RC24934 (W1001-056) January 15, 2010
Computer Science

IBM Research Report

Everett: Providing Branch-Isolation for a Data Evolution Service

Avraham Leff, James T. Rayfield
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Everett: Providing Branch-Isolation for a Data Evolution Service

Avraham Leff
IBM T.J. Watson Research Center
POB 704
Yorktown Heights, NY, USA
avraham@us.ibm.com

James T. Rayfield
IBM T.J. Watson Research Center
POB 704
Yorktown Heights, NY, USA
jtray@us.ibm.com

Abstract

One benefit of Software-as-a-Service (SaaS) is the ability to rapidly deploy iterative improvements without requiring users to upgrade the application on their machine. However, the need to rapidly "develop and test" different versions of an application implies that developers need branch isolation to protect the system from local changes to both data and meta-data in the same way that they traditionally use branch-isolation to protect the system from source-code changes.

Providing branch-isolation for source-code changes has well-known solutions, but these solutions do not extend well to providing isolation for changes to data and meta-data. EVERETT *provides developers the ability to safely – and concurrently – change database values with new business logic or evolve data schema in various ways while sharing the same database. This paper discusses how* EVERETT *provides transparent branch-isolation for a data-evolution service, the algorithmic and data-structure trade-offs that we examined, and evaluates the success of our approach.*

1. Introduction

1.1. Motivation

SaaS development is well-suited for the so-called "agile" development style because successive sets of feature improvements can be deployed with no effort from the existing user base. It becomes possible to get rapid feedback about the usefulness of a given feature, allowing an enterprise to decide whether to further refine or remove the feature. Typically, new application features require new business logic that may affect existing data values as well as evolving the original data-schema. For example, consider an enterprise-directory service that stores employee names and email addresses. Enhancing the service by adding an HR appraisal feature requires an "org-chart" in order to implement the approval chains. Adding the new business logic and user interface requires that employee records be augmented with a "my manager" field. Similarly, changing the "hire date" to a Y2K-compliant format, or allowing employees to have multiple phone numbers, requires evolving the data-schema together with the application. Developers of one feature need their data and meta-data to be protected from developers of other features.

Version control systems such as CVS [13] and SVN [7] have traditionally protected source-code through *branch isolation*, enabling developers to "branch" to create new features and then "join" when the feature is complete. However, there are no general-purpose solutions for similar branch-isolation for changes to application data and meta-data. Changing an existing data-schema is a non-trivial operation, even if the data-evolution is purely additive (e.g., a new "my manager" field). It becomes very dangerous, if not impossible, for two versions of an application to use the same database when the new semantics differ from the existing data-schema semantics (e.g., changing measure units or entity cardinality). Even when branches use the same data-schema, the differing business logic may associate different semantics with a given data-value. As a result, code developed in one branch may easily destroy data created in another branch. For this reason, enterprises typically set up separate test databases for separate feature-sets; physical separation is considered to be necessary to protect each branch's data. However, setting up separate databases introduces considerable "friction" into the rapid prototyping process because developers have to create, manage, and populate their private database with the appropriate subset from the production database. Still worse, if the enterprise decides to incorporate a new feature-set, the evolved data-schema must be manually merged with the existing data-schema.

1.2. Everett: Branch-Isolation for Data

EVERETT applies the concept of branch-isolation used by version-control systems such as CVS and SVN to the data portion of an application. Version-control systems allow developers to safely "branch" code and modify it in isolation of one another while storing their code in the same physical repository. Developers can later merge different versions of the code to create an application that includes features developed independently in the different versions. By analogy, EVERETT is a data evolution service that allows developers to concurrently evolve data-schema in different ways, and to store different data values, while using the same physical database. (The name of our project alludes to its support for concurrent existence of multiple branched histories [14].) Most importantly, these features are transparent to a given data API so that developers do not have to first learn a new API in order to use EVERETT. EVERETT then allows different versions of the evolved data-schema to be merged into a common schema.

EVERETT provides two functions: *branch isolation* and *data-schema evolution*. The challenge in developing EVERETT was the development of algorithms and data-structures such that the data-service provides reasonable time and space performance while providing developers with their private "data sandbox". We have initially focused on branch-isolation as we plan to provide data-schema evolution using branch-isolation to protect application meta-data. This paper therefore focuses on branch-isolation, although we present our approach to data-schema evolution in Section 2.

