Jacobs, A. Kemper, and J. Rittinger, "Multi-Tenant Databases for Software as a Service: Schema-Mapping Techniques," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD '08)*, Vancouver, BC, Canada, 2008, pp. 1195-1206.

[2] S. Aulbach, D. Jacobs, A. Kemper, and M. Seibold, "A Comparison of Flexible Schemas for Software as a Service," in *Proceedings of the 2009 International Conference on Management of Data (SIGMOD ‘09)*, Providence, Rhode Island, 2009, pp. 881-888.

[3] C. A. Curino, H. J. Moon, L. Tanca, and C. Zaniolo, "Schema Evolution in Wikipedia: toward a Web Information System Benchmark," in *10th International Conference on Enterprise Information Systems (ICEIS 2008)*, Barcelona, Spain, 2008.

[4] B. Wall and R. Angryk, "Schema and Data Versioning System," in *Proceedings of the VLDB Endowment (PVLDB)*, 2011, (in submission).

[5] MSDN, "Managing Database Change," http://msdn.microsoft.com/en-us/library/aa833404.aspx.

[6] RightScale, "Create a Staging Deployment ," http://support.rightscale.com/12-Guides/Lifecycle_Management/Performing_Upgrades_in_the_Cloud/02-Create_a_Staging_Deployment 2011.

[7] Innovartis, "Database Change Management Best Practices," http://www.dbghost.com/pdf/Innovartis_An_Automated_Approach_To_Do_Change_Mgt.pdf 2004.

[8] J. F. Roddick, "A Survey of Schema Versioning Issues for Database Systems," *Information and Software Technology*, vol. 37, no. 7, pp. 383-393, 1995.

[9] N. B. Ruparelia, "The History of Version Control," *ACM SIGSOFT Software Engineering Notes*, vol. 35, no. 1, pp. 5-9, January 2010.

[10] E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Communications of the ACM*, vol. 13, no. 6, pp. 377-387, June 1970.

[11] E. F. Codd, S. B. Codd, and C. T. Salley, "Providing OLAP to User-Analysts: An IT Mandate," E.F. Codd and Associates, Technical Report 1993.

[12] E. F. Codd, S. B. Codd, and C. T. Salley, "Beyond Decision Support," *Computerworld*, vol. 27, pp. 87-89, July 1993.

[13] R. Snodgrass and I. Ahn, "A Taxonomy of Time in Databases," in *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data (SIGMOD '85)*, 1985, pp. 236-246.

[14] R. Snodgrass and I. Ahn, "Temporal Databases," *IEEE Computer*, vol. 19, no. 9, p. 35, Sept. 1986.

[15] E. Rahm and P. A. Bernstein, "An Online Bibliography on Schema Evolution," *ACM SIGMOD Record*, vol. 35, no. 5, pp. 30-31, Dec. 2006.

[16] M. Golfarelli, J. Lechtenbörger, S. Rizzi, and G. Vossen, "Schema Versioning in Data Warehouses: Enabling Cross-Version Querying via Schema Augmentation," in *Data and Knowledge Engineering – Special Issue, WIDM 2004*, vol. 59, No. 2, 2006, pp. 435-459.

[17] J. L. Mitrpanont and S. Fugkeaw, "Design and Development of a Multiversion OLAP Application," in *Proceedings of the 2006 ACM Symposium on Applied Computing (SAC '06)*, 2006, pp. 493-497.

[18] H. J. Moon, C. A. Curino, A. Deutsch, C. Hou, and C. Zaniolo, "Managing and Querying Transaction-Time Databases under Schema Evolution," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 1, no. 1, pp. 882-895, August 2008.

[19] F. Wang, C. Zaniolo, and X. Zhou, "ArchIS: An XML-Based Approach to Transaction-Time Temporal Database Systems," *The International Journal of Very Large Databases*, vol. 17, no. 6, pp. 1445-1463, November 2008.

[20] B. Benatallah, "A Unified Framework for Supporting Dynamic Schema Evolution in Object Databases," in *Proceedings of the 18th International Conference on Conceptual Modeling (ER '99)*, London, UK, 1999, pp. 16-30.

