Xˆ 3: A Cube Operator for XML OLAP

Nuwee Wiwatwattana
U of Michigan
nuwee@eecs.umich.edu

H. V. Jagadish*
U of Michigan
jag@eecs.umich.edu

Laks V. S. Lakshmanan
U of British Columbia
laks@cs.ubc.ca

Divesh Srivastava
AT&T Labs–Research
divesh@research.att.com

Abstract

With increasing amounts of data being exchanged and even generated or stored in XML, a natural question is how to perform OLAP on XML data, which can be structurally heterogeneous (e.g., parse trees) and/or marked-up text documents. A core operator for OLAP is the data cube. While the relational cube can be extended in a straightforward way to XML, we argue such an extension would not address the specific issues posed by XML. While in a relational warehouse, facts are flat records and dimensions may have hierarchies, in an XML warehouse, both facts and dimensions may be hierarchical. Second, XML is flexible: (a) an element may have missing or repeated subelements; (b) different instances of the same element type may have different structure. We identify the challenges introduced by these features of XML for cube definition and computation. We propose a definition for cube adapted for XML data warehouse, including a suitably generalized specification mechanism. We define a cube lattice over the aggregates so defined. We then identify properties of this cube lattice that can be leveraged to allow optimized computation of the cube. Finally, we present the results of an extensive performance evaluation experiment gauging the behavior of alternative algorithms for cube computation.

1 Motivation

Online analytical processing (OLAP) is one of the most important applications in the relational database world. As XML becomes widely adopted, its modeling flexibility makes it a natural for representing data that would be less convenient to represent in (normalized) relational form. A natural consequence of this is that XML data finds its way into data warehouses, leading to the need for OLAP on XML data.

Consider, for example, a business such as Amazon, storing data about the books that it sells, and their sales. Figure 1 shows a very small fragment of the sort of data one may expect to see in such a warehouse. Note that the flexibility of XML permits the second publication to have two different values for year and the third publication to have no publisher. This sort of heterogeneity is common in XML, and is to be expected not just in the context of books, but also in other contexts, such as warehouses of information based on electronic catalogs, or records of insurance claims.

The central operation in OLAP is the computation of aggregates by group, for a variety of groups of interest. These computations are frequently combined into the computation of a *data cube*, based upon a multidimensional organization of data. It is usually constructed by aggregating at different granularities, groupings of data attributes viewed as dimensions. There has been considerable research on efficient computation of (relational) data cubes [1, 5, 8, 19, 24].

Relational data cubes have certain properties, which are assumed in cube specification and cube computation algorithms. (1) Data is *summarizable*, in that a coarser aggregate can be computed solely from corresponding finer aggregates (e.g., a group-by on product is determined by group-by on product, location), without access to base data. (This issue arises regardless of the kind of aggregate functions employed, which may even be distributive (like sum) or algebraic (like avg)). (2) The number of groups for a group-by is determined solely by the number of data values (e.g., the number of product groups is determined by the number of distinct products).

However, XML, which has a heterogeneous tree structure, is flexible and thus does not always comply with the above assumptions. This raises both a *computational challenge* and a *semantic challenge* for (the definition of) an XML data cube. It is these challenges that this paper sets out to address. We begin by elaborating on these challenges.

Computational Challenge: The tree structure of XML allows for diversity in data representation which leads to the invalidation of summarizability properties [15]. When summarizability does not hold across data to be aggregated, traditional cube algorithms and optimizations, which rely on this property, will not produce the correct results. Specifically, the coarser level aggregates cannot be computed from aggregating the finer level of aggregation results (roll-up computation). Figure 1 shows a tree representation of a publication database in XML. Element types and attribute types are nodes. Text type is quoted under an element node. An edge is a relationship between two element nodes, a parent and a child. In this example, let us assume that we want to count the number of publications

*Supported in part by NSF under Grants No. IIS-0219513 and IIS-0438909.

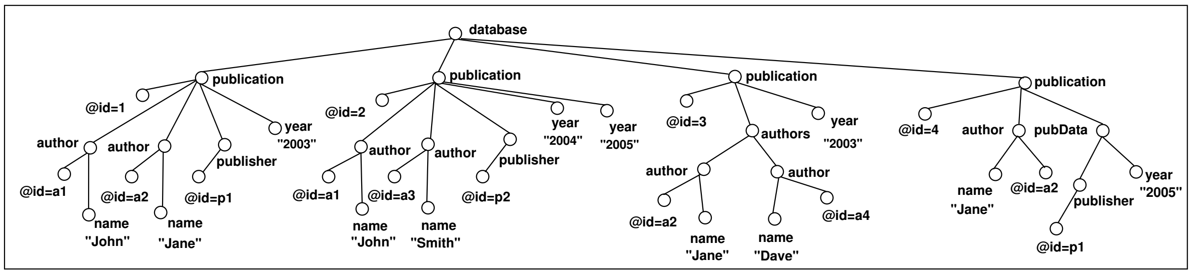

Figure 1. Publication Database in XML.

grouped at all granularities (i.e., cube by) of author's name, publisher and year (assume that the `@id` attribute is a unique identifier and used as the identifier for publications, authors, and publishers). Summarizability is invalidated here in two situations.

Firstly, the non-summarizability stems from a missing association, allowed by an optional sub-element in the underlying schema. In Figure 1, the third publication does not have a publisher child (it is an online article); therefore, the group-by year, publisher will not contain the third publication. Clearly, if we employ the result of this finer group-by to determine yearly count, i.e., the coarser group-by year (the roll-up), we will miss the count of the third publication because the third publication is not counted in the finer group-by year, publisher. Lenz et. al. [15] called this *incompleteness of dimension attribute(s) [coverage] relative to the aggregate measure* (in this example, the attribute is publisher and the measure is publication count). This situation usually does not happen in the relational database world, because database users often cope with it at the data level by patching the missing values by a synthetic value like *'other'*. In XML, the situation needs attention at the evaluation and computation level because optional subelements are normal.