Before proceeding with a discussion of how EVERETT provides branch isolation for data, we explain how EVERETT relates to other work in the data-versioning space (Section 1.3). We discuss the branch-isolation algorithms and data-structures in Section 3. In Section 4 we evaluate our results and summarize EVERETT's status.

1.3. Related Work

Much work was done in the 1980-2000 time frame in the area of temporal databases [5]. Although temporal databases typically maintain a record of past (superseded) record values, they do not need to maintain multiple *concurrent* branches, and thus do not need to solve many of the problems addressed by EVERETT.

For example, a rich temporal database will maintain data along two time axes: *valid time* and *transaction time*. This allows clients to query about past values of data. However, all queries from all clients will be applied to that same set of temporal data. Different clients will not see different results for the same (temporal) query, and changes made by a client will immediately visible to other clients. Also, schema changes made by a client will immediately be visible to other clients. In contrast, EVERETT maintains isolation of data values and schema between clients in different branches. If a client executing in its own branch makes a data or schema change, the change will not be visible to clients in other branches until a join has taken place. Conversely, unlike temporal databases, EVERETT does not allow queries about non-current values of data: all queries are based on the most recent changes made in the branch being queried.

Rails Migrations [8] provide a systematic way of organizing changes made to the database schema by multiple developers, and for developers to merge schema updates to get to a common state. However, it is somewhat ad-hoc; for example, the migrations are not necessarily executed in the same order for each developer, no checking is done for conflicts between developers, and some tasks must be done outside the migrations framework (e.g., modifying a previously-created migration). This approach is more suited for a situational development group [9]. Most importantly, Rails Migrations do not provide a mechanism for sharing database data. Each user must create and maintain their own copy of the database. With EVERETT, the existing database transparently provides developers with a private copy of the existing data, with no effort required from the developers.

Other projects have focused on methodologies for describing schema changes and for converting data between different schema. The Clio [4] system allows the expression of declarative schema mappings between XML and/or relational schema. The PRISM Workbench [2] provides a language for describing schema changes, query conversion, and data migration. However, neither supports maintenance of multiple concurrent schema by multiple users in the same database system.

2. Application Life-Cycle

This paper focuses on the branch-isolation function that EVERETT provides to a data-evolution service. This section therefore describes the "big picture" of how EVERETT can be used throughout an application's development and test life-cycle.

Version control systems have made developers familiar with the "branch and merge" application life-cycle.

Application development proceeds on a "trunk" while experimental features are developed along "branches" that may optionally be merged back to the trunk. The code in the trunk and individual branches are isolated from one another, giving the freedom to experiment with new features since a developer knows that her changes will not affect anyone else.

EVERETT gives developers exactly the same freedom with respect to the data-schema and data-values portion of an application. A data branch is identified by BRANCHID, and typically has a 1:1 correspondence to a code branch. The state of a data branch is the state of the parent branch at the branch point, together with the set of changes made within the branch. Besides changing data values, developers may add, remove, or change properties of the branch's data-schema. For example, they may add a "my manager" property, replace a "phone" property with a "set of phones" property, and change the units of a "year" property from two characters to four. Because a BRANCHID distinguishes one branch from another, EVERETT can make new properties immediately available to a given branch, while ensuring that code from other branches simply don't see the new data-values or properties. Similarly, EVERETT will raise an exception when code in the current branch accesses properties that have been removed from the current data-schema. Finally, EVERETT supports modification of existing property types by allowing developers to specify "transfer functions" that convert between data values encoded in the parent branch and data values encoded in the current branch. (Transfer functions are similar to the "updater subroutines" used by Multics [10].) The use of transfer functions allows values in the parent branch to be transparently migrated for use by code in the new branch without manually copying and modifying the old data.

Code in the branch accesses the branch data which, at the branch point, is identical to the set of data values that are visible to the parent branch (e.g., the trunk). Importantly, branched applications initially behave exactly the way they did before the branch since the code and data are identical to the pre-branch state. As data are added, removed, or modified, branch queries will begin to return different result-sets since changes made in other branches are isolated from one another.