[21] E. Franconi, F. Grandi, and F. Mandreoli, "Schema Evolution and Versioning: A Logical and Computational Characterisation," in *Selected papers from the 9th International Workshop on Foundations of Models and Languages for Data and Objects, Database Schema Evolution and Meta-Modeling (FoMLaDO/DEMM 2000)*, 2001, pp. 85-99.

[22] F. Grandi and F. Mandreoli, "A Formal Model for Temporal Schema Versioning in Object-Oriented Databases," *Data & Knowledge Engineering*, vol. 46, no. No. 2, pp. 123-167, August 2003.

[23] R. M. Galante, C. S. Santos, N. Edelweiss, and Á. F. Moreira, "Temporal and Versioning Model for Schema Evolution in Object-Oriented Databases," *Data and Knowledge Engineering*, vol. 53, no. 2, pp. 99-128, May 2005.

[24] S-W. Lee, J-H. Ahn, and H-J. Kim, "A Schema Version Model for Complex Objects in Object-Oriented Databases," *Journal of Systems Architecture: the EUROMICRO Journal*, vol. 25, no. 10, pp. 563-577, October 2006.

[25] J. Lu, C. Dong, and W. Dong, "A Equivalent Object-Oriented Schema Evolution Approach Using the Path-Independence Language," in *Proceedings of the 31st International Conference on Technology of Object-Oriented Language and Systems (TOOLS '99)*, 1999, pp. 212-217.

[26] Y-G. Ra and E. A. Rudensteiner, "A Transparent Schema-Evolution System Based on Object-Oriented View Technology," *IEEE Transactions on Knowledge and Data Engineering*, vol. 9, no. 4, July 1997.

[27] J. F. Roddick, F. Grandi, F. Mandreoli, and M. R. Scalas, "Beyond Schema Versioning: A Flexible Model for Spatio-Temporal Schema Selection," *Geoinformatica*, vol. 5, no. 1, pp. 33-50, Mar. 2001.

[28] F. Wang and C. Zaniolo, "An XML-Based Approach to Publishing and Querying the History of Databases," *World Wide Web Journal*, vol. 8, no. 3, pp. 233-259, 2005.

[29] Y. Velegrakis, R. J. Miller, and L. Popa, "Mapping Adaptation under Evolving Schemas," *Proceedings of the 29th international conference on Very Large Data Bases*, vol. 29, pp. 584-595, 2003.

[30] C. A. Curino, H. J. Moon, and C. Zaniolo, "Managing the History of Metadata in Support for DB Archiving and Schema Evolution," in *ER '08 Proceedings of the ER 2008 Workshops (CMLSA, ECDM, FP-UML, M2AS, RIGiM, SeCoGIS, WISM) on Advances in Conceptual Modeling: Challenges and Opportunities*, Berlin, Heidelberg, 2008, pp. 78-88.

[31] H. J. Moon, C. Curino, M. Ham, and C. Zaniolo, "PRIMA: Archiving and Querying Historical Data with Evolving Schemas," in *Proceedings of the 2009 International Conference on Management of Data (SIGMOD '09)*, Providence, RI, USA, 2009, pp. 1019-1022.

[32] H. J. Moon, C. A. Curino, and C. Zaniolo, "Scalable Architecture and Query Optimization for Transaction-Time Databases with Evolving Schemas," in *Proceedings of the 2010 International Conference on Management of Data (SIGMOD '10)*, 2010, pp. 207-218.

[33] C. De Castro, F. Grandi, and M. R. Scalas, "Schema Versioning for Multitemporal Relational Databases," *Information Systems*, vol. 22, no. 5, pp. 249-290, July 1997.

[34] H-C. Wei and R. Elmasri, "Schema Versioning and Database Conversion Techniques for Bi-temporal Databases," *Annals of Mathematics and Artificial Intelligence*, vol. 30, no. 1-4, pp. 23-52, 2000.

[35] C. A. Curino, H. J. Moon, and C. Zaniolo, "Graceful Database Schema Evolution: the PRISM Workbench," *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 761-772, Aug. 2008.

[36] C. A. Curino, H. J. Moon, and C. Zaniolo, "Automating Database Schema Evolution in Information System Upgrades," in *Proceedings of the 2nd International Workshop on Hot Topics in Software Upgrades*, 2009.

