Versions and Workspaces in Microsoft Repository

Thomas Bergstraesser Philip A. Bernstein Shankar Pal David Shutt

Microsoft Corporation

ABSTRACT

This paper describes the version and workspace features of Microsoft Repository, a layer that implements fine-grained objects and relationships on top of Microsoft SQL Server. It supports branching and merging of versions, delta storage, checkout-checkin, and single-version views for version-unaware applications.

1. INTRODUCTION

Microsoft Repository is an object-oriented layer implemented on Microsoft SQL Server. Version 1 of Microsoft Repository (V1) supported objects, relationships, and model extensibility, exposed through Microsoft's Component Object Model (COM). Version 2 (V2), described here, adds version and workspace support and ships in Microsoft SQL Server 7.0. Additional details are in [2,5].

Microsoft Repository is intended primarily for managing metadata consisting of fine-grained objects linked by arbitrary relationships. To be usable in a design environment and sharable by teams, objects must be versioned and managed using checkout-checkin. Since we cannot anticipate all usage scenarios of our diverse user community, the versioning model must be rich and flexible. To satisfy the needs of both shrink-wrapped and custom applications, it must also be easy-to-use, easy-to-customize, and efficient in time and space. Finally, since V2 extends a shipping product, interfaces must be backward compatible with V1.

While versioning techniques are well known, not many products or prototypes support all of the above requirements. Often, only coarse-grained objects can be versioned (e.g., files or configurations), only hierarchical relationships are allowed between versions, or all applications must be version-aware. We present our system as a case study that has none of these restrictions. Other interesting models are described in [3,4,7].

2. VERSION MODEL

The main goal of versioning is to enable the reconstruction of old states of an object. In contrast to V1, where a repository object has one state, in V2 a repository object can have many versions, each representing one of the object's historical states. Versions of an object are related by *successor* relationships, indicating the order in which the states arose. Each version has a globally unique *version id* and its own property and collection values. Collection values in our COM-based architecture represent relationships between objects or, in V2, versions of objects. Type definitions are ordinary repository objects, so in principle they could be versioned; however, this is not supported in the V2 release.

In V2, a version is *frozen* or *unfrozen*. The CreateObject operation creates the first version of a new repository object, initially unfrozen. As in V1, each object has single-valued scalar *properties* and *collections* of relationships [1]. Properties and collections of an unfrozen version are updatable, but those of a frozen one are not. The Freeze operation freezes a version.

Given a frozen version FV of object O, the Create-Version method returns a new version NV of O, where NV is a successor of FV. Requiring a version to be frozen before creating a successor allows the repository engine to use *delta storage*, that is, store only values of a version that differ from its predecessor. If a sequence of N versions has the same value of some properties, then we store one row, not N rows, and tag it by a version range.

The successor relationship between versions induces a directed acyclic graph as follows: Multiple invocations of CreateVersion on the same version, V, cause V to have multiple successors; after the first, each successor starts a new *branch*. Later, two versions of an object may be merged using the MergeVersion method. Methods are available to traverse the version graph by getting a version's successors and predecessors.

MergeVersion on an unfrozen version V of object O takes two parameters: a frozen version FV of O and a flag that identifies either V or FV as *primary*. It makes FV a predecessor of V and merges the state of FV into V as follows: It finds a least common ancestor of V and FV, called the *basis version*, BV, and compares V and FV to BV. For each property P of O, if only one of V or FV has updated P since BV, then the updated value is assigned to P in V. If both V and FV updated P, then the value of the primary is assigned to P in V. The same is true for collections, except that the rule is applied to the whole collection or to each relationship within the collection. A flag on each collection's type definition drives this choice. For example, if a collection has maximum cardinality 1, then merging the whole collection is more appropriate (e.g., consider merging two collections each with one member). One can override the semantics of this built-in merge algorithm in a wrapper, using COM aggregation [6].

3. VERSIONS AND RELATIONSHIPS

For power and flexibility, it is important to support relationships between individual versions of objects. For ease-of-use and backward compatibility with V1, single-version views are important too, which we describe in Section 5.

In V1, each relationship object connects a source object to a target object. In V2, each relationship object connects a source version to a set of target versions. More precisely, each relationship supports a method TargetVersions that returns a collection of all versions of the target object that are related to the source version. Version-to-version relationships are added and removed by adding and removing items in the TargetVersions collection. E.g., Fig. 1 shows a relationship from version 5 of X to versions 1 and

2 of Y. To add a relationship from version 5 of X to version 3 of Y, one adds version 3 of Y to the collection. Like properties, relationships use delta storage, tagging rows by version ranges.