When a decision is made to merge a branch back into the parent branch, the branch code is first merged into the parent branch using standard version control techniques. EVERETT merges data-schema in the following way:

- Added properties are now made visible to parent-branch code.
- Removed properties are hidden from parent-branch code (they are not physically removed from the database, because code in other branches may still be using them). Declaring such removed properties as completely obsolete can be seen as an administration function which is invoked when existing branches are no longer active. This is needed only to save database space, since old properties can remain hidden forever without affecting correctness.
- By using transfer functions, properties modified in the branch-schema are converted over a period of time to the new format. During the transition, individual property values may be converted on demand as needed by running code. "Reverse" transfer functions may be used, as needed, to convert property values back to the old format needed by code in pre-existing branches.

Note that – in contrast to *meta-data* – EVERETT discards data *values* from the child-branch during the merge. In general, enterprises will not want to use test data-values to replace production data-values. The semantics of situations where branch data-values *should* overwrite the parent data-values are application-specific, and are not easily specified to the lower-level EVERETT data-service. Previous work on on merging of database branch values may be found in [1].

3. Algorithms & Data Structures

3.1. The Problem

As mentioned above, this paper focuses on EVERETT's branch isolation function which we formally define as follows. Assume that developers can specify that a given set of code be associated with a development branch B_i. All data and meta-data that are read or written when this code executes are also associated with the same development branch. Also, when developing in B_i, developers can create a *child* branch B_j and thus create a branch tree rooted in B_0. The time at which B_j is created is a *branch point*: parent branch B_i development beyond the branch point is independent of B_j development.

With respect to branch data, the isolation semantics for the branch tree are:

1. B_j code can read parent-branch data that were created or modified up to the branch point.
2. Data created, modified, or deleted by B_j code are not visible to parent branches.

3. B_i data values that are modified or deleted by B_j supersede – for code associated with B_j and all of its child branches – the B_i values.

4. Data created or deleted by B_i code after a branch point are not visible to B_j.

5. Data values that are modified by B_i code after a branch point are not visible to B_j. B_j will continue to see the values that existed at the time of the branch-point (unless superseded by modifications made by B_j itself).

These rules define EVERETT's branch-isolation "CRUD" semantics. Providing branch-isolation is thus a prerequisite to providing data-schema evolution since the latter requires that branch meta-data have branch-isolation semantics.

3.2. The Solution

Copying the entire database for each branch trivially provides branch-isolation, but is very space-inefficient since most data in a parent branch will be read – but not modified – by child branches. For similar reasons, this solution is also time-inefficient since an entire database has to be created on a per-branch basis. EVERETT therefore uses a "log-structured" [3] (at the record level) approach in which CUD (but not query) operations performed by a given branch cause new records to be written to the database. This approach conserves both time and space since new records are not created until a branch actually modifies a datum. Providing the branch-isolation semantics for query, however, is more complicated.

In a non-branch environment, database indices enable efficient queries because the database updates them with the most recent values of the data. However, indexing a multi-branch database is not straightforward:

- Records created by a branch that is not a parent branch of a second branch may not be returned to a query issued by the second branch.

- Even if a child branch creates more recent records than a parent, they may not be returned to a query issued by a parent branch.

- Even if a parent branch creates more recent records than a child, they may not be returned to the child branch if they were created after the branch point.

In other words, the algorithms used for efficient query of single-branch databases cannot be easily adopted by EVERETT. Importantly, augmenting data records with BRANCHID information does *not* help solve the problem since a child branch's data-set is potentially comprised of data from its parent branches. In fact, we expect (because of typical read/write ratios) that most of a child branch's data are associated with a branch that differs from the branch issuing the query. Also, data associated with a parent branch may not be relevant to a branch query if that data were created after the branch point or were modified by the child branch.

To see this more concretely, consider Figure 1 which shows three items created in B_0, followed by a branch point in which B_0 creates B_1. After the branch point, B_0 modifies the value of item$_2$ from B to D; B_1 modifies the value of item$_1$ (from A to E) and deletes item$_3$. The branch-state of B_1 is:

- item$_1$, value E, record associated with B_1.

