# Safe On-the-fly Relational Schema Evolution


## ABSTRACT

Database applications frequently undergo schema changes. To change the schema, the database has to be shut down and migrated to the new version, or run in a mixed-mode that supports old and new clients. The first alternative is problematic for applications that cannot tolerate downtime. The second alternative raises consistency and performance degradation issues. In this paper we present an approach for on-the-fly schema evolution that avoids downtime, ensures consistency, and is transparent to clients. We first study schema evolution and database usage in three popular open source applications over a long period of time. Next, we model the frequently-used schema changes and queries identified in our empirical study in a formalism (calculus and semantics) that extends the SPJR algebra with version information, and use the formalism to construct a proof that our approach is safe and transparent. We then use the formalism to implement support for safe on-the-fly schema evolution in SQLite, a popular open source SQL engine. Finally, we evaluate our approach using real-world schema changes and database usage scenarios. Our experiments indicate that, while certain schema changes result in a small performance penalty, this reduced-performance operation is short-lived. In conclusion, this paper shows that applications can enjoy safe, almost-instantaneous schema upgrades with little performance cost.


## 1. INTRODUCTION

Database evolution is a fact of life. DBMS users, large or small, are under great pressure to update their schemas and database applications to fix bugs and add new features to satisfy their customers. Unfortunately, the requirement that updates be released often and applied quickly is at odds with providing a seamless, uninterrupted service. For example, for high-availability applications, where 24/7 service is essential, stopping the application to upgrade the database is unacceptable; for desktop applications, stopping/restarting the program or the OS is disruptive to users.

To solve this problem for large, enterprise databases, researchers and practitioners have long investigated, and provided solutions for, online database reorganization [27]. However, we are currently missing online upgrades facilities for small-scale databases used in embedded, mobile, or desktop systems. To that end, in this paper we present an approach for enabling on-the-fly schema updates for SQLite, a server-less, zero-config SQL engine [12]. SQLite is extremely popular: it is used in operating systems (e.g., Mac OS X, Solaris 10, OpenSolaris), user space applications (e.g., Apple Mail, Mozilla, the McAfee antivirus suite), and mobile platforms such as Google Android or Symbian. An estimate by the SQLite development team puts the number of SQLite installations upwards of 200 million.

Though we focus on SQLite, our formalism and safety proof are applicable to, and our implementation techniques can be leveraged for, on-the-fly schema evolution support in any relational DBMS. To illustrate this, consider a July 2009 survey on databases supporting online reorganization [27]; the survey identifies three key requirements for such systems. These requirements are explicit goals of our work: (1) *correctness*, i.e., "users must be able to query and update correctly. Similarly, the actions of reorganization must be correct."; (2) *performance*, i.e., "the degradation (if any) of users' performance must be tolerable" and "[the reorganization process] must eventually complete its work."; (3) *error tolerance*, i.e., "data must be recoverable during online reorganization." We now proceed to providing an overview of our system, and the steps we had to take toward fulfilling these key requirements.

To address the tension between applying updates and sustaining continuous service, we introduce a new technique that permits on-the-fly schema evolution via lazy updates to relational databases. Our system allows a database client to switch to a new schema at runtime, without having to shut down the database and compromise service. The technique works transparently to clients: all they have to do is send the server a command `UPDATEDB` specifying the new schema. Our implementation will then initiate a lazy update process, but return the control immediately to the client, so the client can actually perform database operations (queries or updates), right away, at the new schema. The client is not aware of the lazy update going on in the background, except for the fact that an operation might take longer than what it would take if the database was started directly at the new schema. This overhead is transient, though; after the lazy update process has finished, the database operations will run at full speed and the system is ready for another update.

In constructing our on-the-fly update system, we started with a study on schema evolution and query frequency. The study looks at the evolution of three popular open source applications, Mozilla, Monotone and Vienna, and identifies how databases evolve and what kind of queries are most

popular in practice. We found that, over the period we analyzed, there were 42 schema updates in Mozilla, 11 in Monotone, and 7 in Vienna; these updates consist of 120 table or attribute changes in Mozilla, 39 in Monotone and 10 in Vienna. The complete results of our study are presented in Section 2.3.

The problem with allowing a client to start operations that assume a new schema, while the database is still at the old schema and partially converted, is that we can easily get into an unsafe situation. For example, if the schema update adds a new table or a new attribute, the client might ask for it, even though it still doesn't exist in the database, leading to a runtime error. To prevent such errors, we model schema changes, database operations and lazy updates formally, as a calculus that extends SPJR algebra. In Section 3 we present the calculus and a safety proof that basically guarantees that database operations are always safe, even though the database is in a partially-updated state.

The calculus and safety conditions form the basis for implementing on-the-fly updates in SQLite, while keeping the actual update mechanisms transparent to the clients. In Section 4 we present a sample scenario of how a client can use our system in practice. In Section 5 we describe the implementation techniques we used to extend SQLite with on-the-fly update support. To evaluate our implementation (Section 6) we use real-world schema updates and queries from Mozilla, Monotone and Vienna; based on changes and usage patterns we observed in these applications, we construct a variety of benchmarks meant to quantify the impact of lazy on-the-fly updates on the clients using the database. We found that the performance, memory and disk overheads are low, and after signaling a schema update, control returns to the client in less than 400 milliseconds.

In short, this paper makes the following contributions:

- A formalism for reasoning about, and guaranteeing the safety of, lazy on-the fly updates to relational DBMSs.
- An implementation that extends SQLite with support for of lazy, on-the-fly updates.
- An evaluation of our approach using real-world schema evolution and query data.

# 2. MOTIVATION

In this section we describe the motivating factors that lead us to pursue on-the-fly database updates, the rationale for choosing SQLite as a research vehicle, and the empirical findings that drove our selection of schema changes and database operations supported in our system.

## 2.1 Rationale For On-the-fly Updates

The primary motivation of this work is to support online upgrades to database systems. Many large-scale information systems, from banking applications to manufacturing to global services cannot afford to shut down services, or else their business is threatened. Many more small-scale systems, such as desktop or mobile applications use "light" DBMSs to store application data; allowing online upgrades to these DBMSs would provide a better user experience.

The requirement that these applications run continuously is at odds with the need to frequently update them, to add new features and fix bugs. Many solutions exist already for updating "standard" applications written in C, C++, and Java. For example, dynamic software updating systems such as Ginseng [19] and Jvolve [28], or runtime modification systems such as PROSE [20] allow on-the-fly updates to code and in-memory data; some of these systems have proved to be effective for supporting several years' worth of runtime evolution. However, because of their focus on updating code and in-memory state, dynamic updating systems are insufficient for performing online upgrades to systems that require database updates.

For example, in the update from Firefox 3.0b2 to 3.0b3, the attribute `user_title` was deleted from table `moz_history`. In the update from Mozilla 1.9a7 to 1.9a8, a new table `moz_downloads` was added. In the update from Mozilla 1.9a4 to 1.9a5, two new attributes, `dateAdded` and `lastModified` were added to table `moz_bookmarks`. If we use a dynamic updating system for Firefox/Mozilla, we can update the code, but the information stored in the database remains at the old version, which will lead to incompatibility. Our system gives the application the option to conduct an on-the-fly database update whenever necessary, though we must emphasize that dynamic software updating (e.g., on-the-fly code updating or modification) is outside the scope of this work.

We now present two example of updates where our system could be useful in practice:

1. *On-the-fly updates.* Suppose we want to update Firefox 2.0 to Firefox 2.1 on-the-fly, and the database schemas assumed by the two versions are `schema_20` and `schema_21`, respectively. We can use a dynamic software updating tool to update the code from version 2.0 to version 2.1. Once the code update has completed, Firefox sends an `UPDATEDB(schema_21)` to our system, and from that point on, it can safely use (query or modify) the database at the new schema. Based on anecdotal evidence of the time to complete a typical dynamic code update (less than 5 ms [19]), and adding our on-the-fly schema update time (Section 6), the entire process takes less than 400ms—unlikely to be disruptive to the user.
2. *Immediate off-line updates.* Firefox stores user data in a "profile"—a database file—that, depending on settings, can exceed several GBs. Suppose we updated the Firefox 2.0 code to Firefox 2.1 off-line. If we use a database migration mechanism to convert the profile from `schema_20` and `schema_21` upon the first start at version 2.1, converting several GBs will take a considerable amount of time, rendering Firefox unresponsive/unusable until migration is complete. Using our system, we can perform the off-line code update and, when version 2.1 starts, place an `UPDATEDB(schema_21)` call which will allow Firefox to immediately use the profile at the new schema, without delay.