Secondly, the non-summarizability stems from a repeatable sub-element. The first publication is a member of both the groups (John, p1, 2003) and (Jane, p1, 2003). Then, the group (p1, 2003) contains only the first publication and its count should be one. However, the roll-up from the finer level groups mentioned each count as one; added up, the result is two, which is wrong. The roll-up of the group-by publisher, year to publisher of the second publication exhibits the same phenomenon because this publication has 2 editions. Lenz et. al. [15] called this a *non-disjointness of grouped partition relative to dimension (or group-by) attribute(s)* (in this example, the attribute is author; and thus author's name). Again, in the relational database world, the OLAP data is usually modelled as a star or snowflake schema. Each fact tuple is considered as a unique data item that is described by a fixed group of attributes and different instances of a given group-by are disjoint.

Semantic Challenge: In addition to violation of summarizability, in XML, groups are determined not only by values, being grouped at different granularities, but also by the heterogeneity of tree structures. Consequently, specification of a group-by and indeed of a cube has to take this into account. We propose specifying the cube using a tree pattern query. Figure 3(a) shows an example query. Obviously, the third publication sub-tree in Figure 1 does not exactly match the query tree (because of the presence of the sub-element authors), and thus, not in the result set according to the evaluation perspective alone. However, the group-by author's name, publisher and year as specified by the user should semantically include the third publication. The solution is to *relax* our query tree to also include the descendant author. The purpose here is to increase the range of data items at which we are looking. We want to be able to catch the flexibility of the tree structure since a typical data analyzer might prefer for relevant data items to appear in his or her chart. This type of tree relaxation is called a parent-child to ancestor-descendant generalization, because the author could not be just a child sub-element but could be a descendant. As another example, the fourth publication subtree does not match the query tree as well. One way of including it in the range of the cube is to "promote" both publisher and year sub-elements of pubData and delete the pubData element which is now a leaf. These and other similar tree relaxations will be discussed in section 2.2.

The main contributions of this paper are as follows:

- We propose a definition of an OLAP cube operator for XML data that accounts for the great variability in the tree structure of XML. (Sec. 2).
- We present algorithms for computing the cube efficiently. Our algorithms allow for the following two kinds of violation of summarizability: non-disjointness of grouped partition and incomplete coverage of dimension attributes. (Sec. 3).
- We present a detailed empirical evaluation of our algorithms on a variety of real data sets. (Sec. 4).

2 Query Model

2.1 Grouping Specification

XML data has a tree-structured model, and queries against XML are often described in terms of tree patterns.

In TAX [12] and several follow on works, grouping is specified by means of a grouping tree pattern. This pattern is matched against the XML database. Whenever a match is found, it is called a *witness tree*. TAX even has a structure-based grouping operator.

Inspired by this, we will specify grouping in XML by means of a tree pattern and a grouping list. The tree pattern is used to create a set of witness trees. An equality check is performed on corresponding nodes belonging to the grouping list in each witness tree, and all witness trees where these values match are placed into one group. The witness trees grouped in the returned result sets need not have the same pattern as the elements used to perform the grouping.

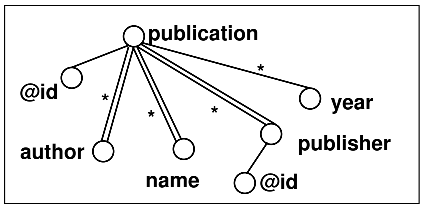

Figure 2. The most relaxed fully instantiated XML tree pattern providing results that cover the entire lattice for Query 1. * designates left outer join.

For example, a simple tree pattern seeking a year node as child of a publication node will match the first three publications in Figure 1, and actually match the second publication twice. If we now specify the value of year as the grouping basis (by naming the year element in the grouping list), then we get three groups. The first, for year 2003, has the first and third publications in it. The two other groups, for years 2004 and 2005, respectively, each have the second publication. The fourth publication did not match the specified tree pattern (since year is not a direct child of publication) and hence is not included in any of the groups.

2.2 Tree Pattern Relaxation

Given the heterogeneous nature of XML data, it may some times be difficult to specify a single grouping tree pattern that will correctly match all the items that the user wishes to group. At other times, the user may only have a rough idea of the schema. Perhaps the slightly different structure of the fourth publication in the example above is something that the user wishes to ignore. To address similar issues for ordinary (selection) queries, the idea of tree pattern relaxation has been proposed [2].

We consider three forms of grouping tree pattern relaxation here:

1. Parent-Child to Ancestor-Descendant Edge Generalization (PC-AD): is a relaxation of structural relationship between two elements from parent-child type to ancestor-descendant type. For example, a pattern publication/author will not match the third publication in Fig. 1 due to the intervening authors element. However, the relaxed pattern publication//author will match all four publications.

2. Sub-tree Promotion (SP): as described in [2], SP moves a subtree rooted at a node n to be a child of the grandparent of n, using the desc. axis. For example, publication[./author/name] can be relaxed to the tree pattern query publication[./author][.//name].

3. Leaf Node Deletion (LND): permits the tree pattern to be considered matched even in the absence of a specified leaf element. For example, a tree pattern that includes publisher cannot be matched by the third publication in Fig. 1. But if this publisher node is made optional, then it could match a tree pattern.

Leaf node deletion alone can be used to reflect the traditional cubing. To see cubing in operation, consider a tree pattern with branching, e.g. publication as root with author and publisher as children. See Fig. 3(j). This pattern groups publications by author and publisher. Eliminating the author leaf node results in a grouping of publications by publisher alone; eliminating the publisher leaf node results in a group of publications by author alone. Eliminating both leaf nodes gives us a group by nothing, placing all publications into a single group.