- item$_2$, value B, record associated with B_0 – even though it now has a value of D in B_0!

- item$_3$, value "deleted", record associated with B_1.

Because "query by BRANCHID" won't provide the right semantics, EVERETT instead introduces the concept of a *segment id* (or SID). A SID uniquely identifies a segment of a branch that has not (yet) branched further. Thus, in Figure 1 there are *three* SID values: SID$_1$ for the B_0 records before the branch point; SID$_1$ for the B_0 records after the branch point; and SID$_2$ for the B_1 records (which has not yet branched). We can now reformulate the semantics of branch-isolation query as "query against the most recent SID in the current branch". However, we cannot simply "query by SID" because most records are not updated in the current branch.

Therefore, in addition to augmenting every record with an appropriate SID, EVERETT uses two data-structures. The first, a *Branch-SegmentId* Map, associates every BRANCHID value with the most recent SID value of that branch. In Figure 1, B_0 is mapped to SID$_1$ and B_1 is mapped to SID$_2$. The second, a *SegmentId-Parent* Map, associates each SID value with the value of this segment's parent SID. In Figure 1, all SID values map to the same parent SID, SID$_0$.

EVERETT implements queries with the following algorithm:

1. Apply the query across all branches.

   This returns all records that *might* be part of the result set. For example, if B_1 executes a "select all" query, this step would return all records in the EVERETT database.

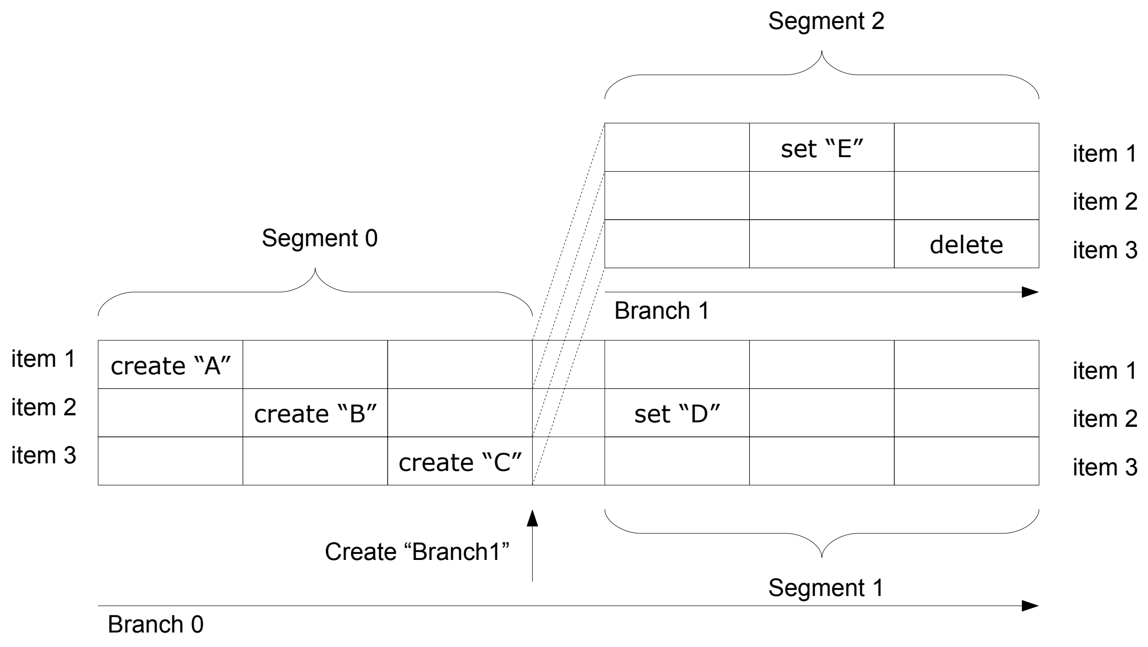

Figure 1. Versioning Data Using EVERETT: Branching and Branch Segments

2. Partition the result set based on common data id.

   In our example, there will be three groups corresponding to item_1, item_2, and item_3.