## 2.2 Why SQLite?

We implemented on-the-fly database updates by extending SQLite. SQLite is server-less, implemented as a library,

| SMO | Mozilla | | Monotone | | Vienna | |
|---|---|---|---|---|---|---|
| | count | % | count | % | count | % |
| ADD COLUMN | 58 | 48 | 11 | 28 | 10 | 100 |
| DROP COLUMN | 30 | 25 | 9 | 23 | - | - |
| CREATE TABLE | 20 | 17 | 9 | 23 | - | - |
| DROP TABLE | 4 | 3.3 | 8 | 20 | - | - |
| RENAME TABLE | 5 | 4.2 | 1 | 2.6 | - | - |
| RENAME COLUMN | 2 | 1.7 | 1 | 2.6 | - | - |
| COL. TYPE CHG | 1 | 0.8 | - | - | - | - |

**Table 1: Schema evolution in our test applications.**

| Command | Mozilla | | Monotone | | Vienna | |
|---|---|---|---|---|---|---|
| | count | % | count | % | count | % |
| SELECT | 291 | 43 | 80 | 43 | 87 | 22 |
| INSERT | 210 | 31 | 45 | 24 | 57 | 14 |
| UPDATE | 91 | 13 | 21 | 11 | 190 | 47 |
| DELETE | 58 | 9 | 22 | 12 | 46 | 11 |
| PRAGMA | 20 | 3 | 13 | 7 | - | - |
| COMMIT TRANS. | 2 | 0.3 | - | - | 8 | 2 |
| BEGIN TRANS. | 1 | 0.15 | - | - | 8 | 2 |
| END TRANS. | 1 | 0.15 | - | - | - | - |
| ROLLBK TRANS. | 1 | 0.15 | - | - | - | - |
| REPLACE | - | - | 2 | 1.08 | - | - |
| VACUUM | - | - | 1 | 0.54 | 8 | 2 |

**Table 2: SQL command frequency.**

and meant to be linked together with the client application. The motivation for choosing SQLite was twofold. First, and most importantly, SQLite is ubiquitous in the construction of mobile and desktop applications and operating systems, as mentioned in Section 1. The second reason for using SQLite is the small code base (the SQL engine consists of around 64,000 lines of C code) which makes it an excellent platform for research.

## 2.3 Schema Evolution And SQL Usage Study

When constructing our on-the-fly updating system, we took a pragmatic approach. The development of our calculus and implementation was driven by answers to two questions:

1. What are the most frequent ways in which schemas change?
2. What are the most frequent SQL commands used in database applications that evolve?

Previous works by Curino et al. [7] and Sjøberg [26] have looked into schema evolution for Wikipedia (4.5 years) and for a health management system (1.5 years), respectively. However, these systems do not use SQLite as a database back-end, so we might not get an accurate answer to our questions. Therefore, to understand how SQLite-based relational databases evolve and are used in practice, we performed our own analysis on three popular open source programs: Mozilla, Monotone, and Vienna. Mozilla (`http://www.mozilla.org`) is a large open source project, that contains, among others, the Firefox browser and the Thunderbird email client. Mozilla uses SQLite to store the browsing history, input forms, cookies, etc. Monotone (`http://www.monotone.ca/`) is a version control system; it uses SQLite to store file revisions, deltas, and branch information. Vienna (`http://www.vienna-rss.org`) is a popular RSS/Atom newsreader for Mac OS X; it uses SQLite to store news folders and messages.

Our analysis covers each official release in the period November 2005–May 2009 for Mozilla, April 2003–May 2009 for Monotone, and July 2005–September 2009 for Vienna. This corresponds to the entire lifetime for Monotone and Vienna, and the period since SQLite was introduced in Mozilla. To answer question (1), in Table 1 we summarize the results, as Schema Modification Operators (SMOs) [25, 8]. For each application, we counted the SMOs over the time span we studied; for each SMO we provide both the total count and percentages. For example, there were 58 `ADD COLUMN` changes in Mozilla, which constitute 49% of the total number of changes for that application. As we can see, the most frequent operations alter table schemas to add or delete attributes (`ADD COLUMN` and `DROP COLUMN`); table creation, deletion and renaming are less frequent; column renaming and column type change are very rare.

The answer to question (2) is presented in Table 2. For all source code files containing database operations, we measured the number of times an SQL command appears in each post-update version; we excluded commands such as `CREATE TABLE`, `DROP TABLE`, and `ALTER TABLE` because these are already reflected in the SMO figures. The first column represents the SQL command; columns 2/4/6 show the count, i.e., number of times, that command appears in the post-update application source code for Mozilla, Monotone, and Vienna, while columns 3/5/7 show the relative frequency of that command relative to the total count. While the absolute count values are interesting on their own, what matters most is the relative frequency. As we can see, the most frequent commands are `SELECT`, `INSERT`, `UPDATE`, and `DELETE`. Our formal calculus only models these commands, and our implementation supports safe queries that are based on these commands. The remaining commands (from `PRAGMA` to `VACUUM`) are used infrequently, hence for the purpose of this paper we do not support them.

In prior work [3] we performed a schema evolution study on Mozilla and Monotone over the same period as for our schema evolution/frequency count study. Therefore, the Mozilla and Monotone data in Table 1 is not a contribution of this paper, but we consider that presenting it is useful nevertheless, to quantify how SQLite-based applications evolve their schemas over time. However, everything else, i.e., the Vienna data in Table 1 and the SQL command frequency data in Table 2 is new in this paper.

## 2.4 The Need For Safety

In prior work, we showed that schema migration in Mozilla is sometimes ad-hoc, i.e., the application does not check the schema version in the database prior to executing queries [3]. Without migration after updating the application, the new queries will run against the old schema, which can lead to data loss or runtime errors. In our approach, new queries always run against the new schema and safety is guaranteed. In Table 3 we show the query failure rates (in the absence of migration or updates) for the Mozilla files exhibiting fre-

| File | Failure rate (%) | | |
|---|---|---|---|
| | min | avg | max |
| nsNavBookmarks.cpp | 6 | 46 | 76 |
| nsAnnotationService.cpp | 20 | 51 | 100 |
| nsUrlClassifierDBService .cpp | 16 | 28 | 33 |
| nsCookieService.cpp | 25 | 34 | 40 |
| nsDownloadManager.cpp | 17 | 27 | 33 |

**Table 3: Query failure rate in Mozilla in the absence of migration or updates.**

quent schema changes. The numbers show the percentage of new queries that would fail if run against the old schema. For example, for nsNavBookmarks.cpp, there were 8 updates (schema changes) for the period we analyzed. If we tried to execute the post-update queries on the pre-update database, at least 6%, at most 76% (on average 46%) of the new queries would fail; the minimum, maximum and averages are computed across all 8 updates. As we can see, the situation is more dire for nsAnnotationService.cpp where, for one of the updates, the 100% figure indicates that *all* post-update queries would fail if run against the pre-update schema.

A convenient aspect of our SQLite-based model is that we need not perform any query rewriting, because of two combined factors: (1) the query text is embedded in the application code (C, C++, Java, etc.), and (2) the schema update is client-initiated. Following a dynamic update to the application code (not the scope of this paper), the code will contain the new query versions; at this point, the client sends an `UPDATEDB` command to SQLite, which will switch the schema to the new version, hence the queries and the schema are in-sync.

# 3. THEORY OF SAFE UPDATES

In this section we present a calculus and semantics for reasoning about lazy updates and proving that they are safe. Our calculus extends SPJR algebra [2] with lazy updates and modification operators; its construction was directly inspired by the most frequent queries and schema changes we observed in practice, as reported in Section 2.3. The main theoretical result is that querying a database that was created and populated at version 1 and lazily updated from version 1 to version 2 yields results indistinguishable from querying a database that was created and populated entirely at version 2.

## 3.1 Syntax

The syntax for our calculus is presented in Figure 1; it is basically an untyped[1] extension of SPJR algebra.

