Improving Cloud Availability with On-the-fly Schema Updates

Iulian Neamtiu
Department of Computer Science and Engineering
University of California, Riverside, CA, USA
neamtiu@cs.ucr.edu

Jonathan Bardin
Laboratoire d'Informatique de Grenoble
Université de Grenoble, France
jonathan.bardin@imag.fr

Md. Reaz Uddin
Department of Computer Science and Engineering
University of California, Riverside, CA, USA
uddinm@cs.ucr.edu

Dien-Yen Lin
Department of Computer Science and Engineering
University of California, Riverside, CA, USA
dienyen@cs.ucr.edu

Pamela Bhattacharya
Department of Computer Science and Engineering
University of California, Riverside, CA, USA
pamelab@cs.ucr.edu

ABSTRACT

Database applications undergo frequent schema changes. To change the schema, the application has to be shut down and migrated to the new version, or run in a mixed mode that supports old and new clients, e.g., via schema versioning. Shutdowns are problematic for applications that cannot tolerate downtime such as embedded, real-time or mission-critical systems; in the Cloud, shutdowns can lead to Service Level Agreement violations or worse, to service interruptions for critical platforms such as healthcare. Mixed-mode operation raises programmability, consistency and performance degradation issues. In this paper we present a system that exploits the march-forward nature of many database applications (no need to support old schema versions after the schema update) to provide on-the-fly updates while ensuring consistency, and being transparent to clients. We first study schema evolution and database usage based on longitudinal studies of four popular open source applications. Next, we implement support for safe on-the-fly schema updates on top of the popular SQLite database engine. Finally, we evaluate our approach using real-world schema changes and database usage scenarios for applications running in the Cloud on the Amazon Web Services platform, and on server systems. We show how on-the-fly schema updates can increase Cloud applications' availability from less than two nines (99%) to more than six nines (99.9999%). More generally, our experiments indicate that SQLite-based applications can enjoy fully automatic on-the-fly schema updates at a low, transient overhead.

The 19th International Conference on Management of Data (COMAD), 19th-21st Dec, 2013 at Ahmedabad, India.

1. INTRODUCTION

Software evolves at a high cadence, with a recent study showing that some cloud applications are updated more than *once a week* [11]. Application developers are under pressure to update their database schemas and code in order to fix bugs and add new features. Unfortunately, the requirement that updates be released often and applied quickly is at odds with providing a seamless, uninterrupted service. In particular, Software as a Service (SaaS) and Platform as a Service (PaaS) providers are bound by Service Level Agreements (SLA) to provide high-availability services, and stopping service to update a database schema might lead to long periods of unavailability which violate the SLA. For example, the Google Apps for Business SLA stipulates that applications must be available 99.9% of the time, otherwise Google is liable to compensate users for unavailability [15]. As another example, when upgrading MediaWiki (the wiki platform used in Wikipedia), out of the 55 possible upgrades among versions 1.1–1.10, only 5 can be performed online [10]. One such upgrade, from version 1.4 to 1.5, required 22 hours of Wikipedia downtime [39].

Furthermore, for high-availability applications, such as healthcare, where 24/7 service is essential, stopping an application to update the database schema is unacceptable. For server and desktop applications, stopping/restarting a program or the OS is disruptive to users. Therefore, there is a tension between the need to frequently update software and the need to abide by high-availability provisions (e.g., as stipulated in the SLA). To solve this tension, we could use solutions such as such as query rewriting [9], schema versioning and temporal querying [25, 28], schema mapping [41, 37], or schema matching [29]. However, the flexibility offered by these solutions must be weighed against their potential cost and operating constraints. For example, a recent schema evolution system can exhibit response lags of up to 30 seconds per schema change while computing mappings between schema versions [25]—depending on the SLA, this unavailability might lead to SLA violations. In another system, rewritten queries run with a permanent overhead and are on average 4.5 times slower than original queries [9]—however,

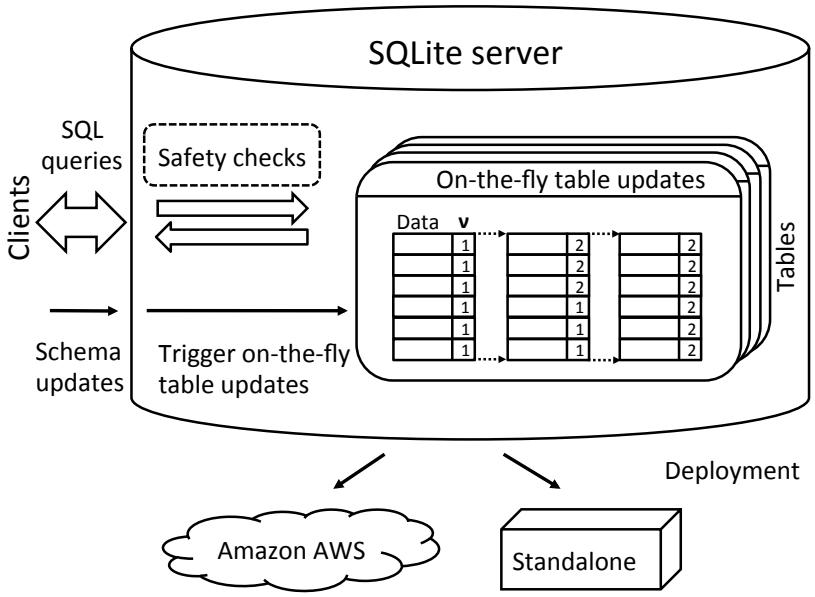

Figure 1: Overview of our system: use, implementation, and deployment.

high overhead is difficult to accept in a pay-for-service environment, e.g., the cloud. Therefore, such solutions might have limited applicability for those non-stop applications that are resource-constrained, cost-sensitive, performance-critical, or do not tolerate high response times.