3. For each unique data id, fetch all SIDs for the id. That is, for all data items which satisfy the query *in some branch at some time*, fetch all the different versions.

4. For each unique data id, determine the most recent record (if any) that is "visible" to the client's branch, and add the record to the result-set if it matches the query.

   Visibility is determined by first extracting (from the *SegmentId-Parent* Map) a path of branch-segments that starts with the most recent segment of this branch and terminates at the branch-root. A record is visible to a given branch *iff* it contains a SID that is on the branch-segment path. The most recent record, then, is the first record EVERETT finds when traversing the branch-segment path.

   In our example, the result-set will contain item_1 with value E and item_2 with value B.

5. Return the filtered result set to the user.

Note that the EVERETT query algorithm includes two nuances. First, EVERETT must not return a record from a given partition if that record has been superseded by a DELETED record. In our example, even though item_3 appears in the partition, it will not be included in the result-set because the segment-path traversal for B_1 will encounter a DELETED record. Second, queries typically are not just "select all", and also specify some predicate ("select by last name"). EVERETT must ensure that a record is added to the result-set only if the most visible record with that data id actually matches that predicate. For example, consider a "query by department(T65)" where B_0 associates a given employee with department $T65$, but that employee's department was modified in B_1 to $B72$. That employee *will* appear in the partition, but must not appear in the result-set because it's most visible value (with department $B72$) no longer matches the predicate.

The following CUD implementation is used to support the EVERETT query algorithm:

- All records are augmented with SID and DELETED fields. (EVERETT removes these EVERETT-specific fields when returning records to the client.)

- CREATE and UPDATE: Add a new record containing values specified by the client; set the SID field to the one specified by the *SegmentId-Parent* Map, and set DELETED to FALSE.

- DELETE: Add a new record with values identical to the current version of the record; set the SID field to the one specified by the *SegmentId-Parent* Map, and set DELETED to TRUE.

3.3. Alternatives

We did consider alternative implementations to the query algorithm discussed above. Given the fact that code-branching systems motivated our EVERETT data-branching project, we first considered whether we could use the algorithms of systems such as CVS and SVN. Upon closer inspection we found that – at the implementation level – code-branching and data-branching require different approaches. Version control systems focus on version history of a record (file) to allow reconstruction of a given version or to show differences between versions. In contrast, databases focus on queries *across* records: i.e., find all records that match a given predicate. As a result, version control systems are inefficient from a time and space perspective for database versioning. In order to use a version-control system for EVERETT, we would have to reconstruct the entire state of a given version, and then execute the query against the reconstructed state. In the actual implementation of EVERETT, we do not reconstruct the entire version. Instead, we query the database directly using standard predicate criteria. Databases are already optimized for these kind of query operations, using indices and query planners. Then, we refine the result-set to apply the branch-isolation semantics.

Rather than first applying the query to *all* branches and then applying the visibility rules, one might plausibly successively apply the query only to the segments that lie in the segment-path from the current branch to the branch-root. The merge of these successive queries (including the refinements used by EVERETT to detect update and delete record "maskings") comprise the client's result-set. We chose not to use this algorithm because its performance depends on the number of parent segments that exist in the database. Recall that two SID values are created *every* time a branch is created: one that will be subsequently used by the parent-branch, and one that will be used by the child branch. The trunk will typically have a considerable number of SID values, each of which has to be queried in this algorithm. Depending on how many branches are actually created in a given environment, the EVERETT approach of applying multiple queries against the entire database will perform better than many queries against subsets of the database.

Listing 1. Client BranchContext Interface

```
/** Set the branch associated with the client
 * to the specified branch, which must
 * already exist.
 */
 public void setBranch(String branchID);

/** Create a new branch off the current branch,
 * but does not change the current branch.
 */
 public void createBranch(String branchID);
```

Listing 2. Server BranchContext Interface

```
/** Return TRUE if the branch exists,
 * FALSE otherwise.
 */
public boolean branchExists(String branchID);

/** Returns the name of the branch currently
 * associated with the client.
 * If the client has not invoked
 * "setBranch()", the system
 * returns the Trunk branch id by default.
 */
public String getBranch();
```

4. Status & Evaluation

EVERETT is currently available as a Java library for in-process use and as a web-service. Both versions are packaged as a data-service [6] that users configure at runtime to execute code and access data in a specific branch.