*Tuples, relations, databases.* Values $v$ can be either attribute names $A$ or constants $d$. Each tuple (record) $U$ consists of attributes $A_i$ with values $d_i$, and a version $\nu$, explained later. In the actual implementation, attribute names are not stored with each tuple, but we use named attributes in our calculus for clarity. Each relation (table) $T$ has an associated name $R$, a schema $\vec{A}$, and an instance $\vec{U}$. A relation schema $\vec{A}$ is a vector containing the table's attributes. An instance $\vec{U}$ is a vector containing tuples $U$. Each database instance $I$ consists of a vector of tables $\vec{T}$.

[1]Though our implementation safely supports attribute type changes, we keep the calculus untyped for simplicity.

| | | | |
|---|---|---|---|
| *Versions* | $\nu$ | ::= | $1 \mid 2$ |
| *Values* | $v$ | ::= | $A \mid d$ |
| *Records* | $U$ | ::= | $\langle\langle A_1 : d_1, \ldots, A_n : d_n\rangle, \nu\rangle$ |
| *Named relations* | $N$ | ::= | $R : \langle\vec{A}\rangle$ |
| *Relation instances* | $T$ | ::= | $\langle N, \langle\vec{U}\rangle\rangle$ |
| *DB schemas* | $\mathcal{R}$ | ::= | $\langle\vec{N}\rangle$ |
| *DB instances* | $I$ | ::= | $\langle\vec{T}\rangle$ |
| *Queries* | $Q$ | ::= | $\langle A : d\rangle \mid R$<br>$\mid \sigma_{A=d}(Q) \mid \sigma_{A=B}(Q)$<br>$\mid \pi_{\vec{A}}(Q)$<br>$\mid Q \times Q'$<br>$\mid Q \bowtie Q'$<br>$\mid \rho_{A_1/B_1,\ldots,A_N/B_n}(Q)$<br>$\mid \mathrm{ins}_{R,\langle A_1:d_1,\ldots,A_n:d_n\rangle}(Q)$<br>$\mid \mathrm{repl}_{R,A:a,B:b}(Q)$<br>$\mid \mathrm{del}_{R,A:a}(Q)$ |

**Figure 1: Syntax.**

*Versions.* The key idea behind lazy updates is to allow old and new tuples to coexist, and when a query comes in, perform a safety check to determine whether the query can be answered immediately or must wait until the lazy update has completed for all the tuples involved in the query. A function $z()$, described later, makes this determination. The version $\nu$ associated with each tuple indicates whether that tuple is stored using the old schema ($\nu = 1$) or the new schema ($\nu = 2$). For example, if an update changes the schema of a relation from $\langle A_1, A_2\rangle$ to $\langle A_1, A_2, A_3\rangle$, and the pre-update relation instance is

$$\{\langle\langle A_1 : 10, A_2 : 20\rangle, 1\rangle, \langle\langle A_1 : 20, A_2 : 40\rangle, 1\rangle\}$$

then a possible post-update scenario is a partially-converted table with two tuples at two different versions:

$$\{\langle\langle A_1 : 10, A_2 : 20\rangle, 1\rangle, \langle\langle A_1 : 20, A_2 : 40, A_3 : 30\rangle, 2\rangle\}$$

From this point on, we will use the terms "old"/"version 1", as well as "new"/"version 2" interchangeably. Note that, for simplicity but without loss of generality, we assume a single update, from version 1 to version 2, in our calculus and exposition. In practice, though, a database can undergo multiple, sequential updates, hence in our implementation the "new" and "old" schemas and tuples always refer to the latest version and the version prior to it.

*Queries.* Queries $Q$ consist of the usual SPJR constructs, i.e., constants $\langle A : d\rangle$, or relation names $R$; selecting tuples where an attribute $A$ is equal to constant $d$, i.e., $\sigma_{A=d}$, and selecting tuples where two attributes are equal, i.e., $\sigma_{A=B}$. Projections $\pi_{\vec{A}}$ extract those attributes specified in $\vec{A}$. The Cartesian product $Q \times Q'$ and natural join $Q \bowtie Q'$ have their standard meaning. Renaming $\rho_{A_1/B_1,\ldots,A_N/B_n}$ simply re-labels existing attributes $B_1, \ldots, B_n$ to produce

a tuple with attribute names $A_1, \ldots, A_n$. Our empirical study (Section 2.3) has shown that, in practice, the most frequently used SQL commands are `SELECT`, `INSERT`, `UPDATE`, and `DELETE`. We have already modeled `SELECT` using $\sigma_{A=d}$ and $\sigma_{A=B}$, and now proceed to showing how we model the remaining three. The addition, replacement and deletion operations are normally known as *updates*, rather than *queries*. To simplify the presentation, however, we categorize them as $Q$'s in our language, though their semantics, as presented in Figure 3 and Section 3.3, are quite different from the semantics of operations that don't modify the database. We can add a new tuple $\langle A_1 : d_1, \ldots, A_n : d_n \rangle$ to a relation $R$ using $\mathbf{ins}_{R,\langle A_1:d_1,\ldots,A_n:d_n\rangle}$, which corresponds to `INSERT`. We can modify a tuple whose attribute $A$ has value $a$ so that its attribute $B$ contains value $b$ using $\mathbf{repl}_{R,A:a,B:b}$, which corresponds to `UPDATE` in SQL; we refer to this modification operator as "**repl**", rather than "**upd**" or similar because the update operation in our system actually refers to a schema upgrade. Finally, we can delete all tuples whose attribute $A$ has value $a$ using $\mathbf{del}_{R,A:a}$.

## 3.2 Auxiliary Definitions

$$
\begin{aligned}
erase(U) &= \langle A_1 : d_1, \ldots, A_n : d_n \rangle \mid \\
&\quad U = \langle \langle A_1 : d_1, \ldots, A_n : d_n \rangle, \nu \rangle \\
U^2 &= \langle \langle A_1 : d_1, \ldots, A_n : d_n \rangle, 2 \rangle \mid \\
&\quad U = \langle \langle A_1 : d_1, \ldots, A_n : d_n \rangle, \nu \rangle \\
T^2 &= \langle N, \langle {U_1}^2, \ldots, {U_n}^2 \rangle \rangle \mid T = \langle N, \langle U_1, \ldots, U_n \rangle \rangle \\
I^2 &= \langle {T_1}^2, \ldots, {T_m}^2 \rangle \mid I = \langle T_1, \ldots, T_m \rangle \\
R_\Delta &= (R^1 \setminus R^2) \cup (R^2 \setminus R^1) \\
z(\vec{A}, \langle \vec{U} \rangle) &= \langle \vec{U} \rangle,\ if\ \vec{A} \cap R_\Delta = \emptyset \\
&\quad \mid \langle \vec{U^2} \rangle,\ otherwise
\end{aligned}
$$

**Figure 2: Auxiliary definitions.**

To simplify the semantics definition, we assume that queries $Q(I)$ are well-formed. For example, if $Q$ is a projection, we assume that the attributes specified in $Q$ appear in $I$'s schema, or if $Q$ is an insertion into relation $R$, the to-be-inserted tuple has the same type as $R$'s schema. In practice, we rely on SQLite's parser for detecting queries that are not well-formed.

Before proceeding to specifying the semantics of queries, we need to introduce several auxiliary definitions (Figure 2). The function $erase(U)$ simply "erases" $U$'s version. The function $U^2$ returns a tuple with the same attribute values as $U$, and version set to 2; by extension, $T^2$ and $I^2$ return table and database instances whose version fields are all 2's. $R_\Delta$ is a shorthand for the symmetric difference between the old schema, $R^1$, and the new schema, $R^2$; in effect, $R_\Delta$ contains all the attributes that were deleted or added.

The *lazy update* function $z(\vec{A}, \langle \vec{U} \rangle)$ is the crucial mechanism for safety; it ensures that all accesses to a table $R$ with schema $\vec{A}$ and contents $\langle \vec{U} \rangle$ are performed at version 2. We have two cases:

1. If the update does not alter the schema $\vec{A}$, i.e., $\vec{A} \cap R_\Delta = \emptyset$, then we can safely access any tuple in $\vec{U}$ regardless of its $\nu$, as the contents of data in $\vec{U}$ is the same in both version 1 and version 2.
2. If the update alters $\vec{A}$, i.e., $\vec{A} \cap R_\Delta \neq \emptyset$, then we have to "stall" the query and force an eager update, i.e., convert all the tuples whose version is still 1 to version 2, hence the result of $z(\vec{A}, \langle \vec{U} \rangle)$ is $\langle \vec{U^2} \rangle$.