We propose an alternative solution to this tension between frequent updates and high availability. Specifically, we target march-forward applications—cloud and server applications whose database schemas change, but multi-version support is not a requirement. We call this scenario "march forward", i.e., applications do not need to access data using the old version schema. To support applying updates while sustaining continuous service, we introduce an on-the-fly schema update technique that exploits the march-forward nature of many applications. Our system allows marching-forward, i.e., switching to a new schema at runtime, without having to shut down the database and compromise availability. Through experiments, we show that our technique supports fully automatic, on-the-fly schema updates with an update lag of at most 593 milliseconds; assuming one update per week, the lag is lower than 605 msec, the six nines (99.9999%) threshold [2]. Moreover, our approach has no permanent performance, memory or disk overhead.

Note how these two features are essential, especially for cloud software: the short update lag will ensure that application providers abide by the SLA, while the lack of a permanent overhead will ensure that no additional costs are incurred when using the pay-for-service cloud computing model. We emphasize that our approach is complementary to the schema versioning/mapping/matching systems presented above, as it targets a different set of applications with a different set of requirements.

Roddick et al. have identified "experience [with change] in real systems" as an important research challenge in evolving DBMSs [30]. We pick up this challenge and present an approach for enabling on-the-fly schema updates in SQLite, a very popular [1] SQL engine [16] used in the construction of cloud software (e.g., the OpenNebula open source toolkit for cloud computing, Ruby on Rails), operating systems (e.g., Mac OS X, Solaris 10, OpenSolaris), and desktop applications (e.g., Apple Mail, Safari, iTunes, Mozilla, McAfee antivirus, Dropbox). Our design and evaluation are based on studying long-term schema evolution in popular open source programs, some of which are migrating into the cloud (Monotone, 6 years; BiblioteQ, 2 years) or running on desktops (Mozilla, 3.5 years; Vienna, 4 years).

Figure 1 presents a high-level view of the use, implementation, deployment, and experimental setup for our system. Clients (left) interact with the database server (right) via SQL queries; the server can be deployed either in the Amazon AWS cloud or as a standalone server (bottom).

A schema update is triggered when the server receives a schema update command specifying the new schema. The technique works transparently to clients: once the server has received a schema update command, our implementation will trigger an on-the-fly update across the entire database; the update proceeds lazily in the background, but control returns immediately to the client, so clients can actually perform database operations (queries or updates), right away, at the new schema, hence sustaining service and abiding by SLAs. Clients are not aware of the lazy update proceeding in the background except for the fact that an operation might take slightly longer than what it would take if the database was started directly at the new schema. This overhead, however, is transient: after the lazy update process has finished, the database operations will run at full speed and the system is ready for another update. While an update is in progress, queries are subject to safety checks (dotted box) and in each table affected by the update, tuples are converted from the old version ($\boldsymbol{\nu} = 1$ in Figure 1) to the new version ($\boldsymbol{\nu} = 2$); we will explain the safety checks and lazy updates in great detail later on.

In constructing our on-the-fly update system, we started

[1] As of October 2013, an estimate by the SQLite development team puts the number of SQLite installations upwards of 500 million [35].

with a study on schema evolution and post-update query failure rate. The study looks at the evolution of four popular march-forward open source applications, BiblioteQ, Mozilla, Monotone and Vienna, and identifies how databases evolve and what kind of queries are most popular in practice. We found that, over the evolution periods we analyzed, there were 19 schema updates in BiblioteQ, 42 schema updates in Mozilla, 11 in Monotone, and 7 in Vienna; these updates consist of 150 table or attribute changes in BiblioteQ, 120 in Mozilla, 39 in Monotone and 10 in Vienna. The complete results of our study are presented in Section 2.2.

Allowing a client to run queries that assume a new schema while the database is at the old schema (or naively allowing queries to execute without checks when the database is partially converted) is unsafe. 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. In Section 3, we show how we prevent such errors via safety checks that guarantee that database operations are always safe, even though the database is in a partially-updated state.

In Section 4 we present a sample scenario of how clients 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 used real-world schema updates and queries from our examined programs; based on changes and usage patterns we observed in these programs, we construct a variety of benchmarks meant to quantify the impact of lazy on-the-fly updates on applications linking with SQLite. We conducted experiments on both Amazon AWS cloud and on standalone servers. The databases we used in our experiments ranged from 1 million to 11 million tuples, and we measured overhead by performing 1,000 queries after each update. We found that the performance, memory and disk overheads are low and temporary, and after signaling a schema update, control returns in 593 milliseconds or less. With traditional migration techniques, a schema update can take up to 6,206 seconds (103 minutes). This difference is substantial: assuming one update per week, on-the-fly schema updates can increase cloud availability from less than two nines (99%) to more than six nines (99.9999%). Finally, we verified that our implementation is correct by comparing lazily-updated database contents and query results with new-version contents and results and found them to be identical.