Developers invoke the BranchContext API (Listing 1) to either start a new branch (as a child of the current branch) or to associate current development with a given branch.

The server-side EVERETT implementation associates distinct BranchContext instances with clients on a per-thread basis. As part of the implementation, the server uses the additional BranchContext methods shown in Listing 2.

Once configured, the versioning provided by EV-

ERETT is transparent to users since the API is identical to the non-versioned API. EVERETT accesses the BranchContext associated with the client to determine the client's BRANCHID. As discussed above, EVERETT maps BRANCHID values to SID values, and thus implements the branch-isolation semantics.

We evaluated the isolation capabilities of EVERETT by comparing its performance relative to the *non-versioned* form of the data-service [6]. Specifically, we implemented the TPC-B benchmark [12], configuring it to use the EVERETT data-service with the code executing in branch B_0. (Although TPC-B has been superseded by other TPC benchmarks, we use it here because it is small enough to implement quickly and "is designed to be a stress test on the core portion of a database system".) We found that the benchmark using the non-versioned data-service performed 3.8 times better than EVERETT. The most likely source of the performance problem is the need to fetch all SIDs for data items which match the query predicate. This causes the number of database queries per EVERETT query to rise linearly with the number data items which match the EVERETT query predicate, even if those data items are not visible in the branch of interest. Also, the number of SIDs fetched for each data item will typically grow over time, so that queries will take longer as the database contains more history.

At this point in the EVERETT project, we have demonstrated that the basic concept is feasible, and that our algorithms and data structures for maintaining branch isolation are correct. Although the performance is adequate for prototyping work, it is not adequate for production applications. Indexing is typically used to improve query performance in database systems. However, relational database indices typically only support range queries, and we do not see a way to implement EVERETT using only range queries. Emerging NoSQL [11] databases provide even less query functionality. We have begun work on new indexing algorithms which will provide a significant performance benefit for EVERETT.

References

[1] B. T. Bennett, B. Hahm, A. Leff, T. A. Mikalsen, K. Rasmus, J. T. Rayfield, and I. Rouvellou. A Distributed Object Oriented Framework to Offer Transactional Support for Long Running Business Processes. In *ACM Middleware*, pages 331–348, 2000.

[2] C. A. Curino, H. J. Moon, and C. Zaniolo. Graceful database schema evolution: the prism workbench. *Proc. VLDB Endow.*, 1(1):761–772, 2008.

[3] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Francisco, CA, USA, 1993.

[4] L. M. Haas, M. A. Hernández, H. Ho, L. Popa, and M. Roth. Clio grows up: from research prototype to industrial tool. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 805–810, New York, NY, USA, 2005. ACM.

[5] R. T. Jensen, C. S. Snodgrass. Temporal data management. *IEEE Transactions on Knowledge and Data Engineering*, 11(1):36–44, 1999.

[6] A. Leff and J. T. Rayfield. EDS: Data service middleware for situational applications. `http://domino.watson.ibm.com/library/cyberdig.nsf/Home`, RC24770, 2009.

[7] C. M. Pilato, B. Collins-Sussman, and B. W. Fitzpatrick. *Version Control with Subversion*. O'Reilly Media, 2nd edition, September 2008.

[8] Rails migrations. `http://guides.rubyonrails.org/migrations.html`, 2008.

[9] C. Shirky. Situated software. `http://www.shirky.com/writings/situated_software.html`, 2004.

[10] P. Stachour and D. Collier-Brown. You don't know jack about software maintenance. *Commun. ACM*, 52(11):54–58, 2009.

[11] M. Stonebraker and J. Hong. Saying good-bye to dbmss, designing effective interfaces. *Commun. ACM*, 52(9):12–13, 2009.

[12] TPC-B. `http://www.tpc.org/tpcb/default.asp`, 1990.

[13] J. Vesperman. *Essential CVS*. O'Reilly Media, 2nd edition, November 2006.

[14] Wikipedia. Hugh Everett III. `http://en.wikipedia.org/w/index.php?title=Hugh_Everett_III&oldid=333333490`, 2009.