[37] C. A. Curino, H. J. Moon, M. Ham, and C. Zaniolo, "The PRISM Workwench: Database Schema Evolution without Tears," in *Proceedings of the 2009 IEEE International Conference on Data Engineering (ICDE '09)*, Washington, DC, USA, 2009, pp. 1523-1526.

[38] C. A. Curino, H. J. Moon, A. Deutsch, and C. Zaniolo, "Update Rewriting and Integrity Constraint Maintenance in a Schema Evolution Support System: PRISM++," *Proceedings of the VLDB Endowment*, vol. 4, no. 2, 2010.

[39] D. Sjøberg, "Quantifying Schema Evolution," *Information and Software Technology*, vol. 35, no. 1, pp. 35-44, 1993.

[40] P. A. Bernstein, "Applying Model Management to Classical Meta Data Problems," in *Proceedings of the First Conference on Innovative Data Systems Research (CIDR '03)*, 2003, pp. 209-220.

[41] P. A. Bernstein and S. Melnik, "Model Management 2.0: Manipulating Richer Mappings," in *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data (SIGMOD '07)*, 2007, pp. 1-12.

[42] R. Fagin, P. G. Kolaitis, A. Nash, and L. Popa, "Towards a Theory of Schema-Mapping Optimization," in *Proceedings of the 27th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS '08)*, 2008, pp. 33-42.

[43] R. Fagin, "Inverting Schema Mappings," *ACM Transactions on Database Systems (TODS)*, vol. 32, no. 4, p. Article 25, Nov. 2007.

[44] M. Arenas, J. Pérez, J. Reutter, and C. Riveros, "Inverting Schema Mappings: Bridging the Gap between Theory and Practice," in *Proceedings of the 36th International Conference on Very Large Databases (VLDB '09)*, Lyons, France, 2009.

[45] M. Stonebraker et al., "C-Store: a Column-Oriented DBMS," in *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB '05)*, 2005, pp. 553-564.

[46] D. J. Abadi, S. Madden, and N. Hachem, "Column-Stores vs. Row-Stores: How Different Are They Really?," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD '08)*, Vancouver, BC, Canada, 2008, pp. 967-980.

[47] Z. Liu, S. Natarajan, B. He, H. Hsiao, and Y. Chen, "CODS: Evolving Data Efficiently and Scalably in Column Oriented Databases," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 3, no. 2, pp. 1521-1524, 2010.

[48] H. Plattner, "A Common Database Approach for OLTP and OLAP Using an In-Memory Column Database," in *Proceedings of the 35th SIGMOD International Conference on Management of Data (SIGMOD '09)*, Providence, RI, USA, 2009, pp. 1-2.

[49] M. Grund et al., "HYRISE: a Main Memory Hybrid Storage Engine," *Proceedings of the VLDB Endowment*, vol. 4, no. 2, pp. 105-116, November 2010.

[50] T. Haapasalo, I. Jaluta, S. Sippu, and E. Soisalon-Soininen, "Concurrency Control and Recovery for Multiversion Database Structures," in *Proceeding of the 2nd PhD Workshop on Information and Knowledge Management (PIKM '08)*, 2008, pp. 73-80.

[51] T. Haapasalo, S. Sippu, I. Jaluta, and E. Soisalon-Soininen, "Concurrent Updating Transactions on Versioned Data," in *Proceedings of the 2009 International Database Engineering & Applications Symposium (IDEAS '09)*, 2009, pp. 77-87.

[52] T. Haapasalo, I. Jaluta, B. Seeger, S. Sippu, and E. Soisalon-Soininen, "Transactions on the Multiversion B+-Tree," in *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology (EDBT '09)*, 2009, pp. 1064-1075.

[53] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer, "An Asymptotically Optimal Multiversion B-tree," *The International Journal on Very Large Data Bases (VLDB Journal)*, vol. 5, no. 4, pp. 264-275, Dec. 1996.

[54] T. Dumitraş and P. Narasimhan, "No Downtime for Data Conversions: Rethinking Hot Upgrades," Parallel Data Laboratory, Carnegie Mellon University, Pittsburgh, PA, CMU-PDL-09-106, 2009.