We have seen that a single tree pattern relaxation, leaf node deletion, can represent traditional cubing (roll-up and drill-down). In addition, there are two more structural relaxations described above for which there is no relational counterpart. These represent an additional degree of complexity in semantics and in computation.

2.3 The Definition of X^ 3

It turns out that not all relaxations are suitable for every query tree pattern. For instance, if the *publisher* element has a child *name*, the sub-tree promotion of author's name will not return the name of the author, but also the name of the publisher, which may not be what the user wants.

Furthermore, it could be the case that the user is not interested in the full cube, but only in portions of it. In the traditional cubing specification, such user requirements are expressed by explicitly stating which dimensions to cube, which to roll up, and which to leave alone.

We follow the same idea and augment the XQuery FLWOR expression to include an `X^3` clause that explicitly identifies the permitted relaxations with each relevant variable in the grouping tree pattern, and ensure that the RETURN clause identifies the aggregate function that needs to be computed. This is illustrated in the following example Query 1.

```
for $b in doc("book.xml")//publication,
    $n in $b/author/name,
    $p in $b//publisher/@id,
    $y in $b/year
X^3 $b/@id by $n (LND, SP, PC-AD),
          $p (LND, PC-AD),
          $y (LND)
return COUNT($b).
```

This example is a formal specification from our running example which simply says that we want a count of the publication grouped by all relevant tree relaxation patterns contributed by the author's name, the publisher's id and the year.

The set of results to be produced comprise a lattice. Each node in this lattice represents a cuboid, which is the set of aggregates obtained for a particular grouping of the base data. There is a global top of the lattice, comprising the finest level of aggregation that is of interest, and a global bottom comprising a coarsest level aggregation where all entities are combined into a single group. An edge from a node in the lattice to a node at a lower level in the lattice represents a relaxation. (In the case of traditional cubing, this relaxation is just the removal of a grouping dimension. In the case of XML, it could be any of the relaxations discussed above.) Fig. 3 shows the complete cube lattice for our running example Query 1.

3 Cube Computation

3.1 Background

Cube computation for relational data has been studied extensively in the literature. The simplest technique is to have a counter for each group. As each tuple is scanned, all relevant counters are incremented. If the number of counters is small enough to fit in memory, this is an efficient, single-pass algorithm. However, in most real cases, the number of counters may far exceed the amount of available memory. In this case, this simple *counter* algorithm will thrash horribly as counters are paged in for each tuple in the scan.

Better techniques have been devised to exploit the fact that aggregates to be computed are not all independent *Bottom Up* algorithms successively refine the data set – beginning with a very coarse partitioning. At each stage, we rely upon the fact that partitioning is hierarchical, that every group in a finer partition is completely included in a corresponding coarser partition. *Top Down* algorithms begin by computing aggregates at the finest level first. Coarser aggregates are obtained by summing up the relevant aggregates one level up. The base data is not touched except for the first time when fine aggregates are computed.

3.2 X^ 3 Lattice Properties

Relational cubing algorithms assume *summarizability* properties in the data. These properties can specifically be written as:

- Pairwise disjointness between instances in the cuboid. A data element should occur in only one group in a cuboid. E.g. a book with two authors, if grouped by author, would be placed in two groups, and hence would violate disjointness.

- Total coverage of a more relaxed cuboid by the union of its adjacent less relaxed cuboids. If a book has no author then it may not participate in any group when grouped by author, and hence would violate coverage.

Each of these properties may independently hold (or not) at every point in the lattice for an XML data warehouse. We consider below, in turn, the impact of this property relaxation on each of the three families of cube computation algorithms.

3.3 Counter-based Algorithm

Since each counter is incremented independently, counter-based algorithms do not depend on the summarizability properties. However, there still is a little bit of work to do on account of there not being a simple notion of a tuple in XML. When the sub-tree corresponding to one countable element is being examined, it may turn out that it has several values for some grouping attributes. This results in a combinatorial number of counters being incremented for a single sub-tree. E.g. a book may have multiple authors and multiple editions. We then have to increment counters for each combination of author and edition, among others.

3.4 Bottom-Up-based Algorithm

Bottom-up algorithms begin with the most relaxed cuboid, and then gradually compute restrictions of it, through recursive partitioning. In the absence of the disjointness property, the restrictions may result not in partitions but in overlapping sets. This requires that at each stage in the recursive refinement, we consider all elements in the child cuboid for each parent cuboid restriction, including those that have already satisfied the restrictions for some other children cuboids.

But there are more fundamental questions to address in the context of XML – what are the "elements" that should be in the sets corresponding to each cuboid? If some subtree nodes are not included then the information for further partitioning may not be available, and may require that the database be accessed again to determine this information, completely defeating the whole point of a bottom up cube computation.