Though we built on SQLite, we believe that our techniques and safety condition 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 [34]; the survey identifies three key requirements for such systems. These requirements were 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."

In short, this paper makes the following contributions:

- A study of query usage and post-evolution query failures in four popular SQLite-based applications.
- An implementation that extends SQLite with support for march-forward, on-the-fly updates, and runs on both desktop/server systems and in the cloud (Amazon AWS); the implementation fulfills the three main requirements of correctness, performance, and error tolerance.
- An evaluation of our approach using real-world schema evolution and query benchmarks on both cloud and server platforms.

2. MOTIVATION

In this section we describe the motivating factors that lead us to pursue on-the-fly database updates 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

SQLite is used in the construction of many critical applications, from cloud software to server OSes, to version control servers, to file storage programs and antivirus suites. These applications do not tolerate restarts and unavailability: long periods of cloud unavailability lead to SLA violations [6, 11]; an unresponsive server OS affects multiple users; shutting down the version control software inconveniences developers; shutting down the antivirus leaves a system vulnerable.

Therefore, we need to find a way to keep these applications running continuously while being able to update them with new features and bug fixes. Dynamic software updating systems such as Ksplice [4] and Jvolve [36] allow on-the-fly updates to code and in-memory data, but not to persistent data. Therefore, such dynamic updating systems are insufficient for performing on-the-fly updates to applications that require changes to database schemas. For example, in the update from Monotone 0.44 to Monotone 0.45, the attributes `id` and `keypair` in table `revision_certs` were renamed to `revision_id` and `keypair_id`; in table `public_key`, the attribute `hash` was deleted and the attribute `name` was added. If we use a dynamic updating system for Monotone, we can update the code, but the information stored in the database remains at the old version, which will lead to incompatibility when the new code tries to access the database. Our system gives the application the option to perform an on-the-fly database update whenever necessary.

2.2 Schema Evolution Study

The development of our system was driven by an empirical analysis of schema evolution, query usage, and post-update query failures. Previous efforts have investigated schema evolution: Mandalapa [21] has studied schema evolution in TikiWiki, Joomla, Slash, MediaWiki (23 years, cumulative); Curino et al. [8] have analyzed schema evolution for Wikipedia (4.5 years); Sjøberg [33] has looked into the schema evolution of a health management system (1.5 years). However, the domain of these studies was not SQLite applications, so the evolution trends and patterns they identified might not translate to our domain. Therefore, to understand how SQLite-based relational databases evolve and are used in practice, in this and prior work we performed our own analysis, covering a cumulative 15.5 years of evolution, on four popular open source programs: BiblioteQ, Mozilla, Monotone, and Vienna.[2]

[2]The studies of query usage and query failure rates are new

Table 1: Schema evolution in our test applications.

SMO	Mozilla		Monotone		Vienna		BiblioteQ	
	count	%	count	%	count	%	count	%
ADD COLUMN	58	48	11	28	10	100	27	18
DROP COLUMN	30	25	9	23	-	-	28	18.6
CREATE TABLE	20	17	9	23	-	-	4	2.7
DROP TABLE	4	3.3	8	20	-	-	8	5.3
RENAME TABLE	5	4.2	1	2.6	-	-	-	-
RENAME COLUMN	2	1.7	1	2.6	-	-	-	-
COLUMN TYPE CHG	1	0.8	-	-	-	-	83	55.3

Table 2: Query failure rate in Mozilla in the absence of migration or updates.

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

BiblioteQ [1] is a catalog and library management suite. Mozilla [26] 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 [24] is a version control system; it uses SQLite to store file revisions, deltas, and branch information. Vienna [38] is a popular RSS/Atom newsreader for Mac OS X; it uses SQLite to store news folders and messages. Our analysis covers the recent evolution (each official release) of the four examined systems: 3.5 years for Mozilla, 6 years for Monotone, 4 years for Vienna, and 23 months for BiblioteQ.

In Table 1 we summarize the schema evolution study results, as Schema Modification Operators (SMOs) [32, 9]. For each SMO we provide both the total count and percentages. For example, there were 58 `ADD COLUMN` changes in Mozilla, which constitute 48% of the total number of changes for that application. As we can see, the most frequent operations alter table schemas to add or delete columns, followed by table addition/deletion/renaming, column renaming and column type change.

3. SAFETY

3.1 The Need for Safety

In prior work we have shown that schema update in Mozilla is sometimes ad-hoc, i.e., the application does not check the schema version in the database prior to executing queries [19]. Without migration after updating the schema, 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 2 we present the query failure rates (in the absence of migration or updates) for the Mozilla files exhibiting frequent 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. We now present some examples of how post-update queries could fail if they execute on pre-update schemas.

Example 1. If an update adds a new table R, then running a post-update query **SELECT** $*$ **FROM** R on a pre-update schema will result in a "Table R does not exist" error.

Example 2. If an update changes the schema of a table R from $\langle A_1, A_2\rangle$ to $\langle A_1, A_2, A_3\rangle$, and the pre-update table instance is:

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

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

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

A post-update query **SELECT** $A_3 \ldots$ **FROM** R will lead to a safety violation, as the attribute A_3 does not exist for each tuple in R.

