On the Use of Query-driven XML Auto-Indexing

Karsten Schmidt, Theo Härder

Dept. of Computer Science
University of Kaiserslautern, Germany

kschmidt|haerder@cs.uni-kl.de

***Abstract*— Autonomous index management in native XML DBMSs has to address XML's flexibility and storage mapping features, which provide a rich set of indexing options. Change of workload characteristics, indexes selected by the query optimizer's "magic", subtle differences in the expressiveness of indexes, and tailor-made index properties ask—in addition to (long-range) manual index selection—for rapid autonomic reactions and self-tuning options by the DBMS. Hence, when managing an existing set of indexes (i.e., a configuration), its cost trade-off has to be steadily controlled by observing query runtimes, index creation and maintenance, and space constraints.**

I. Introduction

Autonomous database management is still a hot topic in DBMS research [1], [2]. Fundamental progress was made in the area of physical design tuning, e.g., [3], [4], and is continued in the XML field, too [5], [6]. However, self-tuning features for indexing, the most effective optimization area, have not fully proven their merit, notably, in XML databases. So far, almost all approaches are limited to a subset of XML index types—primarily simple path indexes [7]–[10] and do hardly exploit the flexible mechanisms provided by native XML storage engines such as [6]. But, utilizing this flexibility leads to enhanced indexing options, aggravating the yet-complex index selection problem even more.

In addition to clustering and structure-specific compression, tailor-made XML storage mappings [11] provide a variety of options for XML-specific indexing. Manually managing transactional XML databases, the *admin* is confronted with alternatives, allowing for fine-grained specification, scope overlapping, query-dependent clustering, and reconfiguration needs due to workload shifts. On the other hand, proper index selection is heavily dependent on the query optimizer, its statistics and its cost estimation. To support the admin's index selection beyond the simple but coarse ones, e.g., content and element indexes, self-tuning of XML indexes has to 1) consider all types of indexes, 2) understand the cost-based selection of an optimizer to provide good-enough statistics, and 3) manage a cost-based workload-dependent index configuration meeting a given size limit.

II. XML Indexing

Current DBMSs usually support two major index types: 1) pure text-oriented indexes, i.e., *content indexes*, and 2) structure-oriented indexes, i.e., *path indexes* [9]. A third group of hybrid indexes combines structural and content predicates.

To exploit indexing capabilities in a native way, we built our indexing framework on two cornerstones for native XML storage. First, prefix-based node labeling, e.g., OrdPaths or DeweyIDs depicted in Figure 1a) for an XML fragment, provides flexible storage maintenance, query support, and stable addressing in case of updates [12]. Second, a *DataGuide* [13] or *Path Synopsis* [6] summarizes the document structure (see Figure 1b)) irrespective of an XSD and enables space-saving XML mapping [11], thereby virtualizing the structure part, i.e., the inner nodes of the document tree. Furthermore, each node of our path synopsis is equipped with a label used as *path class reference* (PCR); when combined with any document node, a PCR identifies its entire path to the root node. The interplay of DeweyIDs and PCRs greatly accelerates many tasks of XML processing and, in particular, storage mapping and index matching during query processing.

A. XML Index Types

According to our prototype DBMS *XTC* [6], we distinguish at least between four index types:

- **Element Index** The logical structure of an element index is comparable to a 1-index [9], where each distinct XML name tag is associated with a list of node labels carrying this tag. Access is limited by a name tag filter, delivering node sequences of different path classes, but requiring false-positive removal.
- **Content Index** It typically indexes the whole content of an XML document. Usually, its operations distinguish between text and attribute nodes and optionally observe typed values (e.g., integer, double, string, etc.).
- **Path Index** It can be specified by one or more XPath expressions. All nodes addressed by these expressions are indexed independently if they are attribute or element nodes. It is easy to map an XPath expression to a set of path classes and to find index matches.
- **CAS Index** A content and structure index extends the path index by combining value-based and path-based indexing to directly assist specific XQuery predicates. In contrast to path indexes, the CAS index contains text values which also may serve as keys and may be typed, too (e.g., integer, string, etc.).