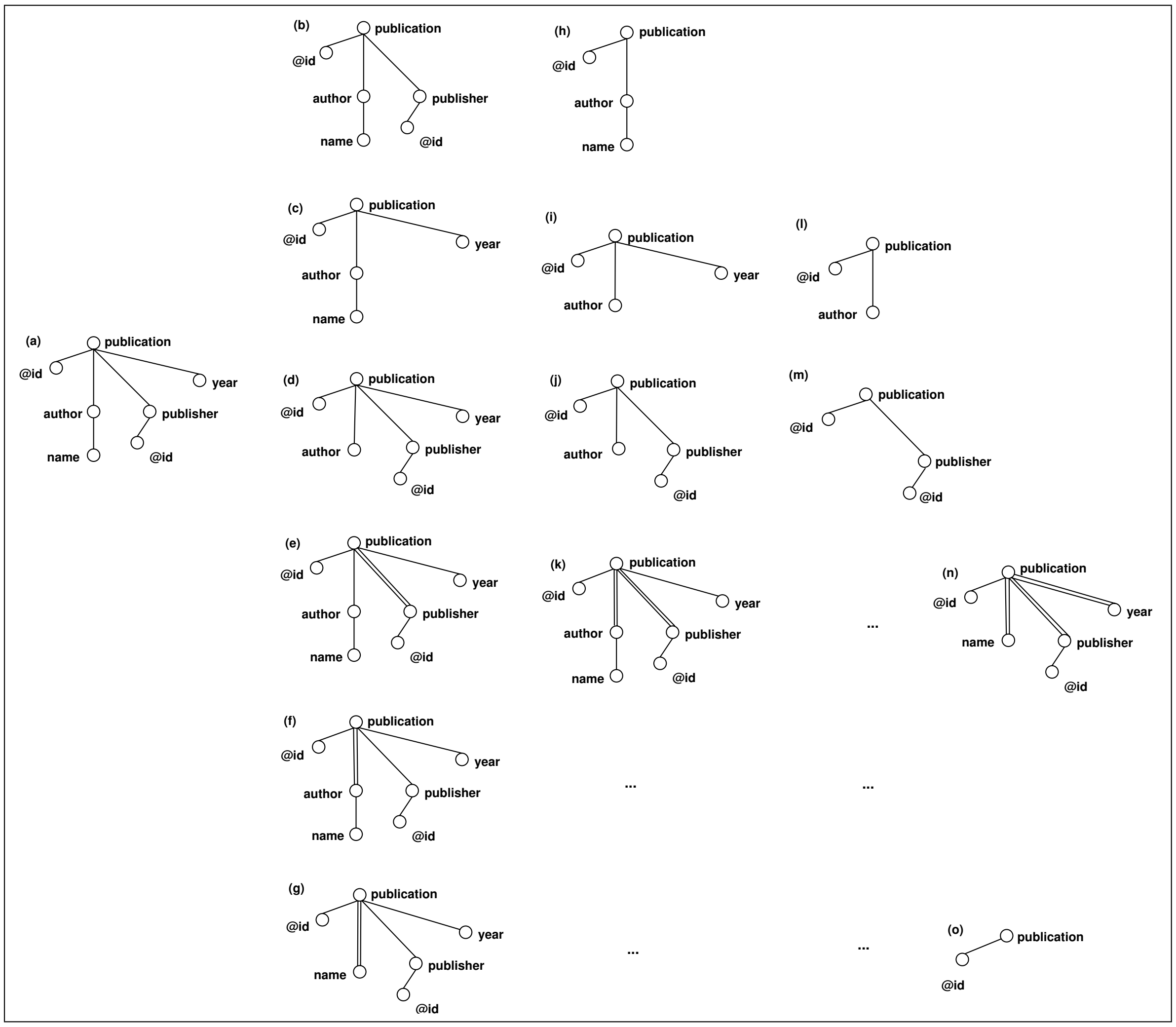

Figure 3. An excerpt from the XML relaxed-cube lattice of XML cube query 1. Each sub-lattice is an XML query tree pattern. (a) is the rigid tree pattern from the query specification. (b) - (g) are one-step relaxed from the rigid pattern. (h) - (k) are two-step relaxed from the rigid pattern. (o) is the most relaxed tree pattern. The parent is the less relaxed lattice point. This picture is actually a 90 degree rotation of the lattice.

We note that groups corresponding to every lattice element are defined by means of tree patterns. A typical way to evaluate a tree pattern is to consider one edge at a time, and evaluate the corresponding structural join. If we start from the root node, and instantiate the tree pattern down, one edge at a time, this is precisely what a bottom-up algorithm requires, except that the intermediate result of the partial match is also used for grouping and aggregate computation.

The only remaining catch is that this works only for the LND relaxation, which is the only one applicable to the relational cube. We would also like to include the other types of relaxations in this process. Towards this end, we define a *most relaxed fully instantiated* tree pattern, in which all specified non-LND relaxations have been applied. The most relaxed tree fully instantiated tree pattern for Query 1 is shown in Figure 2. Once we have found the set of matches to this pattern, all other matches are subsets of it. Therefore, the standard bottom up refinement process can be used to compute restrictions of this set.

The above ideas can be applied to make suitable modifications to any bottom up cube computation algorithm. See [23] to see how this applies to the algorithm in [5].

3.5 Top-down-based algorithm

Top-down algorithms compute the cube from a finer level to a coarser level of aggregation. To be able to compute the coarser level aggregate by merely adding up the finer level aggregates, we require that there be no overlap between finer level aggregates (disjointness) and that the coverage property hold. Even if the coverage property does not strictly hold, it can usually be made to hold through the introduction of a “null value” group in the less relaxed cuboid. E.g. if there is a grouping of books with no authors, then such books will be accounted for properly when we compute the total number of books.

Disjointness is a more serious requirement. If disjointness does not hold, then it is no longer enough just to add up the counts of the less relaxed groups. We need to keep track of the identities of the elements in these groups to make sure that we do not double-count. But that completely defeats the point of a top-down algorithm – we have to touch as many elements as in the base data, rather than as many as there are aggregates.

As in the case of bottom up, even in this case, the presence of various relaxations means that starting with the most relaxed fully instantiated grouping tree pattern is of use. The above ideas can be applied to make suitable modifications to any top down cube computation algorithm. See [23] to see how this applies to the algorithm in [19].