Example 3. The converse situation from example 2 is equally problematic: if an update changes the schema of a table R from $\langle A_1, A_2, A_3\rangle$ to $\langle A_1, A_2\rangle$, then a post-update **SELECT** $*$ **FROM** R could also return values for the deleted attribute A_3 which will confuse or even crash the client.

3.2 Versions

The key idea behind lazy updates is to allow old and new tuples to coexist, and when a post-update 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. 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 exposition. In practice, though, a database can undergo multiple, sequential updates, hence in our implementation the "new" and

for this work; similar schema evolution analysis results were reported in our prior work, albeit with some differences in projects examined and time spans [40, 19].

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

Figure 2: Sample use case for our system.

"old" schemas and tuples always refer to the latest version and the version prior to it.

A convenient aspect of our SQLite-based model is that we need not perform any query rewriting, because the schema update is client-initiated. Hence, clients only send an **UPDATEDB** (new_schema) command to the server when the client code contains the new query versions; the **UPDATEDB** will switch the server schema and contents to the new version, hence the queries and the database are in-sync.

3.3 Safety Checks

Let $R_S = \{R.A_1, R.A_2, \ldots, R.A_n\}$ be the schema of table R consisting of the set of all attributes $A_1, A_2, \ldots, A_n$ in R, prefixed with the table name. Then the schema $\mathcal{R}$ of the entire database is the union of R_S's over all tables R. Let the old database schema be $\mathcal{R}^1$, and the new database schema be $\mathcal{R}^2$. We denote with $\mathcal{R}_\Delta$ the symmetric difference between $\mathcal{R}^1$ and $\mathcal{R}^2$. In effect, $\mathcal{R}_\Delta$ contains all the tables and attributes that were deleted or added. Whenever a table R with schema R_S is accessed after the schema update (but before the lazy schema update has finished), we perform a *safety check* $R_S \cap \mathcal{R}_\Delta$ prior to each query. The safety check is the crucial mechanism for safety; it ensures that all accesses to a table R with schema R_S are performed at version 2. We have two cases:

1. If the schema update does not alter the schema R_S, i.e.,
   $R_S \cap \mathcal{R}_\Delta = \emptyset$, then we can safely access any tuple in the table as the contents of data is the same in both version 1 and version 2.

2. If the schema update alters R_S, i.e., $R_S \cap \mathcal{R}_\Delta \neq \emptyset$, then in theory we would have to "stall" the query until the lazy schema update has completed for that table, i.e., the conversion of all the tuples from version 1 to version 2 has finished.

As we will describe in Section 5, however, in practice our implementation does not stall any query, but rather uses the $R_S \cap \mathcal{R}_\Delta$ check to optimize log propagation; this speeds up query processing significantly, while still preserving safety.

4. USE CASE

Our implementation is designed to be transparent to clients: they trigger the schema update, but the inner workings of the update process, i.e., lazily converting the database, are invisible. To show how our implementation is meant to be used in practice, in Figure 2 we present an actual use case. The clients can either create the database, or open an existing database. Suppose a database instance is created, that contains two tables, R1 and R2 (lines 3–4). After using the database (lines 8–9) we decide to update the schema. To accomplish this, an **UPDATEDB** command is sent to the server with the new schema as argument (lines 13–14). 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 clients. From this point on, the clients 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**; this is simply an implementation choice, as we could have just as easily chosen to specify the differences between the new and old schema using SMOs instead [32, 9].

5. IMPLEMENTATION

Our on-the-fly schema update implementation extends SQLite version 3.6.16. 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 update is completed. The application simply links with our update-capable SQLite; clients signal an update by sending an **UPDATEDB** command. After an update has been signaled, our system first waits for pending queries to finish, then executes three main tasks before returning control to the client: (1) computes the differences between the old and new schemas, (2) performs some immediate updates, and (3) starts a background thread which will carry out the lazy (deferred) updates. These three tasks must be completed quickly, to avoid delaying the client; in Section 6 we show that, on our workloads, **UPDATEDB** finished in at most 593 milliseconds. After the lazy update is complete, i.e., all the tuples in the database are at version 2, the safety checks are turned off, and our system imposes no query overhead.

Schema differencing. The difference between the old and new schemas (the $\mathcal{R}_\Delta$ from Section 3.3) 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 compare the tables, attributes and attribute types in the old and new schemas and identify all additions, deletions, renamings, and type changes.

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 additions, deletions and renamings (`CREATE TABLE`, `DROP TABLE`, `RENAME TABLE`) are performed immediately, using the built-in SQLite primitives for these operations. Some table schema changes (`ADD COLUMN`, `RENAME COLUMN`) are also done immediately since SQLite natively supports column addition and renaming via the `ALTER TABLE`

statement. For `DROP COLUMN`, we hide the column so it won't be visible to queries, and initiate a lazy update, as described next. Finally, for `COLUMN TYPE CHANGED`, we change the affinity (SQLite term for "preferred" storage type) associated with that column, and initiate a lazy update. As mentioned in Section 3.2, immediate updates ensure that table and database schemas are always at the newest version.

Once the immediate updates have completed, we initiate the lazy update process 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.

Lazy updates. To perform lazy updates, we introduce a new SQLite thread, called the "background thread" that essentially carries out the tuple-by-tuple, table-by-table conversion to the new version.