As we will describe in Section 5.3, our implementation uses the $\vec{A} \cap R_\Delta$ check to optimize log propagation and to avoid calling $z(\vec{A}, \langle \vec{U} \rangle)$ for each query; this speeds up query processing significantly, while still preserving safety.

Note that we do not tack versions $\nu$ onto table schemas $\langle \vec{A} \rangle$ or databases schemas $\mathcal{R}$. The rationale is that table and database schemas are always at the newest version. That is, when an update is signaled, the table and database schemas are changed immediately, as explained in Section 5.2. This simplifies our formalism without affecting safety, and, in practice (Section 6.2) is a fast operation that can be executed in-line when the update is signaled and before returning control to the client.

## 3.3 Semantics

We now proceed to explaining the query semantics in our calculus; the semantics is defined in Figure 3. (CONST) and (REL-INST) are non-recursive queries that return a constant, and a relation instance, respectively. The selection and projection rules — (SELECT-1), (SELECT-2), and (PROJECT) — first execute the remainder of the query $Q$ recursively, then choose the qualifying tuples/attributes, "strip" the version information for all tuples and add version 2, i.e., the format expected by the client that places the query. Rules for Cartesian product (CROSS-PROD), natural join (JOIN) and renaming (RENAME) also have the usual semantics, and all the resulting tuples have version 2. The (INSERT), (DELETE), and (REPLACE) queries require an $R$ argument, i.e., the relation where tuples are inserted into, deleted from, or updated. The semantics first "splits" the database instance into $I'$ and $T$ (where $T$ is $R$'s table instance), then constructs $T'$, the new version of $T$, and finally "glues together" the result as $\{I', T'\}$. For (DELETE) and (REPLACE), $\vec{U_{bf}}$ represents all the tuples in $R$ before the deletion/insertion; for (DELETE), $\vec{U_{af}}$ represents all the tuples in $R$ after the deletion. Note that (INSERT), (DELETE), (REPLACE), just like all the other queries, are purely functional, i.e., they cannot change the input arguments, $I$ or $R$. In practice, though, the way we implement these queries is by modifying the relation $R$ (and therefore the database instance $I$) in-place.

## 3.4 Safety Proof

We can now proceed to stating and proving the update safety theorem.

**Theorem 3.1** (Update safety)**.** *Let $I$ be a database instance that may contain version 1 tuples, $I^2$ be the corresponding instance where all tuples are at version 2, and $Q$ a query. Lazy updates are safe, that is:*

$$erase([\![Q]\!](z(I))) = erase([\![Q]\!](I^2))$$

$$
\begin{array}{lrcl}
\text{(CONST)} & [\![\langle A : d\rangle]\!](I) & = & \{\langle A : d\rangle\} \\
\text{(REL-INST)} & [\![R]\!](I) & = & I(R) \\
\text{(SELECT-1)} & [\![\sigma_{A_i=d}(Q)]\!](I) & = & \{U^2 \mid U \in \mathit{erase}([\![Q]\!](I)) \wedge U = \langle A_1 : d_1, \ldots, A_i : d, \ldots, A_n : d_n\rangle\} \\
\text{(SELECT-2)} & [\![\sigma_{A_i=A_j}(Q)]\!](I) & = & \{U^2 \mid U \in \mathit{erase}([\![Q]\!](I)) \wedge U = \langle A_1 : d_1, \ldots, A_i : d, \ldots, A_j : d, \ldots, A_n : d_n\rangle\} \\
\text{(PROJECT)} & [\![\pi_{A_1,\ldots,A_i}(Q)]\!](I) & = & \{\langle\langle A_1 : d_1, \ldots, A_i : d_i\rangle, 2\rangle \mid \\
 & & & \langle A_1 : d_1, \ldots, A_i : d_i, \ldots, A_j : d_j, \ldots, A_n : d_n\rangle \in \mathit{erase}([\![Q]\!](I))\} \\
\text{(CROSS-PROD)} & [\![Q \times Q']\!](I) & = & \{\langle\langle A_1 : d_1, \ldots, A_n : d_n, B_1 : d'_1, \ldots, B_m : d'_m\rangle, 2\rangle \mid \\
 & & & U = \langle A_1 : d_1, \ldots, A_n : d_n\rangle \wedge U' = \langle B_1 : d'_1, \ldots, B_m : d'_m\rangle \wedge \\
 & & & U \in \mathit{erase}([\![Q]\!](I)) \wedge U' \in \mathit{erase}([\![Q']\!](I))\} \\
\text{(JOIN)} & [\![Q \bowtie Q']\!](I) & = & \{\langle\langle A_1 : d_1, \ldots, A_n : d_n, B_1 : b_1, \ldots, B_k : b_k, C_1 : d'_1, \ldots, C_m : d'_m\rangle, 2\rangle \mid \\
 & & & U = \langle A_1 : d_1, \ldots, A_n : d_n, B_1 : b_1, \ldots, B_k : b_k\rangle \wedge \\
 & & & U' = \langle B_1 : b_1, \ldots, B_k : b_k, C_1 : d'_1, \ldots, C_m : d'_m\rangle \wedge \\
 & & & U \in \mathit{erase}([\![Q]\!](I)), U' \in \mathit{erase}([\![Q']\!](I))\} \\
\text{(RENAME)} & [\![\rho_{A_1/B_1,\ldots,A_N/B_n}(Q)]\!](I) & = & \{\langle\langle A_1 : d_1, \ldots, A_n : d_n\rangle, 2\rangle \mid \langle B_1 : d_1, \ldots, B_n : d_n\rangle \in \mathit{erase}([\![Q]\!](I))\} \\
\text{(INSERT)} & [\![\mathrm{ins}_{R,\langle A_1:d_1,\ldots,A_n:d_n\rangle}(Q)]\!](I) & = & \{I', T'\} \mid \{I', T\} = \mathit{erase}([\![Q]\!](I)) \wedge \\
 & & & T = \langle R : S_R, \langle \vec{U}\rangle\rangle \wedge T' = \langle R : S_R, \langle \vec{U}, U'\rangle\rangle \wedge \\
 & & & U' = \langle\langle A_1 : d_1, \ldots, A_n : d_n\rangle, 2\rangle \\
\text{(DELETE)} & [\![\mathrm{del}_{R,A:a}(Q)]\!](I) & = & \{I', T'\} \mid \{I', T\} = \mathit{erase}([\![Q]\!](I)) \wedge \\
 & & & T = \langle R : S_R, \langle \vec{U_{bf}}\rangle\rangle \wedge T' = \langle R : S_R, \langle \vec{U_{af}}\rangle\rangle \wedge \\
 & & & \vec{U_{af}} = \{U \mid U \in \vec{U_{bf}} \wedge U = \langle\langle A_1 : d_1, \ldots, A : d, \ldots, A_n : d_n\rangle, 2\rangle \wedge a \neq d\} \\
\text{(REPLACE)} & [\![\mathrm{repl}_{R,A:a,B:b}(Q)]\!](I) & = & \{I', T'\} \mid \{I', T\} = \mathit{erase}([\![Q]\!](I)) \wedge \\
 & & & T = \langle R : S_R, \langle \vec{U_{bf}}\rangle\rangle \wedge \langle T' = \langle R : S_R, \langle \{\vec{U}, \vec{U'}\}\rangle\rangle \wedge \\
 & & & \vec{U} = \{U \mid U \in \vec{U_{bf}} \wedge U = \langle\langle A_1 : d_1, \ldots, A : a, \ldots, A_n : d_n\rangle, \nu\rangle \wedge a \neq d\} \wedge \\
 & & & \vec{U'} = \{\langle\langle A_1 : d_1, \ldots, B : b, \ldots, A_n : d_n\rangle, 2\rangle \mid \\
 & & & \qquad \langle\langle A_1 : d_1, \ldots, A : a, \ldots, A_n : d_n\rangle, \nu\rangle \in \vec{U_{bf}}\}
\end{array}
$$

**Figure 3: Query semantics.**

Put simply, the theorem states that, from a client's perspective, the result of running a query $Q$ on a database instance $I$ that was created at version 1 and is being lazily updated to version 2 is indistinguishable from the result of running $Q$ on a database that was created at version 2. Note that we use the *erase*() function to strip off version information $\nu$ before passing on the results to the client, as $\nu$ is just an implementation artifact the client should not be aware of.