3.6 Discussion

The above algorithm descriptions assumed that no intermediate result is materialized. In many cases, we may be better off to materialize some intermediate cube results. The incompleteness of coverage directly affects the computation from these intermediate results. The solution is to accompany intermediate results that we will need at a later time with the attributes to be aggregated (keeping track of fact items), just as we had to for top down computation.

We saw above that there is a significant impact on computational algorithm, and hence cost, due to the absence of summarizability properties. We note, that even though we cannot assume these properties in general, there certainly are many instances in XML data where these properties hold. It turns out that the algorithms above can be customized, at each point of recursive refinement (or recursive aggregation), to make local use of summarizability properties, even if these are not applicable globally. This motivates the question how can we know when these properties will hold.

3.7 Inferring Lattice Properties From The Schema

In many cases, XML data comes with a schema (DTD or XML Schema). The lattice properties are thus inferrable from the knowledge of schema that is available. For instance, see Query 1 in Fig. 3 (a). If the schema of this XML data says that author element is possibly repeated, every lattice point that includes author element does not have the disjointness property because the repeated author elements result in different witness trees. If the schema says that the publisher element can be missing, total coverage property will not hold between (parent and child) lattice points that include publisher , e.g. //publication/publisher[./author/name][.year] $\rightarrow$ //publication/publisher[./author/name] (applying LND to year). As another instance, if the schema says that every path from publication to name goes through author, then lattice points - //publication/author/name and //publication//name - (which is obtained after applying SP name and LND author) have the same coverage.

4 Experimental Evaluation

Rather than implementing cube algorithms in a standalone mode, we implemented the three cube algorithms discussed in this paper in C++ on the TIMBER native XML database [11]. The XML data file was loaded into TIMBER, and evaluated using the available structural join algorithms. In this section, we report results from the COUNT operation only (other distributive and algebraic operators are expected to produce similar results). TIMBER was run on a single processor PentiumM 1.6GHz equipped with 1Gbyte of memory, 60GBytes of disk storage and Windows XP Pro operating system. The buffer pool size was set to 512Mbytes and data page size was configured to 8Kbtyes.

We experimented with two very different datasets. The first dataset that we used is the TreeBank dataset from UW Repository [21]. TreeBank has an average depth of 8, and the maximum depth is 36. When loaded in TIMBER, the data size is 576 Mbytes including necessary indices, with two and a half million elements. The second dataset is the DBLP XML records [6] that has 6 million elements, and maximum depth of 7. The DBLP data size is 1GByte including indices when loaded. Treebank is a highly heterogeneous recursive dataset and thus can be used to create different types of cubing load scenarios based on the specification. DBLP, on the other hand, is much more regular. It is not deep at all and has a fixed number of elements that are missing or repeated as suggested by its DTD.

To measure the time of the cube operator accurately, we pre-evaluated the query tree pattern, and materialized the results into a file. The file was then read in and the cubing was performed. The results were written into files. All data partitioning and sorting used the quicksort for an in-memory sort, and the mergesort for an external sort. All running times were measured with a cold cache and include both the I/O time and the CPU time.

The counter-based algorithm is denoted as *COUNTER*. The bottom-up algorithms implemented are the non-collapsing XMLized versions of [5]. There are two versions of bottom-up algorithms: *BUC* is the XML-aware relaxed tree pattern version in which overlapping data is taken care of, and *BUCOPT* is the optimized version of BUC which will exploit the fact that there is no overlap between children of a lattice node (disjointness holds). Recall that we do not have an optimized version when total coverage holds, because the bottom-up computation is not affected by this property. The top-down algorithms are XMLized versions of the PartitionCube/MemoryCube algorithm in [19]. *TD* is the standard (unoptimized) version. *TDOPT* is the optimized version that saves work by exploiting disjointness. *TDOPTALL* is the optimized version when both disjointness and total coverage hold globally. TDOPTALL, in particular, makes use of the computation from finer level. Note that it is not possible to get an optimized version of TD when disjointness does not hold but total coverage holds because the identifier of the data must be retained (to eliminate duplicates), and that would require as much work as going through all input trees.

Given the heterogeneity of Treebank dataset, we configured each experiment by controlling the behavior of the matching input trees according to two properties of summarizability: total coverage and disjointness. Treebank is an encrypted text data from WallStreet journal, so we grouped a marked-up element by the value of the marked-up text under it, and by the pattern of the marked-up elements.

Our key interest is in the impact of the two summarizability properties. We study three cases. First, we study the performance when disjointness property holds, but total coverage does not hold. Second, we study the case when both summarizability properties hold. Last, we consider the case when neither summarizability properties holds. We do not report results for the fourth case, where total coverage holds but disjointness does not, since no algorithm is able to benefit from the total coverage property in this case, leading to performance as if neither summarizability property holds.

For each of these three cases, we generate data sets that lead to dense cubes (lots of data points aggregated in each data cube cell) and those that lead to sparse cubes (many cube cells have no data points at all)[1].

For each of these six settings (three times two), we report results for all five algorithms. Along the X-axis for each graph, we plot the number of axes used. (An axis represents one path binding in the cube specification or one dimension in relational cube, e.g. $n in $b/author/name. All axes had the same input cardinalities). Running time in seconds is plotted on the y-axis.

4.1 Total Coverage does not Hold, Disjointness Holds

We controlled the input to have disjointness of grouping data by writing queries against the Treebank dataset that filter out duplicates within each group data so that BUCOPT and TDOPT can correctly be applied at every node. The cube result size ranged from 135 thousand to 19 million cells. Fig.5 shows the result of computing cubes using different algorithms from 10^5 input trees. Fig.6 shows the performance in computing cubes that are relatively denser

[1]The cube is sparse when the product of the cardinalities for group-by(s) is large relative to the number of tuples that actually appear in the result. Otherwise, it is called dense. Typically the cube gets more sparse when the number of dimensions increases. [5].