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, the thread's operations depend on the schema modification operator (SMO) that R has undergone. If the SMO is `ADD COLUMN` or `RENAME COLUMN`, which SQLite supports natively, the thread does not have anything to do. If the SMO is `DROP 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. For the other changes that require lazy updates, we have changed the SQLite low-level methods in charge of handling queries to allow query processing to proceed in parallel with the lazy update yet maintain consistency, as follows. In each SQLite method corresponding to an SQL clause (e.g., `SELECT`, `DELETE`, `REPLACE`, `INSERT`, `WHERE`) we check whether an update is pending, and whether the query is done on an updated table. If both conditions are true, and the current thread is not the background thread in charge of the update, we change the targeted table and attribute names so that they correspond to the old version. Furthermore, if the request involves writing to the database (e.g., `DELETE`, `REPLACE`, `INSERT`) we duplicate the original query on the shadow table—a mechanism also known as log propagation [23]. The query rewriting is straightforward and relatively costless since we change the column and table index rather than rewriting the query.

To maintain consistency, the operations of the background thread and the main SQLite thread (or "foreground 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 implementing this by having the background thread call `yield()` before each `INSERT` request to relinquish the CPU to any waiting foreground thread.

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 checks and lazy updates introduced in Section 3.2 are actually implemented. To ensure that R and R' are consistent, all modification requests (`INSERT`, `UPDATE`, `DELETE`, `REPLACE`) on those R's that had a change in schema are replicated into R' as well; the technique is similar to log propagation [20, 23, 17]. Modification requests on new tables or tables whose schemas were not changed proceed right through, without replication. These techniques effectively implement the $R_S \cap \mathcal{R}_\Delta = \emptyset$ check. Moreover, in the implementation we do not actually store a version ν with each tuple. Rather, the background thread keeps track (using SQLite's internal `ROWID`) of the tuples that have been, or are yet to be, converted from version 1 to version 2; this reduces overhead.

Error tolerance. Our system is fail-stop and transactional for the immediate update phase, i.e., if an errors occurs during this phase, or power goes down, **UPDATEDB** returns an error and the database remains unchanged. 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, we resume the lazy update process the next time SQLite starts, based on the update information in our persistent table.

6. EVALUATION

To evaluate the impact our approach has upon on-the-fly updateable databases, we performed cloud- and standalone-system experiments based on real-world schema update scenarios and database benchmarks. The scenarios and benchmarks are based on actual updates and queries we observed in practice (Section 2.2). While similar schema evolution work [25] has also measured system usability (user effort saved for query rewriting), such a measure does not apply to our system, as there is no query rewriting in our model—when the application code is updated, the new queries are part of it.

6.1 Schema Update Scenarios and Benchmarks

Schema update scenarios. Table 3 describes each scenario. The "old schema" and "new schema" columns show a summary of old and new database schemas i.e., total number of tables and attributes. The "changes" columns show how the schema has changed. For example, scenario 1 starts with 2 tables and 6 attributes in total, and the schema update adds 3 attributes; scenario 6 starts with 10 tables and 46 attributes, and the schema update adds a table, renames an attribute, and changes the type of an attribute; scenario 10 starts with 8 tables and 26 attributes, and the schema update adds 6 attributes and changes the type of an attribute.

Benchmark configurations. To quantify the on-the-fly update overhead, we measured query completion time, memory, and disk usage in two configurations: $\mathcal{R}^{1\to 2}$, i.e., create and populate the database at schema version 1, and then perform an update that changes the schema from version 1 to version 2; and $\mathcal{R}^2$, i.e., create and populate the database directly at schema version 2.

Query benchmarks. Prior work in enterprise-grade schema evolution has used databases containing a total 2.13 million tuples [9]. Therefore, to effectively compare our work with prior efforts, each table we used in our experiments was populated with 1 million tuples, which means the entire database contained between 1 million and 11 million tuples, depending on the scenario. Our performance experiments

Table 3: Schema update scenarios: old schemas, new schemas, and changes to tables/attributes.

Scenario ID	Old Schema		New Schema		Changes						
	Tables	Attributes	Tables	Attributes	Tables			Attributes			
					add	delete	rename	add	delete	rename	type change
1	2	6	2	9	-	-	-	3	-	-	-
2	1	4	1	6	-	-	-	2	-	-	-
3	3	18	3	21	-	-	1	3	-	-	-
4	2	7	2	5	-	-	-	-	2	-	-
5	1	7	1	7	-	-	-	1	1	-	1
6	10	46	11	55	1	-	-	-	-	1	1
7	3	18	3	18	-	-	-	-	-	-	1
8	3	17	3	17	1	1	-	-	-	-	-
9	10	46	11	55	1	-	1	-	-	1	-
10	8	26	8	32	-	-	-	6	-	-	1

focus on four aspects: update time, query performance, disk space overhead, and memory footprint. To measure the temporary overhead our approach imposes, i.e., how much slower the queries run while the database is being updated, how much memory and disk space is used, we compared the time to run 1,000 queries against the new schema for the database ($\mathcal{R}^2$) and 1,000 queries to be executed during the lazy update ($\mathcal{R}^{1\to 2}$). The queries model the statement frequency observed in our study on open source programs: 40% `SELECT`, 30% `INSERT`, 20% `UPDATE`, 10% `DELETE`. The database contents, as well as the new tuples and attributes used in `INSERT` and `UPDATE` have been created randomly.