[55] T. Dumitraş and P. Narasimhan, "Why Do Upgrades Fail and What Can We Do About it?: Toward Dependable, Online Upgrades in Enterprise Systems," in *Middleware '09: Proceedings of the 10th*

ACM/IFIP/USENIX International Conference on Middleware, 2009.

[56] R. Chatterjee, G. Arun, S. Agarwal, B. Speckhard, and R. Vasudevan, "Using Data Versioning in Database Application Development," in *Proceedings of the 26th International Conference on Software Engineering (ICSE'04)*, 2004, pp. 315-325.

[57] M. Callaghan, "Online Schema Change for Mysql," Facebook, http://www.facebook.com/notes/mysql-at-facebook/online-schema-change-for-mysql/430801045932 2010.

[58] S. Noach, "oak-online-alter-table," http://openarkkit.googlecode.com/svn/trunk/openarkkit/doc/html/oak-online-alter-table.html 2008.

[59] M. Lee and A. J. Offutt, "Algorithmic Analysis of the Impact of Changes to Object-Oriented Software," in *IEEE International Conference on Software Maintenance*, Monterrey, CA, 1996, pp. 171-184.

[60] M. Lee, "Change Impact Analysis of Object-Oriented Software," George Mason University ISSE Dept., Fairfax, VA, Technical Report ISSE-TR-99-06, 1998.

[61] B. Ryder and F. Tip, "Change Impact Analysis for Object-Oriented Programs," in *Proc. SIGPLAN/SIGSOFT Workshop Program Analysis for Software Tools and Eng. (PASTE '01)*, 2001, pp. 46-53.

[62] X. Ren, B. G. Ryder, M. Stoerzer, and F. Tip, "Chianti: a Change Impact Analysis Tool for Java Programs," in *Proceedings of the 27th International Conference on Software Engineering (ICSE '05)*, 2005, pp. 664-665.

[63] N. Dor, T. Lev-Ami, S. Litvak, M. Sagiv, and D. Weiss, "Customization Change Impact Analysis for ERP Professionals via Program Slicing," in *Proceedings of the 2008 international symposium on Software testing and analysis (ISSTA '08)*, Seattle, WA, USA, 2008, pp. 97-108.

[64] P. Tonella, "Using a Concept Lattice of Decomposition Slices for Program Understanding and Impact Analysis," *IEEE Transactions on Software Engineering*, vol. 29, no. 6, pp. 495 - 509, June 2003.

[65] K. S. Herzig, "Capturing the Long-Term Impact of Changes," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2 (ICSE '10)*, 2010, pp. 393-396.

[66] H. Xiao, J. Guo, and Y. Zou, "Supporting Change Impact Analysis for Service Oriented Business Applications," in *Proceedings of the International Workshop on Systems Development in SOA Environments (SDSOA '07)*, 2007, p. 6.

[67] M.K. Abdi, H. Lounis, and H. Sahraoui, "Predicting Change Impact in Object-Oriented Applications with Bayesian Networks," in *33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC '09)*, 2009, pp. 234-239.

[68] B. Wall, N. Richter, and R. Angryk, "Creating Concept Hierarchies in an Information Retrieval System," in *Proceeding of the 5th IEEE International Conference on Data Mining (ICDM '05), Workshop on Foundations of Semantic Oriented Data and Web Mining*, Houston, TX, USA, 2005,

pp. 99-105.

[69] B. Wall, N. Richter, and R. Angryk, "Generating Concept Hierarchies from User Queries," in *Data Mining: Foundations and Practice*, T. Y. Lin et al., Eds. Berlin, DE: Springer-Verlag, 2008, pp. 423-441.

7. Appendix

The following are details of the planned implementation of the versioning framework. As mentioned previously, the versioning system targets a MySQL database, so modifications and additions to SQL are designed to be compatible with MySQL syntax.