*Proof.* By induction on the structure of $Q$.

**case** (CONST) **:**

By definition,

$$[\![\langle A : d\rangle]\!](I) = \{\langle A : d\rangle\}$$

hence we have

$$[\![\langle A : d\rangle]\!](z(I)) = [\![\langle A : d\rangle]\!](I^2) = \{\langle A : d\rangle\}$$

and therefore

$$\mathit{erase}([\![\langle A : d\rangle]\!](z(I))) = \mathit{erase}([\![\langle A : d\rangle]\!](I^2)) = \{\langle A : d\rangle\}$$

**case** (REL-INST) **:**

By definition,

$$[\![R]\!](I) = I(R)$$

Let

$$I = \langle R : \langle \vec{A}\rangle, \langle \vec{U}\rangle\rangle, I'$$

that is, $I'$ represents all the relations in the database but $R$. Therefore,

$$I(R) = \langle \vec{U}\rangle$$

We now have two cases.
(I) If $\vec{A} \cap R_\Delta = \emptyset$ then

$$z(\vec{A}, \langle \vec{U}\rangle) = \langle \vec{U}\rangle$$

Since the schema of $R$ has not changed, we have

$$\mathit{erase}(\vec{U}) = \mathit{erase}(\vec{U^2})$$

and

$$\mathit{erase}([\![R]\!](z(I))) = \mathit{erase}([\![R]\!](I^2)) = \mathit{erase}(\langle \vec{U}\rangle)$$

(II) If $\vec{A} \cap R_\Delta \neq \emptyset$ then from the definition of $z$():

$$z(\vec{A}, \langle \vec{U}\rangle) = \langle \vec{U^2}\rangle$$

hence

$$\mathit{erase}([\![R]\!](z(I))) = \mathit{erase}([\![R]\!](I^2)) = \mathit{erase}(\langle \vec{U^2}\rangle)$$

**case** (SELECT-1) **:**

We need to prove

$$erase([\![\sigma_{A_i=d}(Q)]\!](z(I))) = erase([\![\sigma_{A_i=d}(Q)]\!](I^2))$$

By induction on $Q$, we have

$$erase([\![Q]\!](z(I))) = erase([\![Q]\!](I^2))$$

Let $\langle \vec{U'} \rangle = erase([\![Q]\!](z(I)))$, hence $\langle \vec{U'} \rangle = erase([\![Q]\!](I^2))$. We can now simply use the (SELECT-1) definition to show that

$$[\![\sigma_{A_i=d}(Q)]\!](z(I)) = [\![\sigma_{A_i=d}(Q)]\!](I^2) = \{U^2 \mid U \in \langle \vec{U'} \rangle\}$$

hence

$$erase([\![\sigma_{A_i=d}(Q)]\!](z(I))) = erase([\![\sigma_{A_i=d}(Q)]\!](I^2))$$

**case** (SELECT-2) **:**

Using induction as in the (SELECT-1) case, we have

$$erase([\![Q]\!](z(I))) = erase([\![Q]\!](I^2))$$

and by applying (SELECT-2) we get

$$erase([\![\sigma_{A_i=A_j}(Q)]\!](z(I))) = erase([\![\sigma_{A_i=A_j}(Q)]\!](I^2))$$

**case** (PROJECT) **:**

Using induction as in the (SELECT-1) case, we have

$$erase([\![Q]\!](z(I))) = erase([\![Q]\!](I^2))$$

and by applying (PROJECT) we get

$$erase([\![\pi_{A_1,\ldots,A_i}(Q)]\!](z(I))) = erase([\![\pi_{A_1,\ldots,A_i}(Q)]\!](I^2))$$

**case** (CROSS-PROD) **:**

By induction on $Q$ and $Q'$ we have

$$erase([\![Q]\!](z(I))) = erase([\![Q]\!](I^2))$$

and

$$erase([\![Q']\!](z(I))) = erase([\![Q']\!](I^2))$$

By applying (CROSS-PROD) we have

$$erase([\![Q \times Q']\!](z(I))) = erase([\![Q \times Q']\!](I^2))$$

$$= \{\langle U, U' \rangle \mid U \in erase([\![Q]\!](I)), U' \in erase([\![Q']\!](I))\}$$

**case** (JOIN) **:**

Similar to (CROSS-PROD), by using induction on $Q$ and $Q'$ and applying (JOIN).

**case** (RENAME) **:**

Similar to (PROJECT), by using induction on $Q$ and applying (RENAME).

**case** (INSERT) **:**

We need to prove

$$erase([\![\text{ins}_{R,\langle A_1:d_1,\ldots,A_n:d_n\rangle}]\!](z(I))) =$$

$$erase([\![\text{ins}_{R,\langle A_1:d_1,\ldots,A_n:d_n\rangle}]\!](I^2))$$

Using the definition of (INSERT), we have

$$[\![\text{ins}_{R,\langle A_1:d_1,\ldots,A_n:d_n\rangle}]\!](z(I)) = z(\{I', T'\}) = \{z(I'), z(T')\}$$

and

$$[\![\text{ins}_{R,\langle A_1:d_1,\ldots,A_n:d_n\rangle}]\!](I^2) = \{I', T'\}^2 = \{I'^2, T'^2\}$$

where $T'$ is the updated instance of relation $R$ after inserting $\langle A_1 : d_1, \ldots, A_n : d_n \rangle$, and $I'$ consists of the other relations in the database. By using (REL-INST) we have that, for each relation $R' \in T'$,

$$erase(z(R')) = erase(R'^2)$$

hence

$$erase(z(T')) = erase(T'^2)$$

and the remaining burden of proof is

$$erase(z(T')) = erase(T'^2)$$

From the definition of (INSERT) we know that

$$T' = \langle R : S_R, \langle \vec{U}, \langle \langle A_1 : d_1, \ldots, A_n : d_n \rangle, 2 \rangle \rangle \rangle$$

where $T = \langle R : S_R, \langle \vec{U} \rangle \rangle$. Since $\vec{U} = I(R)$, we can use (REL-INST) to get

$$erase(z(\vec{U})) = erase(\vec{U}^2)$$

and the remaining burden of proof is

$$erase(z(\langle \langle A_1 : d_1, \ldots, A_n : d_n \rangle, 2 \rangle)) =$$

$$erase(\langle \langle A_1 : d_1, \ldots, A_n : d_n \rangle, 2 \rangle^2)$$

From the definition of $erase()$, i.e., strip out the associated $\nu$ for a tuple, we have

$$erase(z(\langle \langle A_1 : d_1, \ldots, A_n : d_n \rangle, 2 \rangle)) =$$

$$\langle A_1 : d_1, \ldots, A_n : d_n \rangle$$

and

$$erase(\langle \langle A_1 : d_1, \ldots, A_n : d_n \rangle, 2 \rangle^2) =$$

$$\langle A_1 : d_1, \ldots, A_n : d_n \rangle$$

so we are done.

**case** (DELETE) **:**

Follows by a similar argument to (INSERT), i.e., splitting $I$ into $\{I', T\}$ and proving $erase(z(I')) = erase(I'^2)$ and $erase(z(T')) = erase(T'^2)$.

**case** (REPLACE) **:**

Follows by a similar argument to (INSERT), i.e., splitting $I$ into $\{I', T\}$ and proving $erase(z(I')) = erase(I'^2)$ and $erase(z(T')) = erase(T'^2)$.

□

# 4. USAGE SCENARIO

As mentioned previously, our implementation is designed to be transparent to SQLite clients: they trigger the schema upgrade, though the inner workings of the update process, i.e., lazily converting the databases instance, are invisible.

