XArch: Archiving Scientific and Reference Data

Heiko Müller Peter Buneman Ioannis Koltsidas
School of Informatics
University of Edinburgh, U.K.
{hmueller@inf, opb@inf, i.koltsidas@sms}.ed.ac.uk

ABSTRACT

Database archiving is important for the retrieval of old versions of a database and for temporal queries over the history of data. We demonstrate XArch, a management system for maintaining, populating, and querying archives of hierarchical data. XArch is based on a nested merge approach that efficiently stores multiple versions of hierarchical data in a compact archive. By merging elements into one data structure, any specific version is retrievable from the archive in a single pass over the data and efficient tracking of object history is possible. XArch implements this approach and extends it in two important ways. First, in order to merge large hierarchical data sets, elements need to be sorted according to their key values. We developed an efficient algorithm for sorting hierarchical data in secondary storage and modified the nested merge algorithm accordingly. Second, we designed and implemented a declarative query language that enables one both to view data from particular versions and to track the history of objects. We demonstrate this using both molecular biology and demographic reference data as examples.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications—*Scientific databases*; H.3.7 [**Information Storage and Retrieval**]: Digital Libraries

General Terms

Algorithms, Design, Management

1. INTRODUCTION

Scientific databases on the Web are a primary source of information in ongoing research efforts. However, the data is subject to continuous change and often only the most recent versions are preserved. For many database providers it has become common practice to overwrite existing database states when changes occur and regularly publish new releases of the data on the Web. Failure to archive earlier states of the data may lead to the loss of scientific evidence, and the basis of findings may no longer be verifiable. Recently, some database providers have initiated archiving efforts that allow to view entries exactly as they were in the past [9]. Database archiving is, however, not only important for verification of scientific findings. Consider the fact that nearly every dictionary, gazetteer, encyclopaedia or reference manual that one traditionally found on the reference shelves of libraries is now available on the Web. In many cases one is interested not just in an old version of the data, but one is interested in forming queries over the *history* of the data. A prime example of this is the CIA World Factbook [1], possibly the most widely used source of demographic information. Over the past 18 years the Factbook has moved from an annually printed distribution to a web resource. Over that period it has kept a remarkably uniform structure. Apart from one or two spreadsheets that contain only a small portion of the data, there appears to have been no attempt to bring all past versions into a common form. In fact, although all past versions are available on line (the printed versions have been transcribed into a variety human and machine-readable formats), it is quite hard even to find all these versions. Temporal queries such as "How did electricity generation in China change over the past 15 years?" can be extremely valuable, but to answer such queries using the data in its current state is extremely time-consuming. This emphasises the need to have both a uniform archival representation and a query language in order to make temporal queries like this easy to formulate and efficient to execute.

Scientific and reference data is predominantly kept in well-organised hierarchical data formats having a key structure that provides a canonical identification for each element of the hierarchy. Each element is uniquely identified by the path in which it occurs and the values of some of its sub-elements. In [6] a nested merge approach to archiving is developed that efficiently stores multiple versions of hierarchical data in a compact archive by "pushing down" time and introducing timestamps as an extra attribute of the data. Archives of multiple document versions are generated by merging the documents into a single document and annotating the elements with timestamps. Corresponding elements in different versions are identified based on their key values. The archiver stores each element only once in the merged hierarchy to reduce storage overhead. Archived elements are annotated with timestamps representing the sequence of version numbers in which the element appears. This approach has several advantages: (i) By merging el-

SIGMOD'08, June 9–12, 2008, Vancouver, BC, Canada.
ACM 978-1-60558-102-6/08/06.