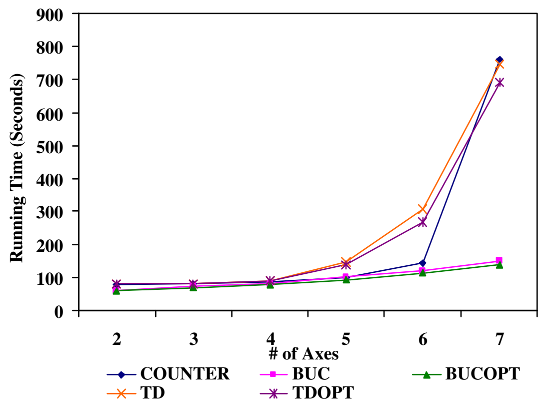

Figure 4. Performance in seconds for computing sparse cubes from Treebank 10^4 input trees with total coverage does not hold, disjointness holds.

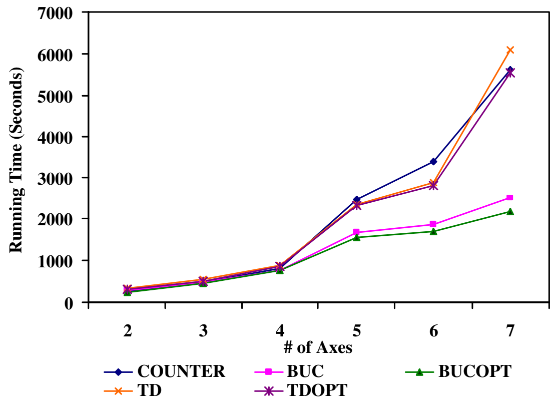

Figure 5. Performance in seconds for computing Sparse Cube, 10^5 input trees with total coverage does not hold, disjointness holds.

than Fig.5. We made the cube dense by grouping only the first character of the marked-up text and choosing the elements that has lesser value range. Note that TD, TDOPT and COUNTER did not finish in a reasonable time (10,000 secs) for 7 axes in Fig. 6. This is why the corresponding curves stop at 6 axes in the figure.

The counter-based algorithm is as fast as, or faster than other algorithms when the cube is small and so fit in memory. The bottom-up algorithm is well-known for its ability to perform well in computing sparse cubes as we see here. However, the top-down algorithm cannot outshine in computing dense cubes because elements in the XML input were possibly missing. Without the benefit to compute finer level aggregates from coarser level aggregates that are smaller than base data, TD (as well as TDOPT) is even worse than the counter-based algorithm due to the exponential number of (external) sorts required.

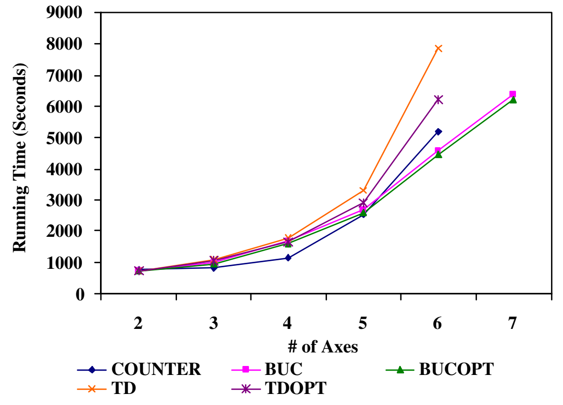

Figure 6. Performance in seconds for computing dense cubes from Treebank 10^5 input trees with total coverage does not hold, disjointness holds.

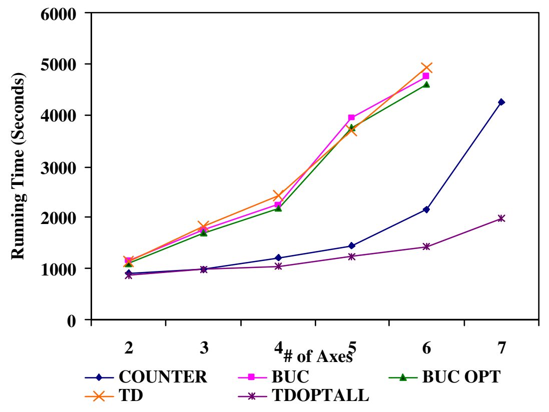

Figure 8. Performance in seconds for computing dense cubes from Treebank 10^5 input trees with total coverage and disjointness hold

4.2 Total Coverage and Disjointness Hold

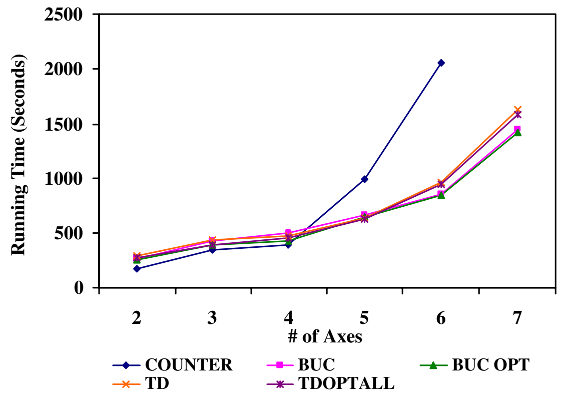

Figure 7. Performance in seconds for computing sparse cubes from Treebank 10^5 input trees with total coverage and disjointness hold

Fig. 7 and Fig. 8 show time to compute sparse and dense cubes at 10^5 number of matching input trees. The performance is similar to that shown previously for relational cubes, i.e. the bottom-up algorithms are good for sparse cubes, and the top-down algorithms are good for the dense cubes. Since both summarizability properties hold, we were able to run the TDOPTALL algorithm rather than merely TDOPT.