To show how our implementation is meant to be used in practice, in Figure 4 we present an usage scenario. The

```
1  ...
2  // create and populate
3  // database at schema version 1
4  SQLiteExec("CREATE TABLE R1(schema_R1);
5              CREATE TABLE R2(schema_R2);");
6  SQLiteExec("INSERT INTO R1 ...");
7  // use database at version 1
8  SQLiteExec("SELECT * FROM R2 ...");
9
10 // initiate the update; the argument
11 // passed to UPDATEDB is schema version 2
12 SQLiteExec("UPDATEDB(
13             CREATE TABLE R1(schema_R1');
14             CREATE TABLE R3(schema_R3);)");
15
16 // use the database at version 2
17 SQLiteExec("INSERT INTO R1 ...");
18 SQLiteExec("SELECT * FROM R3 ...");
```

**Figure 4: Sample usage for our system.**

database client, e.g., Mozilla, links with our update-capable SQLite, and can access/query the database via SQLiteExec calls. The client can either create the database, or open an existing database. Suppose our client creates a database instance that contains two tables, R1 and R2 (lines 4–5). After using the database (lines 6–8) the client decides to update the schema. To accomplish this, all the client has to do is call UPDATEDB with the new schema as argument (lines 12–14). Note that the new schema contains two tables, R1 and R3, meaning that relation R2 has been deleted, relation R3 has been added, and possibly, though not necessarily, the schema of relation R1 has changed, e.g., by adding or deleting attributes. Upon receiving the `UPDATEDB` command, our update-capable SQLite will then initiate the lazy database update process and immediately return control to the client. From this point on, the client can safely assume the database has been updated to version 2, and can start using the database (lines 17–18).

In our current implementation, the entire new schema has to be specified as an argument to `UPDATEDB`; if this turns out to be burdensome, we imagine giving the client the option to specify just the differences between the new and old schema using SMOs [25, 8].

# 5. IMPLEMENTATION

Our on-the-fly schema update implementation extends SQLite 3.6.16, and is written in C. The implementation does not require any pre-update preparation on behalf of the application, and does not impose any runtime cost prior to the update, or after the full update is completed. The application simply links with our update-capable SQLite and signals an update by sending an `UPDATEDB` command. After the full update is completed, i.e., all the tuples in the database are at version 2, the safety checks are turned off, and our system imposes no runtime penalty.

After an update has been signaled via `UPDATEDB`, our system first waits for pending queries to finish, then executes three main tasks before returning control to the client:

| SMO | Schema | Tuples |
|---|---|---|
| `CREATE TABLE` | I | - |
| `DROP TABLE` | I | - |
| `RENAME TABLE` | I | - |
| `ADD COLUMN` | I | I |
| `DROP COLUMN` | I | L |
| `RENAME COLUMN` | I | L |
| `COLUMN TYPE CHANGE` | I | L |

**Table 4: Immediate (I) vs lazy (L) updates.**

1. Compute the differences between the old and new schema (Section 5.1).
2. Perform the immediate updates (Section 5.2).
3. Start a low priority background thread which will carry out the lazy updates (Section 5.3).

These three tasks must be completed quickly, to avoid delaying the client; in Section 6.2 we show that, in practice, `UPDATEDB` finishes in at most 361 ms.

## 5.1 Schema Differencing

The difference between the old and new schemas (the $R_\Delta$ from Section 3) must be computed for two reasons: to implement the safety check, and to discern between immediate and deferred updates. The differencing algorithm is straightforward. We first compare the sets of table names in the old and new schemas, to identify tables that were added or deleted. To identify renamings, we perform a pairwise comparison between the schemas of added and deleted tables; if the schema of an added table $R_A$ matches the schema of a deleted table $R_D$ (same attributes, and attributes have the same types), we consider this change as a renaming $R_D \rightarrow R_A$ rather than deleting $R_D$ and adding $R_A$.

We then proceed to identifying table-level schema changes, i.e., for each table, we compute the attributes that were added, deleted, or had a change in type. For detecting attribute renaming, we use the same strategy as for tables: we declare a renaming $A_A \rightarrow A_D$ if, within a table, attribute $A_D$ was deleted, $A_A$ was added, and their types are the same. Note that, when combining renaming with another change (e.g., renaming table $R$ and changing its schema as well, or renaming attribute $A$ and changing its type) our algorithm will signal this change as a deletion followed by an addition. The first column of Table 4 summarizes all the schema changes our differencing algorithm detects.

## 5.2 Immediate Updates

After detecting schema differences, our implementation proceeds to performing some *immediate* changes, i.e., change the database schema to the new schema. Table 4 shows, for each SMO-style change, whether that change is immediate or lazy (deferred). We can see in column 2 that table additions, deletions and renamings (`CREATE TABLE`, `DROP TABLE`, `RENAME TABLE`) are performed immediately, using the built-in SQLite primitives for these operations. Table schema changes (`ADD COLUMN`, `DROP COLUMN`, `RENAME COLUMN`, `COLUMN TYPE CHANGE`) are also done immediately, as follows: SQLite

natively supports column addition without physically extending the table; for column deletion, we hide the column so it won't be visible to queries; for column renaming, we simply change the column name; for column type change, we change the affinity (SQLite term for "preferred" storage type) associated with that column. As mentioned in Section 3.2, this ensures that table and database schemas are always at the newest version.

Once the immediate updates have completed, we initiate the lazy update process (column 3 of Table 4), that gradually converts the database instance (i.e., all the tuples) to the new version. At this point we are ready to process client requests at the new version. Any query that tries to use the old schema will result in an error, since such use is unsafe.

## 5.3 Lazy Updates

To perform lazy updates, we introduce a new SQLite thread, called the "background thread" that essentially carries out tuple-by-tuple, table-by-table updates to the new version. To maintain consistency, the operations of the background thread and the main SQLite thread—which is responsible for processing queries from clients—are mutually exclusive. The main thread has higher priority, and can preempt the background thread.

We now present a high-level view of background thread's operations. The thread iterates over all the tables that require conversion. For each table $R$, its operations depend on the schema modification (SMO) that $R$ has undergone. If the SMO is `ADD COLUMN`, which SQLite supports natively, the thread does not have anything to do, as all the tuples have the default initializer for that column already. If, on the other hand, the SMO is `DROP COLUMN`, `RENAME COLUMN`, or `COLUMN TYPE CHANGE`, then a new *shadow table* $R'$ is silently filled out by the background thread with tuples copied from $R$ and updated. Finally, when all the shadow tables $R'$ are filled out, our implementation atomically switches the old tables with the shadow ones, which effectively marks the end of the lazy update process. At this point we are ready to proceed with another schema update.

*Safety and optimizations.* We now describe how the safety check and lazy updates introduced in Section 3.2 are actually implemented. To ensure that $R$ and $R'$ are consistent, all modification requests (`INSERT`, `UPDATE`, or `REPLACE`) on those $R$s that had a change in schema are replicated into $R'$ as well; the technique is similar to log propagation[15, 17] and DB2's online schema change support [10]. Modification requests on new tables or tables whose schemas were not changed proceed right through, without replication. These techniques effectively implement the $\vec{A} \cap R_\Delta = \emptyset$ check and the $z(\vec{A}, \langle \vec{U} \rangle)$ function. Moreover, in the implementation we do not actually store a version $\nu$ with each tuple. Rather, the background thread keeps its own track (using SQLite's internal `ROWID`) of the tuples that have been, or are yet to be, converted from version 1 to version 2.

*Error tolerance.* Before starting the lazy update process, we write all the necessary update steps in a persistent table that is hidden from the user. If SQLite is shut down abruptly, i.e., due to power loss, the shadow tables $R'$ have not been switched for the original ones, $R$, and we restart the update process from scratch the next time SQLite starts, based on the update information in our persistent table. This is similar to regarding the update as a transaction that was aborted due to shutdown, and is re-tried upon restart. If, on the other hand, a lazy update has not finished but a client wants to exit, we force the client to wait until the update is complete.

# 6. EVALUATION

To evaluate the impact our approach has upon on-the-fly updateable databases, we performed experiments based on real-world schema update and database usage *scenarios*. The experiments focus on four aspects: update time, query performance, disk space overhead, and memory footprint.

## 6.1 Schema Change And Query Scenarios

The schema update scenarios and benchmarks are based on actual updates and queries we observed in practice in Mozilla, Monotone, and Vienna (Section 2.3). Table 5 describes each scenario. Columns 2 and 6 show a summary of old database schemas, i.e., total number of tables and attributes in that schema. Columns 3–5 and 7–10 show how the schema has changed. For example, scenario 2 starts with 3 tables and 7 attributes in total, and adds two attributes; scenario 8 starts with 3 tables and 18 attributes, deletes one table and adds a new table. Finally, the last four columns show the SQL commands we used in the benchmark. For example, we constructed scenario 1 based on Mozilla code; the code contains a mix of `SELECT`s, `INSERT`s and `REPLACE`s, hence our benchmark includes these three SQL statements to accurately model actual use. The number of tuples used in our benchmarks (e.g., initial table sizes, `SELECT` results, or tuples `INSERT`ed) varied between 10,000 and 50,000.

