Toward On-line Schema Evolution for Non-stop Systems

Amol Deshpande and Michael Hicks
University of Maryland, College Park

September 7, 2005

1 Introduction

Schemas are a central component of any database system, and have been an area of intense research from the very beginning of the field. In the past, this work has mainly focused on using conceptual modeling techniques to generate schemas from application requirements, schema normalization, schema mapping and schema matching in data integration systems. Dynamically changing schemas, on the other hand, have received much less attention, especially in the context of relational database systems. We conjecture three reasons for this: (1) a perception that schemas do not need to evolve; i.e., that they are essentially static; (2) a belief that if they do need to evolve, it can be done off-line; and (3) that even if on-line evolution was desirable, it is too hard to implement. With the increasing number of web services, long running transactions, and other applications that cannot afford to stay offline for even minutes, and with increasing DBA-less installations of databases (e.g. "Digital Home"), we believe the first two of these reasons are no longer valid. This motivates our interest in the third issue.

Supporting on-line evolution is non-trivial. Almost every aspect of a running database system is tied to the schema of the database. Most important are obviously the application programs (many of which are outside the domain of the database system) that interact with the schemas directly, using queries over the schemas. Many of these programs may be affected by a change to the schema. Physical data structures like indexes are tied to the schema as well, and may have to updated if new tables need to be created. If the schema change is to be affected while the database system is running, many of the internal components (e.g., query processor component) may be affected by a schema change (especially in presence of live "cursors"). If the changes require creating or removing tables, then the concurrency/locking components are also affected.

In this paper, we propose an approach to on-line schema evolution that coordinates the updating of the schema with the updating of applications, employing two mechanisms. First, applications compatible with the new schema are

modified on-the-fly using a technique called dynamic software updating (DSU). These software updates are coordinated, at a time determined in part by an off-line program analysis, with an update to the database, preventing applications from querying the old, invalid schema. Second, database contents are migrated *lazily*, as required by the applications programs, in a fashion analogous to "page faults" in the virtual memory subsystems. Except for small local changes (e.g. adding a column), this would be non-trivial in relational database systems.

2 Example

To make the need and process of on-line schema evolution more concrete, we present an example inspired by `http://www.agiledata.org/essays/databaseRefactoring.html`. Suppose our database schema currently defines an Address table:

```
TABLE Address
  AddressID: integer   Street:   char{40}    City: char{20}
  StateCode: char{2}   ZipCode:  integer
```

Now suppose that we wish to alter this schema to support foreign addresses. For example, Canadian addresses use an alphanumeric postal code, like R2D 2C3, instead of zip codes like 90210. Because the ZipCode column is numeric, the definition of the Address TABLE needs to be changed. We decide to add two fields, PostalCode and Country, and remove the field ZipCode. Existing records in the table can be converted to the new schema by simply generating a string from the existing ZipCode and storing it the PostalCode entry, and by storing the default value of Country as "USA".

A typical way of effecting this kind of change is to perform it off-line. Following `http://www.objectivity.com/WhitePapers/schema.shtml`, the process would be:

1. Modify the application to use the new schema. For example, if we had an application that reads orders from the database and prints out address labels to ship them to, this application would be changed to refer to the new columns. Moreover, the data entry application would be changed to add an additional Country field (e.g., in its HTML form), and the formatting rules for postal codes would be relaxed.

2. Specify a *conversion function* to convert each record in the old schema in the new schema (as described above).

3. Shut down the database and applications, halting current processing.

4. Perform monolithic data conversion: read the old data from the database; (2) convert each record using the conversion function; and (3) write the resulting database with the new schema.

5. Restart the database and the application.

There are three main problems with this approach. First, it terminates existing transactions, which is inconvenient for users and possibly bad for business. Second, any soft state in applications, like caches for improving performance, is lost when applications are shut down. Third, it makes the database unavailable for the time it takes to evolve the existing database. It may be possible reduce this pause by doing some of the evolution off-line, on a checkpointed copy, and then only deal with the records that have changed or been added since that time, once the on-line database is shut down. However, for large databases, this could still be a significant period of time.

A possible solution is to use views. In particular, we can relax the need to update the old applications by allowing them to view the new schema as if it were the old. In this case, the view would only be defined for those addresses whose Country was "USA" simply because it is not possible to convert from a generic postal code back to a numeric zip code. This may be acceptable in some cases, but clearly some applications will act incorrectly. For example, an application that wishes to send a mailing out to all past customers will only send it to those in the US, and an application that wishes to count total past customers will miss any new international ones. Therefore, while views may be useful in some cases, we do not believe they are a solution in general.

3 Enabling On-line Schema Evolution

The goal of an on-line schema evolution approach is to relax points 3, 4, and 5 in the scenario above, as follows:

3′. Having modified the application to use the new schema, we now have two versions of the application. From these two versions, we develop a *dynamic patch* that can be used update a running instance of the old application to an instance of the new one.