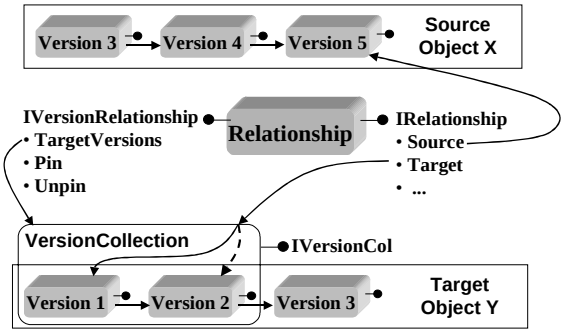

Figure 1 Versioned Relationship

4. WORKSPACE MODEL

To support team-oriented activities, a repository needs support for long-lived design transactions, to let users isolate their working versions from each other and make updated versions sharable later in a controlled way. V2 uses the time-tested approach to these requirements: checkout-checkin to and from private workspaces.

A workspace is a virtual repository that contains a subset of the objects and versions in the global repository. Each workspace is represented by an ordinary repository object, so workspaces can be grouped into collections and have custom relationships.

There is a collection of all the versions in a workspace. Standard collection operations are used to explicitly add/remove versions to/from a workspace, thereby making them visible/invisible in the workspace. A version can be added to many workspaces, but at most one version of an object is in each workspace. Thus, a workspace is a single-version view of a subset of the repository.

After a client opens a repository session, S, its context includes the entire repository. A client can access a workspace W in S. Workspaces support the session interfaces, so a client can use W as a logical repository session. By executing operations in the context of W instead of S, the client only sees objects in W, relationships on such objects, and those relationships' target objects in W. The client uses S instead of W to see the entire repository.

A workspace's support of session interfaces enhances backward compatibility. A V1 application accesses (non-versioned) objects using a repository session as its context. By replacing its session references by workspace references, the application can still use session interfaces on those workspace objects, so no other changes are needed. The resulting application only accesses objects that are in the workspace. A common usage scenario might be to populate a workspace and use it for a long period, during which many short transactions can be run.

A short transaction is associated with a session, not a workspace. So it can include operations on many workspaces, allowing complex models of sharing to be built with workspace primitives. For example, a transaction could checkin a version in the "test" workspace, freeze it, and then add it to the "released" workspace.

5. SINGLE-VERSION MODEL

Objects can be accessed without referring to specific versions, using the V1 API. V2 continues to support this non-version interface for backward compatibility, so an application written for V1 will continue to run on V2. This also allows tools to avoid the versioning model, if they don't need it.

V2 automatically resolves versions when accessed via non-version operations. For the object of interest, version resolution returns the version contained in the current workspace context. Or, if there is no workspace context, it returns the most recently created version of the object. E.g., executing get_Object(objID) in the context of a workspace W returns the version of that object currently in W, or null if no version of that object is in W. If accessed in the context of the whole repository, it returns the latest (most recently created) version of the object. Similarly, when traversing a relationship in the context of the repository, it gets either the pinned (i.e. default) version in the relationship, if there is one (the dashed arrow in Fig. 1), or the latest one, if not.

Notice that an application written for V1 is oblivious to the new versioning and workspace features of V2. Since it is written for V1, it creates one version of each object and only views the objects in the context of the whole repository, so version resolution is unnecessary. Even if a V2-aware application creates new versions and moves them in and out of workspaces, the V1 application only sees the latest or pinned version of every object.

6. CONCLUSION

At first, versioning seems like a straightforward addition to a database system. Our experience is the opposite. It is much harder than it looks to get simple and powerful versioning interfaces with good performance. Complexity arises from the combination of features: branching/merging, version-to-version relationships, and workspace scoping. These features have richer semantics than linear version sequences with containment-only relationships. Given this semantic richness, some extra performance cost seems unavoidable. There is an execution cost, mainly for version resolution and workspace scoping. We reduced these costs by careful interface design, so that version resolution semantics translates into acceptably efficient SQL calls, and by custom storage structures, such as an optimized table for workspace containment relationships. There is also storage cost, which we reduced by using delta storage.

7. ACKNOWLEDGMENTS

We're grateful to the entire Microsoft Repository team for their effort in turning these ideas into a product, especially Jason Carlson, Murat Ersan, Jayaram Mulupuru, and Ragini Narasimhan

8. REFERENCES

[1] Bernstein, P., B. Harry, P. Sanders, D. Shutt, J. Zander, "The Microsoft Repository," *Proc. 23rd VLDB*, 1997, pp. 3-12

[2] Bernstein, P.A., T. Bergstraesser, J. Carlson, Shankar Pal, P. Sanders, D. Shutt, "Microsoft Repository Version 2 and the Open Information Model," *Information Systems 24(2),1999.*

[3] Chou, H.-T. and W. Kim, "A Unifying Framework for Version Control in a CAD Environment," *Proc 12th VLDB*,1986, pp 336-344

[4] Katz, R.H. "Toward a Unified Framework for Version Modeling in Engineering Databases," *ACM Computing Surveys 22, 4* (Dec. '90).

[5] Microsoft Repository, http://www.microsoft.com/repository.

[6] Rogerson, D., *Inside COM*, Microsoft Press, 1997

[7] Sciore, E., "Versioning and Configuration Management in an Object-Oriented Data Model," *VLDB Journal 3,1994, pp. 77-106.*