## 6.2 Benchmarking And Results

*SQLite test configurations.* To quantify on-the-fly update overhead, we measured query completion time, memory, and disk usage in two configurations: *updateable*—using our update-enabled SQLite to create and populate the database at schema $\mathcal{R}_1$, performing an update that changes the schema to $\mathcal{R}_2$, and then running the query benchmark suite, and *stock*—using an off-the-shelf version of SQLite, creating and populating the database at schema $\mathcal{R}_2$, and then running the query benchmark suite.

*Hardware and software configuration.* We conducted our experiments on a two-CPU, quad-core Xeon@2.33GHz system with 12GB of RAM and a RAID5 array of three Western Digital RE3 1TB@7200 rpm hard drives. Since SQLite is a server-less library linked into the client program (the client and SQLite run in the same process) there was no need for separate client and server systems. The test system ran CentOS 5.3, kernel version 2.6.18. Both stock SQLite and updateable SQLite configurations were compiled using with `gcc` 4.1.2, compiler flags `-g -O2` (these are the out-of-the-box compilation options for the official SQLite distribution). Time measurements are performed using the CPU's time stamp counter (TSC).

In all cases, we report the median of 5 runs.

| Scenario ID | Tables | | | | Attributes | | | | | Benchmark | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | old version count | changes | | | old version count | changes | | | | SELECT | INSERT | REPLACE | DELETE |
| | | add | delete | rename | | add | delete | rename | type change | | | | |
| 1 | 1 | - | - | - | 4 | 3 | - | - | - | ✓ | ✓ | ✓ | |
| 2 | 3 | - | - | - | 7 | 2 | - | - | - | ✓ | ✓ | | |
| 3 | 2 | - | - | - | 5 | 3 | - | - | - | | ✓ | ✓ | |
| 4 | 1 | - | - | - | 7 | 2 | 2 | - | - | | | | ✓ |
| 5 | 1 | - | - | - | 13 | 3 | - | - | - | | | | ✓ |
| 6 | 11 | - | - | - | 55 | - | - | 1 | - | ✓ | | | |
| 7 | 3 | - | - | - | 18 | - | - | - | 1 | | ✓ | | ✓ |
| 8 | 3 | 1 | 1 | - | 18 | - | - | - | - | | ✓ | | |
| 9 | 11 | - | - | 1 | 55 | - | - | - | - | | ✓ | | ✓ |

**Table 5: Schema update scenarios (old version schemas and changes to tables/attributes) and benchmarks.**

*Update time.* A key requirement of our system was to give the client the appearance that the update has completed immediately. We measured the time to execute the `UPDATEDB` command, i.e., the time difference from the moment the client signals a schema change to the moment when control returns to the client. Column 2 of Table 6 shows these times for each schema change scenario. As we can see, we could always complete an `UPDATEDB` in at most 361 milliseconds, which is crucial for ensuring our approach disrupts the client as little as possible.

*Query performance.* We want to keep the client-perceived query processing overhead low for the period the database is undergoing lazy conversion. Therefore, we measured the query benchmark completion times in stock and updateable configurations. Column 3 of Table 6 shows the time increase, in percents, for the updateable version as compared to the stock version. For example, in scenario 1, the updateable version took 2.04% longer to complete the benchmark; in scenario 3 the value is slightly negative because we use medians rather than averages. We emphasize that query processing overhead is transient and goes to zero after the lazy update is completed.

*Disk space footprint.* To get an idea of the additional steady-state disk space overhead our approach imposes, we measured the size of the on-disk database file at the completion of lazy update thread, again in the two configurations. We found (column 4 of Table 6) that the updateable case uses between 0.36% and 28.57% more disk space than stock.

*Memory footprint.* The additional memory space used for lazy updates (e.g., shadow tables) will temporarily result in higher memory footprints for applications that link with our update-enabled SQLite system. Therefore, we compared *peak* memory footprint in the stock and updateable cases. This information will give us an idea of maximum memory requirements, which is especially important for mobile and embedded devices. We obtained peak memory usage from the `VmPeak` field of `/proc/<pid>/status` as reported by the Linux kernel. In column 5 of Table 6 we see that the extra memory required ranged from 16KB to 12.7MB.

| Scenario ID | Overhead | | | |
|---|---|---|---|---|
| | (ms) | % | | (kB) |
| | `UPDATEDB` | Time | Disk | Memory (peak) |
| 1 | 112 | 2.04 | 1.87 | +10,264 |
| 2 | 119 | 3.12 | 3.45 | +10,264 |
| 3 | 70 | -0.08 | 0.36 | +10,269 |
| 4 | 53 | 6.03 | 6.85 | +12,643 |
| 5 | 86 | 8.70 | 0.61 | +10,260 |
| 6 | 361 | 3.01 | 28.57 | +10,588 |
| 7 | 151 | 2.00 | 5.21 | +12,768 |
| 8 | 36 | 16.00 | 10.05 | +16 |
| 9 | 287 | 5.75 | 18.49 | +16 |

**Table 6: Update time and overhead.**

*Online schema changes.* $\mathcal{PRISM}$ [8] is a system aimed at replacing the current manual schema evolution process with a fully-automated one. The database administrator has to specify schema differences between the two versions, as SMOs. $\mathcal{PRISM}$ then provides an assessment whether the schema change is information-preserving, and automatically rewrites the old queries into queries that can run on the new schema version; it also generates an inverse (new to old) transformation automatically, as well as data migration scripts that can migrate data between the old and new versions. At this point, the system is ready to migrate and answer new queries (directly), as well as old queries (via rewriting). We take a more pragmatic approach: we do not require that SMOs be specified, but strive to provide automatic conversion to the new schema because of the more simple ways SQLite-based systems tend to change, as we observed in practice for our applications. We also do not perform any query rewriting, because it is not necessary: when the application switches to the new version, it already has the new version queries built in (Section 2.4). Finally, our system has no notion of history beyond "old" and "new" versions; nevertheless, this does not preclude it from supporting long-term evolution by simply applying updates in

## 7. RELATED WORK

sequence.

Ronström [24] presents a system for online schema changes in the context of telecommunication systems. The system supports a more complex set of schema changes (e.g., splitting and merging tables horizontally or vertically, adding indexes) than ours. To ensure safety, their approach relies on user-supplied "test" transactions that are meant to be executed after the schema change has completed; if the test transactions fail, the schema change is aborted and the system rolls back to the old schema. Our system does not support such complex schema changes; based on our empirical study, these complex schema changes are rarely used. Our approach to guaranteeing safety is different, too: we use safety checks to prevent unsafe queries, rather than allowing all changes and having to roll back. An evaluation of their system is not provided.

Løland and Hvasshovd [15] describe a system for allowing online full outer joins and vertical table splits. After an update has been signaled, their system creates the required new tables and starts populating them; for all the post-update operations a log is kept, and just prior to phasing off the old tables, log propagation will ensure the newly-added tables contain all the required data. They provide proofs sketches for ensuring the consistency of the old and new tables. They also conduct an evaluation of their system and show that the transient overhead imposed by their system is typically 2% (and at most most 11%) for throughput and typically 5% (and at most 30%) for response time. While our safety guarantee is somewhat similar to theirs, we provide a full safety proof; the set of schema changes their and our systems support differs. Their implementation consists of a Java-based prototype, whereas we implement and evaluate our approach on a popular SQL engine.

*Schema evolution studies.* Curino et al. [7] have studied schema evolution for Wikipedia from April 2003 to November 2007. Their study provides both micro- and macro-classification of changes. The micro-classifications correspond to SMO syntax, which is a superset of the changes we investigate; in particular, we do not collect data on `DISTRIBUTE TABLE`, `MERGE TABLE`, `COPY COLUMN`, and `MOVE COLUMN`. Macro-classifications include changes to indexes, keys, types, syntax and engine; while we could collect such macro-level changes, we decided to only consider changes to types, as these have a direct impact on our implementation. Their study, just like ours, finds that the most frequent changes are column addition and deletion; however, column renaming seems to be much more frequent in Wikipedia than in our applications. Interestingly, they find query failure rate in Wikipedia to be about 10% short-term (i.e., when skipping one update or migration) and 84% long term (if skipping 40 or more updates or migrations). The short-term number is lower than what we find for Mozilla (Table 3).

Sjøberg [26] presents a schema evolution study on a health management system over 1.5 years; their findings are similar to ours, i.e., most frequent changes are field additions/deletions and table additions/deletions.