6.2 Experimental Setup

To quantify the impact of our approach, on both cloud and desktop/server systems, we performed measurements, and report results, in two settings: *Amazon AWS* and *Standalone*. In particular, we are interested in how much our approach increases availability, and in quantifying the availability vs. overhead trade-off. The overhead is especially relevant in the cloud setting, as the cloud uses a pay-for-service model; we found however, that this overhead is small and temporary. We now proceed to describing the experimental setups and results.

Amazon AWS. The cloud experiments were conducted on a Amazon EC2 Micro instance. The Micro instance offers 613MB of RAM, 8GB of disk and up to 2 EC2 Computing Units (for short periodic bursts). One EC2 Computing unit provides the equivalent CPU capability of a 1.0–1.2GHz 2007 Xeon processor. The system ran 32-bit Linux with a Red Hat distribution (the Amazon EC2 default image for a Micro instance).

Standalone. We conducted the standalone experiments on a two-CPU, quad-core 2.33GHz Xeon system with 12GB of RAM and a RAID5 array of three Western Digital RE3 1TB@7200 rpm hard drives. The test system ran 32-bit CentOS 5.5, Linux kernel version 2.6.18.

In both setups, SQLite was compiled with `gcc`, flags `-g -O2` (these are the out-of-the-box compilation options for the official SQLite distribution). Time measurements were performed using the CPU's time stamp counter (TSC).

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

6.3 Results

Table 4: Amazon AWS: update time.

Scenario ID	UPDATEDB (msec)	Update completion (msec)
1	228	228
2	119	119
3	98	98
4	110	4,063,530
5	87	6,206,550
6	586	4,637,834
7	166	4,273,845
8	65	65
9	572	572
10	593	6,713,257

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. Columns "**UPDATEDB**" in Tables 4 and 5 show these times for each schema change scenario, in the cloud and standalone configurations, respectively. As we can see, in the cloud we could always complete an **UPDATEDB** in at most 593 milliseconds, which is crucial for ensuring our approach disrupts the clients as little as possible. If we assume one update per week, these 593 msec/week translate to 30.8 seconds of unavailability per year, which is more than *six nines* (99.9999%) availability. For the standalone system, the unavailability is even smaller: we could always complete an **UPDATEDB** in at most 251 milliseconds.

The last columns of Tables 4 and 5 show update completion time, i.e., the time it takes for the lazy update to finish and the entire database to be converted; this is the time a client would have to wait without on-the-fly updates. As we can see in Table 4, in the cloud setup, without on-the-fly updates some clients would need to wait 6,206 seconds (103 minutes), which is unacceptably long—assuming one update per week, this translates to less than *two nines* (99%) availability. In the standalone case (Table 4), the longest unavailability is 265,058 msec (4 minutes).

Query performance. We want to keep the client-perceived query processing overhead low for the period the database is

Table 5: Standalone: update time.

Scenario ID	UPDATEDB (msec)	Update completion (msec)
1	61	61
2	32	32
3	115	115
4	33	186,436
5	47	265,058
6	209	209,171
7	81	195,251
8	72	72
9	251	251
10	193	230,767

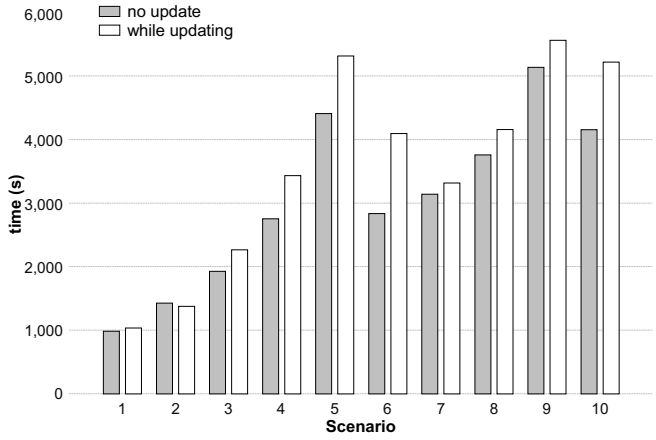

Figure 3: Amazon AWS: query execution time in the *no update* case ($\mathcal{R}^2$) v. query execution time *while updating* ($\mathcal{R}^{1\to2}$).

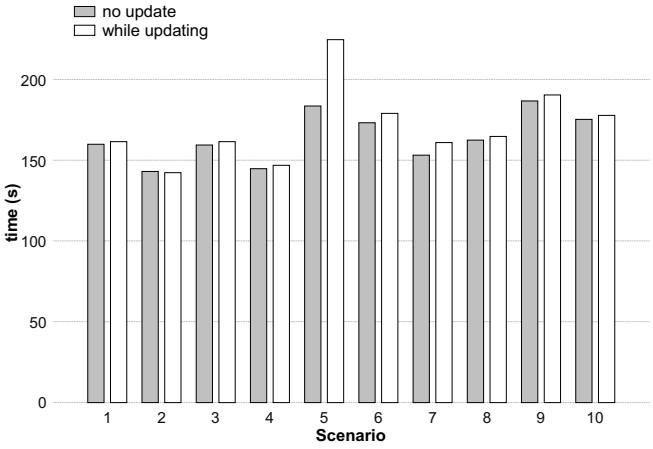

Figure 4: Standalone: query execution time in the *no update* case ($\mathcal{R}^2$) v. query execution time *while updating* ($\mathcal{R}^{1\to2}$).