Client applications that connect to a database schema must be able to specify a version; this version is an attribute of the session, so setting it will be implemented as an extension to the standard SQL statement for setting session attributes. The syntax is shown in Figure 9.

```
<SQL session statement> ::=
   <set catalog statement> |
   <set schema statement> |
   … |
   <set version statement>

<set version statement> ::=
   SET VERSION <version specification>

<version specification> ::=
   DEFAULT |
   <unsigned integer> |
   <unsigned integer> <period> <unsigned integer>
```

Figure 9 – SQL to Select Version

```
<SQL schema definition statement> ::=
   <schema definition> |
   <table definition> |
   … |
   <version definition>

<version definition> ::=
   CREATE VERSION <new version specification>

<new version specification> ::=
   NEXT |
   TRUNK <period> NEXT |
   HEAD <period> NEXT
```

Figure 10 – SQL to Create Version

Specifying only the major version number implies a value of NULL for the minor version number. Any session that does not specify a version will use the trunk version by default. This provides backward compatibility for applications that are not version-aware.

An application will be able to determine the current version of the session via two new session variables, *major_version* and *minor_version*. *major_version* is a positive integer, and *minor_version* is either NULL or a positive integer. The values of these variables can be obtained by querying the special view *information_schema.session_variables* or by using *SHOW VARIABLES*.

New versions will be created using a new SQL statement with the syntax shown in Figure 10. A *CREATE VERSION NEXT* statement will generate an error if a head version already exists. A *CREATE VERSION HEAD.NEXT* statement will generate an error if a head version does not exist. Creating a new version implicitly changes the session to that version, so the database client can query the *major_version* and *minor_version* session variables to determine what version numbers have been assigned to the newly created version.

```
<SQL schema manipulation statement> ::=
   <drop schema statement> |
   <drop table statement> |
   … |
   <drop version statement>

<drop version statement> ::=
   DROP VERSION <drop version specification>

<drop version specification> ::=
   <unsigned integer> |
   <unsigned integer> <period> <unsigned integer>
```

Figure 11 – SQL to Drop Version

```
<SQL merge statement> ::=
   MERGE VERSION
```

Figure 12 – SQL Statement to Merge Version

Support will be added to discard versions using a new SQL statement with the syntax shown in Figure 11. A *DROP VERSION* statement that attempts to remove a non-existent version or the trunk version will generate an error. If the head version is dropped and there are branches created from it, those branches are also discarded.

A branch will be merged back into its source version using the SQL statement with the syntax shown in Figure 12. When a branch is merged back into its source version, the version is implicitly changed to the source version, as if a *SET VERSION* statement had been executed. The exception is merging the head into the trunk; after the merge is complete, the version number is unchanged. This is now the trunk version rather than the head though.

It will be possible to modify a table so that any data changes to that table will not be included when the branch is merged using the extension to the SQL ALTER TABLE statement shown in Figure 13. This command will be ignored if executed in the trunk version.

```
<SQL alter table statement> ::=
   ALTER [ONLINE | OFFLINE] [IGNORE] TABLE
         <tbl_name> <alter specification>
                    [, <alter specification] …

<alter specification> ::=
   <table_options> |
   ADD [COLUMN] <col_name> <col specification> |
   … |
   DISABLE MERGE
```

Figure 13 - SQL to Disable Data Merge for Table

```
catalog_name   VARCHAR(512),
schema_name      VARCHAR(64) NOT NULL,
major_version    INT NOT NULL,
minor_version    INT,
created TIMESTAMP NOT NULL,
merged           TIMESTAMP,
dropped TIMESTAMP
```

Figure 14 – New Version Metadata

Metadata about the versions will be available using the new view *information_schema.versions*. It will include the columns shown in Figure 14. The *merged* column indicates when the branch has been merged with its source version. For minor versions, their changes will be merged into their source branch. For major versions, the trunk will be merged into the head, at which time that version becomes the trunk. After a branch has been merged, that version is no longer accessible. A *SET VERSION* or *DROP VERSION* statement that specifies a version that has already been merged will generate an error.

There will always be at least one entry in the *versions* view for each distinct *schema_name*, with *major_version* 1, *minor_version* NULL, and *merged* and *dropped* NULL. Given the constraints of the versioning system, there will be at least one and at most two versions with a NULL minor version and NULL merged and dropped time.

The existing *information_schema.schematas* view will also be augmented with a new column, *major_version*. This NOT NULL integer contains the version number of the trunk version for each schema.