4′. We upload the dynamic patches to the applications using the DB, and the database patch to the database. When the applications reach a suitable *safe point*, the database patch is triggered, and the applications are patched. Updating at a safe point ensures that the newly-updated applications will always refer only to the new schema in their queries, never the old.

5′. With each access to the database, the DBMS will modify the on-line contents as necessary, depending on the query. For example, if an application in our scenario only performs a query on the street address of each element in the Address table, the fact that we have removed the ZipCode field will be of no consequence. Indeed, the DBMS could start converting the contents of the database concurrently with the program, if desired. However, when a query is submitted that wishes to group together those addresses in a particular postal code, then the table will need to be converted as defined in the database patch.

To realize this approach, there are some challenging problems to solve. First, we need a way to dynamically update applications, which requires sophisticated run-time support, and we have to ensure that following an update, the application will always use the new schema, which requires program analysis or run-time checking. Second, we need a way to actually change the on-line contents of the database, and to do it as lazily as possible, to reduce pauses. We consider each of these elements in turn.

3.1 Dynamic Software Updating

Run-time support for dynamic software updating is reasonably well-understood. In particular, a special compiler can be used to insert indirections to permit functions and typed data values to change over time. In our past work, we have developed DSU systems that are extremely flexible: nearly any change one can express to the source code can be realized on-line through a dynamic patch. Experiments with various servers show only a 1-3% slowdown due to compiler support for updateability.

An important aspect of software updating is *timing*. For on-line schema evolution, our goal is to ensure that following an application update, it will never access the DB at the old schema. To ensure this, we can analyze a program that uses the database-to-be-changed, and its dependencies on the current schema will be discovered, for each program point. For example, say function `f()` performs an SQL query on the Street column of the Address table. This query places a constraint on the way Table may be updated: Address may not be changed to remove Street or change its type while `f()` is executing up through the the query. To do so would cause the query to fail, or worse, to succeed but result in incorrect program behavior. Conversely, an update to the schema that is *consistent* with the discovered dependencies is *safe*. We have used an analysis like this to ensure that software updates do not cause type errors due to poor timing, and we have found it to be flexible and efficient. Extending to the database will also require considering cursors and other forms of dependence.

Because the DBMS and its programs run concurrently, we need to find a time at which the changes to the programs and the schema are sure to be consistent. The simplest approach is to force each program to synchronize at a safe update point. When all programs have reached such a point, then they can be updated and the schema change process can be started. The challenge is making this simple idea scale. In particular, a serious concern is deadlock: if one program is waiting on an update point, it may fail to release locks needed by another program (e.g., locks held because of a currently-running transaction). Static analysis and dynamic detection techniques exist to prevent or mitigate deadlocks, and we would apply them to this situation.

3.2 Lazy Data Migration

To avoid the long pause that could result if the database contents are converted at update-time, we propose to apply the conversion function lazily, as driven by the application:

1. The DB patch creates a skeleton relation over the new schema in the database. Any new tuples are inserted directly into the new relation.

2. When a query is posed over the new schema, the DBMS checks if the table is up-to-date. If not, it converts those tuples required to answer the query and stores them in the new schema. Then it answers the query. For example, if a query asking for only addresses from zip code 10001 arrives, the DBMS can fetch those tuples from the old relation, convert them to the new format, and insert the tuples into the new relation. After this is done, we can answer the query itself. The steps of converting the data into new format, and answering the query, can be merged in many cases thus saving us redundant disk accesses. On the other hand, if a query asking for only addresses from PostalCode "R2D 2C3" arrives, only the newly-inserted data will satisfy such a predicate, so this query can be answered without converting any data from the old relation.

3. Will we need to use *semantic caching*-like techniques to reason about the data that has already been converted. For example, after the above query has been answered, we can tag the new relation with predicate "ZipCode = 10001" thus specifying exactly which data has already been converted. Doing this correctly and efficiently is one of the biggest challenges in this work.

If the techniques outlined in the above step fail to provide enough information to reason about the data, we always have the (drastic) option of converting the entire relation. Though this still provides some level of "laziness" (not all the relations need to be converted at once), this is clearly not acceptable in general, and we hope to avoid having to fall back to this option in most cases.

4 Toward On-line Schema Changes

There is beginning to be some interest in supporting on-line schema changes, but none of it is as complete or as ambitious as what we have proposed for relational databases. For example, SQL:1999 supports a series of ALTER directives, but these only permit fairly simple changes to the schema. DB2 V8 supports some schema changes, and could support the simple change we have described above, but does not consider the impact of the change on applications. Indeed, the combination of our DSU techniques with DB2 could be a reasonable starting place for this work. Finally, OODBs often do provide some support for on-line evolution, but this is made simpler by the fact that data and code are stored

and accessed together in the database. Our approach is able to bridge code and data changes when these elements are stored and accessed separately.

We are excited about the benefit of on-line schema evolution: because data and applications can be changed on the fly, services become both more available and more agile. To focus this research agenda, we wish to better understand how schema changes are done in practice, and what the commonly occurring schema changes are. We plan to talk with industry customers to identify these issues.