Since the degree of relaxation in this setting is one step less than the first setting, the average cube size is smaller, and the computation is faster. The cube result size ranges from 8 thousand to 6 million, smaller numbers than the first setting as they have lesser degree of relaxation. Even so, some of the algorithms failed to finish for the 7 axes case.

4.3 Total Coverage and Disjointness do not Hold

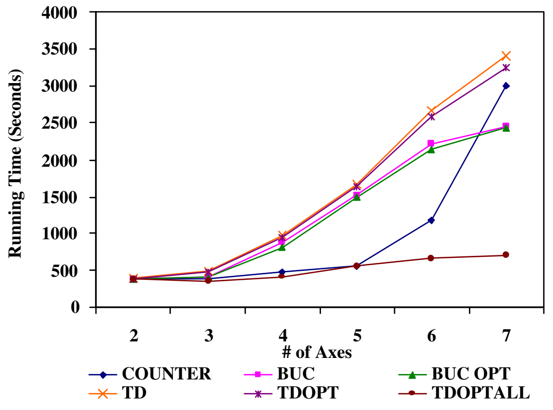

Figure 9. Performance in seconds for computing dense cubes from 10^5 Treebank input trees with total coverage and disjointness do not hold.

If summarizability does not hold, BUC and TD (unoptimized) are the only choices. The results for sparse data are very close to those shown in Fig. 5, and we omit these.

However, for dense data we see some differences from the corresponding Fig. 6, and thus show the new results explicitly in Fig. 9.

Even though optimized versions of the algorithms compute incorrect results, due to the summarizability properties not holding, we still ran them, just to see what the running time would be. BUCOPT and TDOPT did not provide much performance benefit, in spite of computing wrong results! But TDOPTALL did very well indeed, exploiting the dense

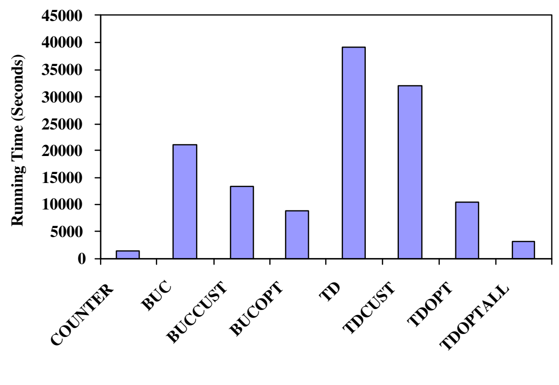

Figure 10. Performance of computing a cube /article by /author, /month, /year, and /journal on the DBLP dataset.

cube well. Of course, the results it computed were not correct, so the performance is moot. COUNTER did comparable to TDOPTALL, at least for low dimensions, but suffered from the usual exponential meltdown as the dimensionality grew higher.

4.4 Scaling Experiment

Fig. 4 and Fig. 5 show our scalability experiment of a sparse cube for input trees size of 10^4 and 10^5, respectively, when total coverage does not hold but disjointness holds. Reports on scalability test from other settings are omitted as they exhibit similar trends. (Actually, this is a scale down experiment, since we reported the full scale numbers for all the other cases already).

It is no surprise that larger data sizes require proportionately longer running time. What is more interesting is that the benefits of the optimized versions of the algorithms is greater at larger scale. Also, the counter-based algorithm runs out of memory and begins thrashing for a smaller number of axes (on which the cube size is exponentially dependent) when the input data set size increases.

4.5 DBLP Experiment

The DBLP dataset has better defined structure, and richer semantics, than the Treebank dataset. We used this dataset for an experiment involving customized optimization. Rather than globally requiring summarizability properties to hold, we developed new versions of the bottom up and top down algorithms, called BUCCUST and TDCUST, respectively, which locally exploited summarizability properties at nodes where these held, but ran the full (unoptimized) algorithm at other nodes. Thus, these algorithms were able to obtain the benefits of optimization without computing incorrect results.

In Fig. 10 we show results from running one representative query: cube articles by /author, /month, /year, and /journal. The number of input trees is 220 thousand. For the DTD of DBLP, we know that author is possibly repeated and missing, year and journal are mandatory and unique, and month is possibly missing.

The DBLP cube is dense, and the dimension number is low (4), so it is not a surprise the COUNTER wins. More interesting, we find that BUCCUST has performance significantly better than BUC, while still producing correct results, which the even faster BUCOPT does not. Similarly, TDCUST does a little better than TD, but not as well TDOPT, let alone TDOPTALL, neither of which computes the correct result.

4.6 Performance Summary

Overall, the graphs show the following points. The counter-based algorithm is always optimal when the cube (our counters) is small enough to fit in memory, which is the case when the number of axes is small. When the cube is large, the counter-based algorithm will start to thrash, and even not be able to finish in one pass (because we ran in Windows OS, and so, a process cannot allocate more than 2GBytes)(e.g. in Fig. 5, at 6 axes, we had to do 2 passes, at 7 axes we needed 5 passes). The top-down based algorithm incurs more overheads when the total coverage does not hold, because the efficiency of the algorithm relies heavily on the computation of a coarser aggregate from finer aggregate without keeping individuals around. For sparse data cubes, BUC is a clear winner, as in the relational case. But when the summarizability properties do not hold, particularly for dense cubes, we may have no choice but to use COUNTER.

In summary, summarizability together with cube characteristics determine the choice of the algorithm. The bottom-up algorithm is best in average for a high dimensional cube. The counter-based is best for a low dimensional cube. Only if the cube is dense and total coverage is known to hold that we can efficiently use the top-down algorithm. Knowing that disjointness holds does also improve the performance for both the top-down and the bottom-up algorithms. Having the knowledge of the schema enables us to take advantage of the summarizability that holds at certain lattice points.