Most of these index types provide various kinds of clustering (i.e., based on PCRs or DeweyIDs) and allow for efficient prefix-based key compression.

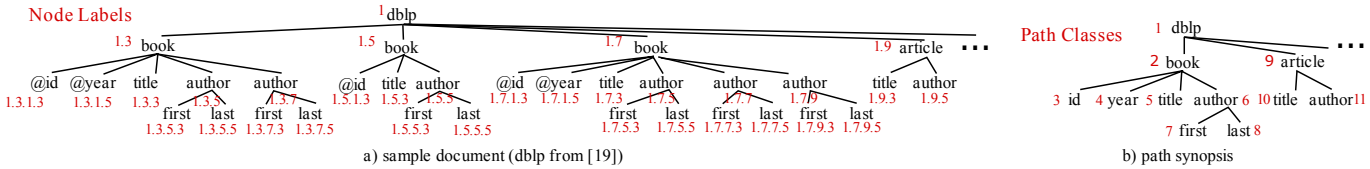

Fig. 1. Sample XML document in a) carrying prefix-based node labels (DeweyIDs) and b) related path synopsis

The so-called *twig* index is yet another kind of index tailor-made for complex twig queries. However, dealing with update queries causes clearly more maintenance cost compared to the afore mentioned types. An index-defining twig, i.e., a tree-pattern query, may need to be executed decoupled from the actual query, only to decide whether the index needs maintenance or not. In contrast, maintenance in case of the four simpler index types can always be decided using a simple set matching to detect path-class inclusion.

B. Variety of Index Use

All index types introduced in Section II-A serve certain XML query specifics. For instance, depending on the availability of such an index variety, XMark query 01 [15]

```
let $auction := doc("auction.xml") return
for $b in $auction/site/people/person[@id = "person0"]
return $b/name/text()
```

can be processed with increasing sophistication:

- *Without secondary index*: Query evaluation plan (QEP) in Figure 2a) indicates that several scans (red boxes) over the entire document index are necessary to join (green boxes) the XPath steps. Expensive navigations (blue box) are needed to retrieve all "@id" attribute values.
- *+Content Index*: This option may replace the navigation operator by a content index scan (dark green box) to reduce document index access (see Figure 2b)).
- *+Element index*: In Figure 2c), the QEP can replace three document scans by element index scans (purple boxes). Note, the resulting node streams may contain nodes originating from different paths to be filtered out.
- *+Path index*: The QEP in Figure 2d) illustrates the use of a path index (brown box) that covers, at least, the path $/site/people/person$. Exploiting this path index allows to remove the left QEP part, avoiding two element index scans and an access to the document root node.
- *+CAS index*: Such indexes have higher selectivity compared to generic content indexes. Hence, an additional CAS index (dark green box in Figure 2e)) can be used to substitute the content index scan. The CAS index result needs to be sorted by their labels (document order).

Even this simple query example impressively illustrates the variability of XML index support. Apparently, it is fairly difficult for a DB admin to define suitable indexes beyond the simple element or content index. Moreover, the variety and flexibility of XML structures and, in turn, query predicates (e.g., wildcards (*), descendant axis (//)) make it impossible to specify static index configurations facing ad-hoc queries.