Table 6: Amazon AWS: overhead.

Scenario ID	Query (%)	Disk (%)	Memory (kB)
1	4.85	0	0
2	-3.69	0	0
3	14.93	0	0
4	19.77	30.23	29,824
5	17.06	67.69	29,824
6	30.76	8.05	29,628
7	5.34	7.12	29,628
8	9.63	0	0
9	7.65	0	0
10	20.40	5.24	29,628

undergoing lazy updating. Therefore, we measured the time to complete 1,000 queries in the no-update case ($\mathcal{R}^2$) and the completion times when the 1,000 queries are executed during the lazy update of the database ($\mathcal{R}^{1\to2}$). Figures 3 and 4 show the result of our benchmark. In each figure, the x-axis represents the scenario id, the grey bar is the no-update query completion time ($\mathcal{R}^2$) and the white bar is the lazy-update query completion time ($\mathcal{R}^{1\to2}$). Columns 2 of Table 6 and Table 7 show the same information but expressed as time increase, in percents, for $\mathcal{R}^{1\to2}$ when compared with $\mathcal{R}^2$. For example, in scenario 1, the lazy update version took 4.85% longer (cloud) and 1% longer (standalone) to complete the benchmark; in scenario 2 the values are slightly negative due to jitter and 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 (at most 6,206 seconds in our scenarios).

Disk space footprint. The additional disk space used for lazy updates (e.g., shadow tables) will temporarily result in higher disk space usage. To get an idea of the temporary disk overhead our approach imposes, we measured the size of the on-disk database file in the $\mathcal{R}^2$ and $\mathcal{R}^{1\to2}$ configurations; for the $\mathcal{R}^{1\to2}$ configuration, we measured disk usage just before deleting the shadow tables. We found (columns 3 of Tables 6 and 7) that the on-the-fly update case uses between 0% and 68% more disk space than the unmodified SQLite (we present percentages, rather than actual disk space because the overhead depends on the database size). However, this overhead is temporary; after completing the lazy update, the disk usages in both configurations are identical. Note that disk usage would be higher if we used standard migration techniques, as in that case we would need space for both the old ($\mathcal{R}^1$) and new ($\mathcal{R}^2$) databases.

Memory footprint. The additional memory space used for lazy updates will temporarily lead to higher SQLite memory footprints in cases where shadow tables are used. To measure the temporary overhead, we compared the additional *peak* memory footprint in the $\mathcal{R}^{1\to2}$ case compared to the $\mathcal{R}^2$ case. This information will give us an idea of maximum additional memory requirements, which is especially important for cloud settings and mobile devices where memory is at a premium. We obtained peak memory usage from the `VmPeak` field of `/proc/<pid>/status` as reported by the Linux kernel. In columns 4 of Tables 6 and 7 we see that the extra memory required ranged from 0 KB to 29,628 KB.

Table 7: Standalone: overhead.

Scenario ID	Query (%)	Disk (%)	Memory (kB)
1	1	0	0
2	-0.5	0	4
3	1.26	0	0
4	1.39	28.37	28,744
5	22.39	66.27	28,744
6	3.35	7.5	28,840
7	5.04	6.8	28,744
8	1.44	0	0
9	1.97	0	0
10	1.40	4.4	28,744

CPU utilization. There is a trade-off between completing the lazy update quickly and having a responsive system, as foreground and background threads compute for hardware resources (mainly CPU). On one hand, we want the foreground thread that carries out post-update client queries to be slowed down as little as possible. On the other hand, slowing down the background thread will delay completing the update. We measured the parameters of this trade-off in the cloud. Thanks to Amazon EC2's monitoring facilities, we found out that the CPU utilization is 100% during the lazy update process, which means neither the client queries, nor the update were slowed down unnecessarily. For the standalone system, the lack of virtualization makes collecting accurate information about CPU utilization difficult, hence we did not attempt it.

Verifying correctness. To verify that our on-the-fly schema update implementation is correct, for each scenario we compared the results of running a query on $\mathcal{R}^2$ with the results of running a query on $\mathcal{R}^{1\to2}$. We also compared database contents between a database constructed at version 2 ($\mathcal{R}^2$) and the database contents after finishing the update from version 1 to version 2 ($\mathcal{R}^{1\to2}$). To do so, we saved query results and database contents in text form, and used the Unix `diff` command to confirm that the texts corresponding to $\mathcal{R}^2$ and $\mathcal{R}^{1\to2}$ were identical.

Limitations. Our implementation does not yet support more complex SMOs such as vertical/horizontal split or merge. We believe that adding support for such SMOs is feasible and plan to explore it in future work.

7. RELATED WORK

Online schema changes. $\mathcal{PRISM}$ [9] is a system aimed at replacing the current manual schema evolution process with a fully-automated one. The database admin 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); old queries incur a permanent performance overhead. For now we do not require that SMOs be specified, but provide automatic conversion to the new schema, though in the future we plan to support more complicated schema changes which will require programmers to specify SMOs. 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 queries built in (Section 3.1). 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 sequence.