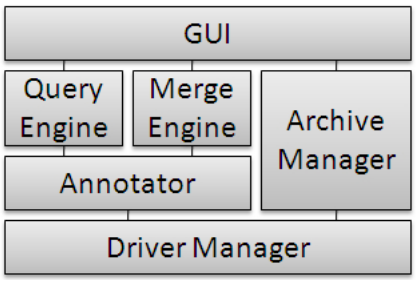

Figure 1: XArch System Architecture

ements into one data structure any specific version is retrievable from the archive in a single pass over the data, (ii) the storage space required is comparable to that of delta-based approaches that keep a sequence of records of changes between pairs of consecutive versions, (iii) tracking object history is easy, and (iv) the archive is stored in a human readable format, namely XML.

Based on the algorithms described in [6], we implemented the archive management system XArch. The system allows one to create new archives, to merge new versions of data into existing archives, and execute both snapshot and temporal queries using a declarative query language. We extend the initial approach in several ways. In order to eliminate the restriction on archive sizes due to main memory limitations, we developed an algorithm for efficient sorting and merging of hierarchical data in secondary storage. For querying archives, we implemented a query language that allows retrieval of old data versions and tracking of object history. We further integrated existing techniques for publishing XML to allow populating archives directly with data from relational databases.

2. SYSTEM OVERVIEW

We implemented XArch as a stand-alone Java application. The overall architecture is shown in Figure 1. We introduce the concept of IO driver to abstract from the data model of documents and data sources. A driver is basically a wrapper around any tool that generates XML output. Instead of writing the output to a file the driver calls the appropriate methods of a provided callback handler that transform elements, attributes and text into internal node objects. We currently provide several different drivers: an XML driver based on the *Simple API for XML (SAX)* [3], a driver for exporting data in a relational database based on XML publishing techniques as described in [4], and, as examples of drivers specific to a data format, drivers for Swiss-Prot flat files [2], and for the CIA World Factbook HTML pages on the Web [1]. Creation and deletion of archives is handled by the Archive Manager. The Annotator assigns key values to nodes based on relative keys [5] using the algorithm described in [6]. In the following, we briefly describe our algorithm for sorting hierarchical documents in secondary storage and our archive query language.

2.1 Sorting and Merging

The Merge Engine merges a new version of data into a given archive. When dealing with serialised hierarchical data formats, like XML, one usually employs a depth-first method to access the data. The problem with the nested merge approach described in [6] is that it does not manifest this natural access pattern. The reason is that in order to identify correspondences between children of merged nodes, one must read the complete subtrees of each such node. Thus, numerous passes over the data may be required to compute the nested merge outcome. If, however, the datasets are ordered according to their keys the situation is greatly alleviated. Assuming an ascending order, whenever two nodes are merged, one can follow a more traditional merge approach: sequentially scan the nodes' children and compare their values. The child with the smaller value is output (including the complete subtree rooted underneath it) to the new archive, after being annotated with the proper timestamp. This procedure ensures a total ordering among all children of any node. In case nodes with equal key values are encountered, we recursively merge their children. The described procedure ensures that the archive is always sorted on node keys. More importantly, only the incoming version has to be sorted in advance to performing nested merge.

The common approach for sorting large datasets in external memory is external merge sort. External merge sort splits the dataset into multiple sorted runs during a single pass over the data. Runs are then merged to generate the sorted output. Depending on the number of runs created in the first phase, merging may require multiple passes over the data. Splitting hierarchical data is, however, not straightforward. We developed an algorithm that "vertically" splits a hierarchical document into sorted runs [8]. We start by reading the portion of the document that fits into main memory and sort the data. Unkeyed nodes are pushed to the end of their parent node in document order. We output the tree as a sorted run. We keep all those node in memory whose children have not been read completely and read the next portion of the tree. By duplicating incomplete nodes we ensure that each run represents a proper tree rooted under the same original root node. Sorted runs are then merged to form the sorted document. Our experiments show that our algorithm is superior to existing bottom-up approaches, such as NeXSort [10], that collapse the tree into files of sorted subtrees. The reason is that the reconstruction phase of such algorithms accesses the subtree files in random order, which incurs an I/O penalty.