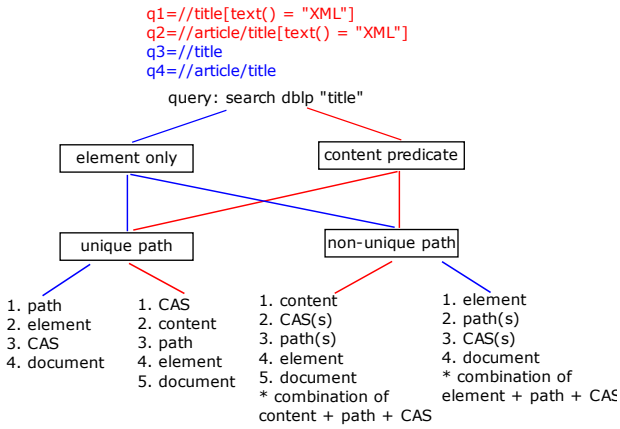

Fig. 3. Index-type decision tree

C. Index Selection in XML

Besides the flexibility of XML, another major challenge comes up when space restrictions are to be met and/or insert, update, and delete (IUD) operations are present. Thus, the well-known index selection problem (from the relational world) is substantially more complex for XML indexing. The evaluation of a path expression containing multiple path steps may exploit different index types in various combinations (to avoid unwanted document scans). As a typical example, Figure 3 reveals considerable complexity to only search for a proper index type—a base task of a query optimizer. Although the listed queries are fairly similar, a heuristics-based query optimizer would identify the query properties and usually look for indexes in differing orders given in the illustration. Because a full set of indexes can never be maintained, index selection typically favors those serving different queries. Furthermore, this simplified decision tree hides the necessity that a cost-based selection usually accounts for size (height) and cluster properties of an index. Moreover, a combination of different indexes and types is possible as well, thus considerably increasing the search space.

Containment Problem: Digging a little deeper in the search space of alternatives reveals the *containment* problem for XML indexes. Related to the introductory example of Section II-B, it is difficult to strictly distinguish index uses as long as they are overlapping, contained in each other, or equal. For the user or DB admin, containment may not be visible during index definition. For instance, an element index specified for "$person$" nodes may have the identical scope as a path index defined on "$//people/person \vee //show/person$". Therefore, an index configuration may contain redundancy to

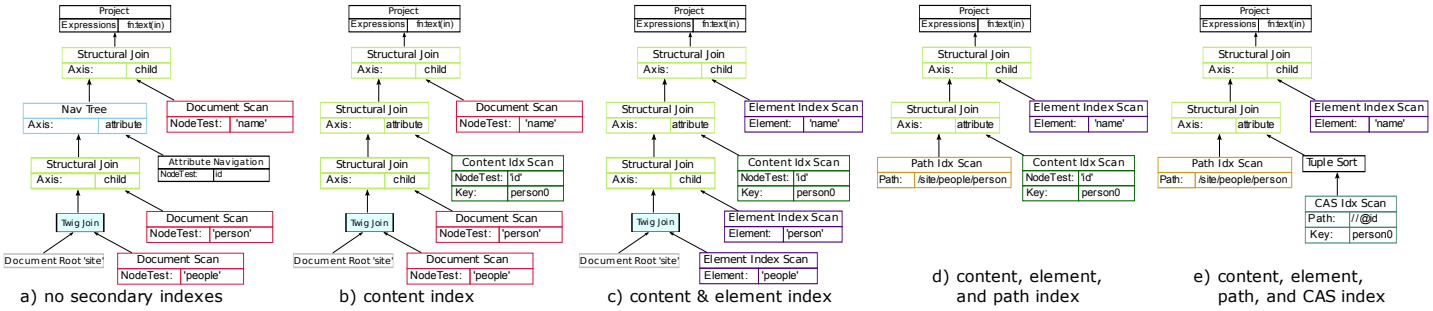

Fig. 2. XMark query 01: Index-driven alternatives in QEPs using XTCcmp [14].

be maintained during updates, too. In contrast, the system may prefer one of these index alternatives due to cost effectiveness resulting from a different clustering or compression overhead. Due to the expressiveness of XPath, it is easy to define indexes containing a subset or being a superset of an existing one. For instance, the set of nodes addressed by the path $/a/b/c/d$ are $\subseteq$ compared to the path $/a//d$. Adding wildcard steps and different index types amplifies the containment problem.

Generalized Indexes: XML indexing allows for tailor-made index definitions to favor specific queries, e.g., a full path containing only child axes. To support as many queries as reasonable, specialized indexes should be combined to more general indexes to share physical structures and, at the same time, improve buffer usage. On the other hand, such a shared usage would provoke increased contention produced by parallel IUD queries, which, in turn, would again advocate the use of more specific indexes. Apparently, this flexibility for indexing and storing XML has to be identified and exploited when autonomous query-driven XML indexing is addressed.