*Lazy updates.* Lazy updates were used by other researchers in prior work, in the context of object-oriented databases [9, 4] and persistent object stores [6]. Automatic object upgrades on top of Thor [6] enjoy strong safety guarantees (i.e., preserving object invariants) in the presence of lazy upgrades; this is accomplished by using object encapsulation and enforcing so-called modularity conditions (i.e., making sure that any transaction that uses yet-to-be upgraded objects will first transform object to the new version). Our system also guarantees that any post-update accesses to yet-to-be-converted tuples will see the new version, though enforcing this condition in our system is simplified by the lack of object dependencies.

*Large-scale and commercial DBMSs.* DBMSs used in large applications provide limited support for schema changes, and most frequent changes cannot be performed online. Moreover, these DBMSs cannot be linked into existing applications since they require a dedicated server. MySQL [1] and Microsoft SQL Server [16] permit table renaming and column addition/deletion/renaming/type change. IBM DB2 permits table renaming, column addition/renaming and a limited set of column type changes [13]. For MySQL, according to the manual for version 5.1, all changes except attribute renaming will create and fill out an extra table that will replace the old one, and "updates and writes to the [old] table are stalled until the new table is ready". Microsoft SQL Server "blocks all outside operations until the [schema modification] lock is released". IBM DB2 will return new-schema results for `SELECT`s, though the underlying data is not changed; it is changed whenever rows are inserted/updated [10].

Edition-based redefinition in Oracle 11g Release 2 [21] allows online upgrades via "hot rollover". Changes are installed in a new edition; new-version clients will see the data at the new schema via the new-edition view, while old clients will access the data at the old schema via the old-edition view. Supporting both versions simultaneously is required because of long-lived transactions [11] and clients who wish to first test the new version before switching to it. Cross-edition triggers propagate changes between the two views; these triggers are programmer-defined, must be idempotent and the question of invertibility (hence consistency between the two editions) is left to the programmer. We take a different approach (one active version at a time) because the update is client-initiated, hence there is no need to support two versions simultaneously; this way we avoid consistency issues and having to write triggers.

*Schema versioning, schema matching and schema mapping.* The primary difference between these approaches and ours is the application domain: we are interested in forward-only updates to small and medium applications, whereas they target large, multi-version, possible heterogeneous, databases. Schema versioning and temporal querying approaches such as $\mathcal{PRIMA}$ [18] or timetravel in Ganymed [22] store data at multiple versions and allow queries that span multiple schema versions. Schema mapping systems such as MACES [30] or ToMAS [29] allow heterogeneous systems that assume different schemas to interact properly via mappings. Schema matching [23] and ontology mapping [14] try to address the problem of accessing data where the client and server formats are different by providing matching or mapping functions between the two formats. *Lenses* [5], an ap-

proach for bi-directional transformation between strings belonging to different languages, are similar to schema matching or schema mapping, and provide an elegant solution to the invertibility problem. While essential in heterogeneous systems and providing strong safety guarantees, versioning, matching, and mapping approaches have limited applicability for SQLite-based applications. First, their power comes at a cost—using dedicated systems or separate applications—hence unlikely to be easily integrated in embedded, mobile, or desktop applications that form our target. Second, our applications are unlikely to go back to old versions (i.e., they always "march forward"), therefore a transient performance overhead is preferable to the continuous overhead associated with storing old versions and mapping/matching.

# 8. CONCLUSIONS

In this paper we present an approach for safe on-the-fly schema evolution. This work was motivated by the ubiquity of SQLite, and the lack of an online schema upgrade mechanism for it. We first conduct an empirical study on the evolution of three open source applications that use SQLite. We find that schemas usually change in simple ways, and the queries used in these applications use a small set of SQL commands. We then implement support for on-the-fly changes to schemas by extending SQLite in a client-transparent manner. We use a calculus that can model frequently-used schema changes and SQL commands, and prove that schema updates preserve query safety. We evaluate our approach on real-world schema changes and queries extracted from our examined open source applications and find that support for online schema updates does not significantly impact application performance. We believe that our safe on-the-fly schema update system fulfills the key requirements for online database updates: correctness, performance, and error tolerance. Our system has the potential to improve software availability and user experience for a variety of embedded, mobile and desktop applications.

# 9. REFERENCES


[1] *MySQL 5.1 Reference Manual*. http://dev.mysql.com/doc/refman/5.1/en/alter-table.html.

[2] S. Abiteboul, R. Hull, and V. Vianu, editors. *Foundations of Databases: The Logical Level*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[3] Anonymous. Title withheld to protect anonimity. Available upon request.

[4] J. Banerjee, W. Kim, H.-J. Kim, and H. F. Korth. Semantics and implementation of schema evolution in object-oriented databases. *SIGMOD Rec.*, 1987.

[5] A. Bohannon, B. C. Pierce, and J. A. Vaughan. Relational lenses: a language for updatable views. In *PODS*, 2006.

[6] C. Boyapati, B. Liskov, L. Shrira, C.-H. Moh, and S. Richman. Lazy modular upgrades in persistent object stores. In *OOPSLA*, 2003.

[7] C. Curino, H. J. Moon, L. Tanca, and C. Zaniolo. Schema evolution in wikipedia - toward a web information system benchmark. In *ICEIS (1)*, 2008.

[8] C. A. Curino, H. J. Moon, and C. Zaniolo. Graceful database schema evolution: the prism workbench. *VLDB*, 2008.

[9] F. Ferrandina, T. Meyer, R. Zicari, G. Ferran, and J. Madec. Schema and database evolution in the o2 object database system. In *VLDB*, 1995.

[10] C. Friske. DB2 online schema changes - what's new in DB2 version 8. *SHARE*, Session 1323, February 2004.

[11] H. Garcia-Molina and K. Salem. Sagas. *SIGMOD Rec.*, 16(3):249–259, 1987.

[12] D. R. Hipp. Sqlite. http://www.sqlite.org/.

[13] IBM. *DB2 Version 9.1 for z/OS information*. http://publib.boulder.ibm.com/infocenter/dzichelp/v2r2/index.jsp?topic=/com.ibm.db29.doc/db2prodhome.htm.

[14] Y. Kalfoglou and M. Schorlemmer. Ontology mapping: the state of the art. *KER*, 18(01):1–31, 2003.

[15] J. Løland and S.-O. Hvasshovd. Online, non-blocking relational schema changes. In *EDBT 2006. LNCS 3896*, pages 405–422, 2006.

[16] Microsoft Corporation. *Microsoft SQL Server 2008 - Locking in the Database Engine*. http://msdn.microsoft.com/en-us/library/ms175519.aspx.

[17] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, 1992.

[18] H. J. Moon, C. A. Curino, A. Deutsch, C.-Y. Hou, and C. Zaniolo. Managing and querying transaction-time databases under schema evolution. *VLDB*, 2008.

[19] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical dynamic software updating for C. In *PLDI*, 2006.

[20] A. Nicoara, G. Alonso, and T. Roscoe. Controlled, systematic, and efficient code replacement for running java programs. In *Eurosys*, 2008.

[21] Oracle. *Edition-Based Redefinition*. http://www.oracle.com/technology/deploy/availability/pdf/edition_based_redefinition.pdf.

[22] C. Plattner, A. Wapf, and G. Alonso. Searching in time. In *SIGMOD*, pages 754–756, 2006.

[23] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10(4):334–350, 2001.

[24] M. Ronström. On-line schema update for a telecom database. In *ICDE*, 2000.

[25] B. Shneiderman and G. Thomas. An architecture for automatic relational database system conversion. *ACM Trans. Database Syst.*, 7(2):235–257, 1982.

[26] D. Sjøberg. Quantifying schema evolution. In *Information and Software Technology*, volume 35, pages 35–44, January 1993.

[27] G. H. Sockut and B. R. Iyer. Online reorganization of databases. *ACM Comput. Surv.*, 41(3):1–136, 2009.

[28] S. Subramanian, M. Hicks, and K. S. McKinley. Dynamic software updates: a vm-centric approach. In *PLDI*, 2009.

[29] Y. Velegrakis, R. J. Miller, and L. Popa. Mapping adaptation under evolving schemas. In *VLDB*, 2003.

[30] C. Yu and L. Popa. Semantic adaptation of schema mappings when schemas evolve. In *VLDB*, 2005.