We use our sorting algorithm and modify the nested merge algorithm in [6] to overcome restrictions on archive sizes. After assigning key values to the nodes in the archive and the input document, we sort the document on node key values and merge the archive and the sorted document into a new archive version. We merge sorted runs directly with the archive whenever the number of runs allows to be merged in a single pass. For our experiments, we transform Swiss-Prot flat files available on the Web into XML format and create two example archives for Release 20 to 25 and 40 to 49. Splitting a file of 500 MB size (Release 41) into sorted runs currently takes 256 *sec* on an 1.8 GHz Intel Core 2 Duo processor machine using using 400 MB of main memory. Merging the sorted runs with an archive of comparable size (Release 40) requires another 193 *sec*. Inserting Release 49 (1.2 GB) into the final archive takes 562 *sec* for sorting and 661 *sec* for merging. Earlier versions of Swiss-Prot are sorted completely in main memory. Release 20 to 25 have size around 65 MB. Inserting the first document takes 44 *sec* and inserting the last document takes 70 *sec* (37 *sec* sorting and 33 *sec* merging).

2.2 Query Language

We implemented a declarative query language, XAQL, for querying archives. XAQL allows retrieval of particular data versions, tracking of object history, and retrieval of timestamps representing the sequence of versions when a given condition was valid. Although there exist a multitude of query languages for hierarchical and temporal databases, none of them is completely satisfactory for our needs or direct applicable to our data model. XAQL is oriented toward OQL [7] since archives are not arbitrary XML documents, but follow a fairly regular structure given by the key structure. XAQL considers keyed nodes as objects and an archive as a nested, timestamped object hierarchy defined by the given key structure. Besides keyed and unkeyed nodes timestamps are a first class concept in XAQL. The general syntax of a XAQL query is as follows:

```
SELECT [TIMESTAMP | select statement]
FROM archive
WITH variable definitions
VERSION timestamp
WHERE condition
```

The `SELECT` and `FROM` clause are mandatory, all other parts are optional. The `SELECT` clause specifies the nodes in the query result. The `FROM` clause specifies the archive to be queried since XArch manages multiple archives. The `WITH` clause allows definition of object variables that are used in select statements and where conditions. Temporal projection is enabled by the `VERSION` clause. When specified, a query is evaluated only on those versions in the archive that are listed in the given timestamp. The `WHERE` clause filters nodes based on a given condition. XAQL currently allows conjunctions of conditions on values of text nodes or unkeyed subtrees. Coincidence queries are supported by the keyword `COINCIDE` in the `WHERE` clause. We further support predicates on the history of nodes and subtrees. Predicate `HAS CHANGES` is true if the subtree rooted under a node has changes in the considered versions. The `WAS MODIFIED` predicate is true if an element itself was changed, i.e., it does not inherit its timestamp from the parent.

Example: Given an archive of a company database as outlined below:

```
<T t='1-5'>
  <COMPANY>
    <DEPARTMENT NAME='Marketing'>
      <EMPLOYEE SSN='111'>
        <NAME>John</NAME>
        <SALARY>
          <T t='1-2'>30,000</T>
          <T t='3-5'>40,000</T>
          ...
```

The following query lists the names of all employees whose salary changed over the first 3 versions:

```
SELECT E/NAME
FROM COMPANY
WITH /COMPANY/DEPARTMENT/EMPLOYEE E
VERSION 1-3
WHERE E/SALARY HAS CHANGES ■
```