III. Towards Autonomous Indexing

Our autonomous index management is based on the self-tuning principle of MAPE-K [16]—a typical *feedback-control-loop* mechanism—, where queries are intercepted or asynchronously forwarded to an index analysis component. During the analysis, (virtual) index candidates are generated, optimized, and by calling the native query optimizer component, validated for the specific use.

We extended MAPE-K to enable more fine-grained definition and monitoring of index candidates leading to a cost-based index configuration model. Recommendations delivered for index creation and deletion were turned into low-priority jobs and enqueued by the database scheduler. As soon as such (physical) indexes are either created or removed, the query processor immediately uses the new index configuration.

A. Statistics for Decision Support

Index candidate optimization and cost estimation relies on statistics covering document tree properties and indexes.

1) Document Statistics: Because only little additional information is sufficient to estimate index properties needed, document statistics can be kept in an extended path synopsis and collected while the document is stored. The nodes of the path synopsis shown in Figure 4 are extended by a path

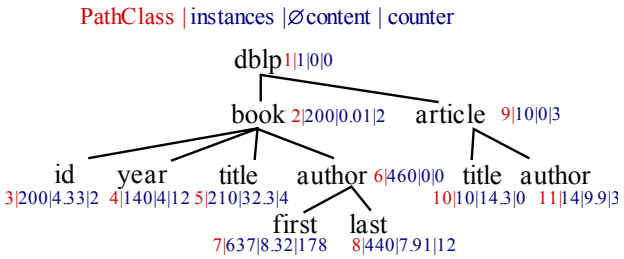

Fig. 4. Extended path synopsis carrying essential document statistics

instance counter, the average content length for this path class, and an update counter for IUD operations. The resulting path synopsis is still a compact data structure (representing typically less than 200 path classes [11]); hence, this extension can be kept in main memory as well (see Table I).

2) Index Statistics: Similar to the document, accurate characteristic values are initially collected during index creation. Therefore, parameters such as B-tree height, no. of leaf pages, cardinality, and index size enhance the index definition.

B. Candidate Generation

Decoupled from the actual index selection of the query optimizer, our index generation solely exploits the internal (normalized) query graph model (QGM) representation [14]. By traversing the QGM bottom-up, starting from the access operators (leave nodes of the QGM, typically scans or navigational operators), axes relationships (e.g., structural joins for child, or descendant axis), value predicates, data types, and node types are collected to assemble step-by-step prospective index candidates. During these steps, candidates with overlapping scopes are retained and collected. The decoupling of this step allows us to apply global knowledge about the documents, queries, indexes, usage, and statistics, which are usually not fully taken into account by a query optimizer.

Cost Model

Autonomous index management dictates several conditions for index selection. At first, the maximum space consumption of all secondary indexes is either an absolute value or is dependent on the actual data size. In the second place, index creation and maintenance negatively impact query processing to be compensated by beneficial index use. Therefore, a cost model has to address creation and maintenance costs as well

as index size. Hence, our extended path synopsis carrying the necessary statistics is used to estimate these costs.

For each index candidate generated, an index size (*IdxStats*) estimation is calculated as shown in Listing 1:

Listing 1. EstimateSize

```
1 IdxStats estimateSize(IndexDef ic, PathSynopsis ps)
2 {
3   // generate path class-based set of synopsis nodes
4   IdxStats stats = new IdxStats(ic);
5   List<PSNode> nodes = match(ic, ps);
6   int size = 0; count = 0, updCnt = 0;
7   for (PSNode node : nodes)
8   {
9     size += node.getInstances() * node.getAvgLength();
10    count += count; updCnt += node.getUpdateCnt();
11  }
12  // estimate index-type-dependent height and #leaves
13  calcIndexStructure(size, stats, count, updCount);
14  return stats;
15 }
```

Because this estimation needs to be done for all virtual index candidates that are generated for the cost-based query optimizer, the matching in line 5 exploits a path cache to speed-up the generation of *PSNode* lists. Larger updates of the extended path synopsis, i.e., new paths added, bulky document updates, or deletions, empty the look-up cache to improve estimation accuracy. The *calcIndexStructure* method in line 13 uses heuristics gained through experiments to account for the indexing overhead. These heuristics consist of cluster-dependent key compression ratios, average descriptor overhead, and typical B-tree occupancy. The resulting *IdxStats* objects contain all index metrics a query optimizer might be interested in.