5 Related Work

The cube operator was introduced in [9] and has since been studied extensively in the literature. Many algorithms have been proposed for fast cube computation, including [1, 5, 8, 19, 24].

The need to allow flexibility in the dimension axes of data warehouses was elucidated in [13]. Considering overlapped data and missing data as uncertainty and imprecision problems in the database, D. Burdick et. al. [3] incorporated statistical criterion into the OLAP data model. The aggregation operators were modified to produce results that correspond to the statistical measures.

Grouping in XML was first discussed in [18], and has since been carefully worked out in [7]. OLAP cube specification for XML that embeds into XQuery and the aggregation operator were defined in [22]. Beyer et. al. [4] proposed an XQuery extension of explicit group-by clauses to accommodate advanced analytic queries. Issues of aggregate computation have been studied in [10, 15]. They identified conditions for summarizability to characterize the OLAP and statistical data. Having complex summarizability involved, [10] extended the multidimensional model to support data heterogeneity.

Structural relaxation for XML querying has been studied by several authors [2, 14, 16, 17, 20].

6 Conclusion And Future Work

As XML becomes the data interchange standard, it is a natural candidate for storing heterogeneous data in a warehouse. The use of XML in this context raises both semantic and computational challenges, due to the flexible tree-structuring of XML. We address these challenges in this paper, with focus on the computation of a data cube. We developed a conceptual model of aggregate relaxation that is appropriate for XML data cube computation. We identified the summarizability properties crucial to efficient computation of such relaxation. We developed variants of standard data cube computation algorithms that could compute the correct results even without the benefit of summarizability. We showed that certain families of cube computation algorithms required certain summarizability properties for efficiency, so that the choice of algorithm should be dictated, unlike in the relational context, by the semantics of the cube being computed. Automated determination of lattice properties from available schemas that helps choosing and optimizing cube computation algorithms is a natural direction for future exploration.

References

[1] S. Agarwal, R. Agrawal, P. M. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan, and S. Sarawagi. On the Computation of Multidimensional Aggregates. In Proc. VLDB, 1996.

[2] S. Amer-Yahia, S. Cho, and D. Srivastava. Tree Pattern Relaxation. In Proc. EDBT, 2002.

[3] D. Burdick, P. M. Deshpande, T. S. Jayram, R. Ramakrishnan, and S. Vaithyanathan. OLAP Over Uncertain and Imprecise Data. In Proc. VLDB, 2005.

[4] K. Beyer, Don Chambérlin, L. S. Colby, F. Özcan, Ha. Pirahesh, and Y. Xu. Extending XQuery for analytics. In Proc. SIGMOD, 2005.

[5] K. Beyer, and R. Ramakrishnan. Bottom-Up Computation of Sparse and Iceberg Cubes. In Proc. SIGMOD, 1999.

[6] DBLP XML records. http://dblp.uni-trier.de/xml/.

[7] A. Deutsch, Y. Papakonstantinou, and Y. Xu. Minimization and Group-By Detection for Nested XQueries. In Proc. ICDE, 2004.

[8] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. Ullman. Computing Iceberg Queries Efficiently. In Proc. VLDB, 1998.

[9] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. In Proc. ICDE, 1996.

[10] C. A. Hurtado, and A. O. Mendelzon. Reasoning about Summarizability in Heterogeneous Multidimensional Schemas. In Proc. ICDT, 2001.

[11] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. V. S. Lakshmanan, A. Nierman, S. Paparizos, J. M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. TIMBER: A native XML database. *The VLDB Journal*, 11(4):274–291, 2002.

[12] H. V. Jagadish, L. V. S. Lakshmanan, D. Srivastava, and K. Thompson. TAX: A Tree Algebra for XML. In Proc. DBPL, 2001.

[13] H. V. Jagadish, L. V. S. Lakshmanan, and D. Srivastava. What can Hierarchies do for Data Warehouses? In Proc. VLDB, 1999.

[14] D. Lee, and D. Srivastava. Counting Relaxed Twig Matches in a Tree. In Proc. DASFAA, 2004.

[15] H. -J. Lenz, and A. Shoshani. Summarizability in OLAP and Statistical Databases. In SSDBM, 2001.

[16] Y. Li, C. Yu and H. V. Jagadish. Schema-Free XQuery. In Proc. VLDB, 2004.

[17] R. A. O'Keefe and A. Trotman The Simplest Query Language That Could Possibly Work. In Proc. INEX Workshop, 2004.

[18] S. Paparizos, S. Al-Khalifa, H. V. Jagadish, L. V. S. Lakshmanan, A. Nierman, D. Srivastava and Y. Wu. Grouping in XML. In EDBT Workshop on XML Data Management (XMLDM'02), 2002.

[19] K. A. Ross, and D. Srivastava. Fast Computation of Sparse Datacubes. In Proc. VLDB, 1997.

[20] T. Schlieder. Schema-Driven Evaluation of Approximate Tree-Pattern Queries. In Proc. EDBT, 2002.

[21] University of Washington XML Repository. Available at http://www.cs.washington.edu/research/xmldatasets/.

[22] H. Wang, J. Li, Z. He, and H. Gao. OLAP for XML Data. In CIT, 2005.

[23] N. Wiwatwattana, H. V. Jagadish, L. V. S. Lakshmanan, and D. Srivastava. Xˆ 3: A Cube Operator for XML OLAP - web supplement. Available at http://www.eecs.umich.edu/db/timber/xcube/.

[24] D. Xin, J. Han, X. Li, and B. W. Wah. Star-Cubing: Computing Iceberg Cubes by Top-Down and Bottom-Up Integration. In Proc. VLDB, 2003.