Ronström [31] 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. For now our system does not support such complex schema changes (based on our empirical study, these complex schema changes are infrequent) though we plan to support them in the future. 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 [20] 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. Our safety guarantee is somewhat similar to theirs, though the set of schema changes we support differs from theirs. 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. [8] 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 2).

Sjøberg [33] 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 column additions/deletions and table additions/deletions.

Schema versioning, schema matching and schema mapping. Schema versioning and temporal querying approaches such as $\mathcal{PRIMA}$ [25] or timetravel in Ganymed [28] store data at multiple versions and allow queries that span multiple schema versions. Schema mapping systems such as MACES [41] or ToMAS [37] allow heterogeneous systems that assume different schemas to interact properly via mappings. Schema matching [29] and ontology mapping [18] address the problem of accessing data where the client and server formats (or languages) are different, by providing matching or mapping functions between the two formats.

These approaches are complementary to ours, because they target a different point in the design space. First, their power comes at a cost—using dedicated systems or separate applications—hence unlikely to be easily integrated in embedded, mobile, desktop, or real-time applications. Second, our applications do not 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 schema mapping/matching, or query rewriting.

Lazy updates. Lazy updates were used by other researchers in prior work, in the context of object-oriented databases [12, 5] and persistent object stores [7]. Automatic object upgrades on top of Thor [7] 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. For example, MySQL [3] and Microsoft SQL Server [22] 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 [17]; however, as we explain next, these changes are not performed on-the-fly, which affects availability. 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 tuples are inserted/updated [13].

Edition-based redefinition in Oracle 11g Release 2 [27] 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 [14] 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.

8. CONCLUSIONS

In this paper we present an approach for march-forward, on-the-fly schema evolution that explores a different point in the design space compared with current schema evolution approaches. This work was motivated by the non-stop requirements of many cloud, desktop, and server applications; and by the ubiquity of SQLite and the march-forward nature of applications using it. We have implemented support for on-the-fly changes to schemas by extending SQLite in a client-transparent manner. We have evaluated our approach on both Amazon Web Services and standalone server setups, found that support for online schema updates does not significantly impact application performance and any overhead is transient. Our system has the potential to improve software availability and user experience for a variety of non-stop applications that use persistent data; moreover, it allows cloud service providers to sustain a high update cadence without violating SLAs.

Acknowledgments

We thank Vassilis Tsotras and the anonymous referees for their helpful comments on this work. This research was supported in part by the United States National Science Foundation grants CCF-0963996 and CCF-1149632.

9. REFERENCES

[1] BiblioteQ. `http://biblioteq.sourceforge.net/`.

[2] High availability. `http://en.wikipedia.org/wiki/High_availability#Percentage_calculation`.

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

[4] J. Arnold and F. Kaashoek. Ksplice: Automatic rebootless kernel updates. In *EuroSys*, 2009.

[5] J. Banerjee, W. Kim, H.-J. Kim, and H. F. Korth. Semantics and implementation of schema evolution in object-oriented databases. In *SIGMOD '87*, pages 311–322.

[6] P. Bhattacharya and I. Neamtiu. Dynamic updates for web and cloud applications. In *Proceedings of the 2010 Workshop on Analysis and Programming Languages for Web Applications and Cloud Applications*, APLWACA '10, pages 21–25, 2010.

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

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

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

[10] T. Dumitraş and P. Narasimhan. No downtime for data conversions: Rethinking hot upgrades. Technical Report CMU-PDL-09-106, CMU, 2009.

[11] T. Dumitraş and I. Neamtiu. Cloud software upgrades: Challenges and opportunities. In *MESOCA'11*.

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

[13] C. Friske. DB2 online schema changes - what's new in DB2 version 8. *SHARE*, Session 1323, February 2004. ftp://ftp.software.ibm.com/software/data/db2/zos/presentations/v8-new-function/online-schema-evolution-imtc-2004-friske.pdf.

[14] H. Garcia-Molina and K. Salem. Sagas. In *SIGMOD Conference*, pages 249–259, 1987.

[15] Google. Google Apps service level agreement, 2012. `http://google.com/apps/intl/en/terms/sla.html`.

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

[17] 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.

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

[19] D.-Y. Lin and I. Neamtiu. Collateral evolution of applications and databases. In *ERCIM Workshop on Software Evolution/International Workshop on Principles of Software Evolution*, pages 31–40, August 2009.

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

[21] V. Mandalapa. A framework for understanding schema evolution in web information systems. Master's thesis, Arizona State University, 2009.

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

[23] 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.

[24] The monotone distributed version control system. `http://monotone.ca`.

[25] 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.

[26] Mozilla Foundation. The Mozilla Project. `http://mozilla.org`.

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

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

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

[30] J. F. Roddick et al. Evolution and change in data management — issues and directions. *SIGMOD Rec.*, 29(1):21–25, 2000.

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

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

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

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

[35] SQLite Team. Most widely deployed SQL database. http://www.sqlite.org/mostdeployed.html.

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

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

[38] The Vienna RSS/Atom reader. `http://http://vienna-rss.org`.

[39] Wikimedia Foundation. MediaWiki 1.5 upgrade, 2005.

[40] S. Wu and I. Neamtiu. Schema evolution analysis for embedded databases. *Third Workshop on Hot Topics in Software Upgrades (HotSWUp'11)*, pages 151–156, April 2011.

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