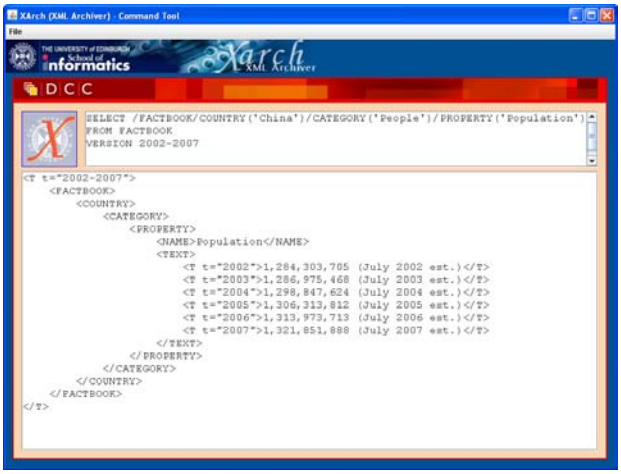

Figure 2: XArch Command Line Interface

Path expressions in XAQL may contain constraints on key values. For example, `/COMPANY/DEPARTMENT('Marketing')` matches the marketing department (assuming departments are keyed by their name). Queries are evaluated in a single scan over the data. We start by determining the longest common prefix (LCP) for all path expressions in the `SELECT` and `WHERE` clause. For nodes that match the LCP the `WHERE` clause is evaluated. If the condition evaluates to true, we output those parts of the subtree that are specified in the `SELECT` clause. The results are merged into a single document. Similar to OQL we allow subselects in the `SELECT` clause. Specifying `TIMESTAMP` instead of a select statement in the `SELECT` clause returns the union of timestamps for all nodes that satisfy the `WHERE` clause.

3. DEMONSTRATION

We demonstrate the full functionality of XArch and its query language XAQL using different data sources. We implemented a graphical user interface that allows to create and delete archives, merge new document versions into an existing archive, and view or export complete archive snapshots. More complex XAQL queries are supported using the graphical command line tool of XArch shown in Figure 2. To demonstrate the ability for handling large data sets, we archive several major releases of the Swiss-Prot Protein Knowledgebase. Using XAQL, we can now answer questions like "Are deleted entries re-entered in later releases?" or "Which entries remained stable over past releases?". Using a simple company database in relational data format, we demonstrate the ability to populate archives directly from relational databases. Finally, we present archives of the CIA World Factbook. The Factbook is currently updated every two weeks. However, only the current version and the annual releases are available on the Web, while all other versions get overwritten. We create an archive of annual releases and an archive containing the most recent changes. Using these archives, we are able to provide answers to questions like "Which countries have recently been added to the Factbook?", "How did the population of European countries change over the last years?", "When was Tony Blair Prime Minister of the U.K. and how did the Gross Domestic Product change during that time?".

4. REFERENCES

[1] https://www.cia.gov/library/publications/the-world-factbook/index.html.

[2] http://www.ebi.ac.uk/swissprot/.

[3] http://www.saxproject.org/.

[4] M. Benedikt, C. Y. Chan, W. Fan, R. Rastogi, S. Zheng, and A. Zhou. Dtd-directed publishing with attribute translation grammars. In *Proc. 28th Int. Conf. on Very Large Data Bases (VLDB)*, pages 838–849, 2002.

[5] P. Buneman, S. Davidson, W. Fan, C. Hara, and W.-C. Tan. Keys for xml. In *Proc. 10th Int. Conf. on World Wide Web (WWW)*, pages 201–210, 2001.

[6] P. Buneman, S. Khanna, K. Tajima, and W.-C. Tan. Archiving scientific data. *ACM Trans. Database Syst.*, 29(1):2–42, 2004.

[7] R. G. G. Cattell, D. K. Barry, D. Bartels, M. Berler, J. Eastman, S. Gamerman, D. Jordan, A. Springer, H. Strickland, and D. Wade. *The object database standard: ODMG 2.0*. Morgan Kaufmann, 1997.

[8] I. Koltsidas, H. Müller, and S. Viglas. Sorting of hierarchical data in external memory. Technical Report EDI-INF-RR-1217, University of Edinburgh, 2007.

[9] R. Leinonen, F. Nardone, O. Oyewole, N. Redaschi, and P. Stoehr. The embl sequence version archive. *Bioinformatics*, 19(14):1861–2, 2003.

[10] A. Silberstein and J. Yang. Nexsort: Sorting xml in external memory. In *Proc. 20th Int. Conf. on Data Engineering (ICDE)*, page 695, 2004.