Candidate Selection

Similar to [17], the index candidates were frequently ranked by their cost-benefit ratio by accounting for QEP benefit and, in turn, index maintenance cost. Because this ranking may require too much time, its frequency is adjusted each time to the recent success of index tuning and overhead. A greedy algorithm observing space restrictions marks indexes according to their rank for the new configuration. Finally, all existing indexes marked for deletion are removed and all virtual indexes marked for materialization enqueued. Similar to MAPE-K, the index manager then invokes queued candidates to be built asynchronously to normal query processing.

IV. Further Optimizations

Due to space restrictions, we only highlight some of our methods. Improving candidate estimation is possible by analyzing the top-k plans, and not only the winner. Several techniques for overhead reduction are available, such as caching of analyses, search space reduction by pruning query-irrelevant candidates. The extended path synopsis helps to solve the containment problem by avoiding index replicas as well as unnecessary scope overlaps and inclusion. Furthermore, virtual indexes can be merged, type-specific preferences are used to improve candidate search, and query analysis is only selectively applied to promising queries. Although not evaluated yet, the index materializer is capable of sharing document scans to build several indexes at a time.

TABLE I
Space Consumption Figures

workload	document / storage	path synopsis	ext. path synopsis	EXSum
XMark	12M / 9.5M	4K	11.2K	14.2K
	112M / 95M	4.1K	11.6K	14.4K
nasa	25M / 13M	0.8K	2.2K	10.5K
lineitem	32M / 13M	0.1K	0.4K	1.6K
treebank	90M / 46M	2.9M	7.3M	63K
dblp	330M / 233M	1.1K	3.3K	6.7K
TPoX collection (scale factor XXS)				
security	126M / 107M	0.7K	2K	11.4K
custacc	58M / 26M	0.7K	2.2K	14K
order	73M / 54M	1K	3.1K	4.6K

V. Evaluation

For the empirical evaluation, we use our own native XML DBMS prototype [6] supporting the four major index types. The following benchmarks are performed on a Pentium IV computer (2 x 3.2 GHz CPUs, 1 GB main memory, 160 GB external memory, Java Sun JDK 1.6).

A. Workloads

We use a variety of workloads to reveal the index tuning capabilities for differing scenarios. The datasets and queries are based on the following XML benchmark suites.

1) XMark: We use XMark [15]—well-known from many projects—to generate XML documents of varying sizes and for queries heavily exploiting different kinds of indexes. This workload mainly consists of a scalable XML document and a sequence of XQuery statements searching or aggregating data.

2) TPoX: This benchmark [18] generates IUD workloads. Although we confined our experiments to single-user mode, TPoX is suitable for concurrent query processing, too.

3) Document Collection: To stress our methods by a wide range of different document types, we also measured overhead-related aspects for a well-known and often used reference collection of XML documents [19].

B. Statistics Maintenance

In our first experiment, we measured the typical overhead of statistics maintenance for autonomous indexing. For this reason, we stored documents of all three workloads and measured additional storage as well as processing time. Table I contains storage consumption figures for a collection of documents, their path synopses, and for the sake of comparison, storage figures needed by the full-fledged XML statistics framework EXsum [20]. These figures confirm a fairly small footprint of the path synopsis extension ($< 1\%$ of the actual document(s)).

Figure 5 reveals the processing time overhead in percent. The first column indicates the amount of extra processing time consumed to gather the statistical values for the extended path synopsis. The second column adds the additional overhead for collecting the index statistics. Fortunately, in almost all workload scenarios, the maximum of both is below $\sim 5-6\%$

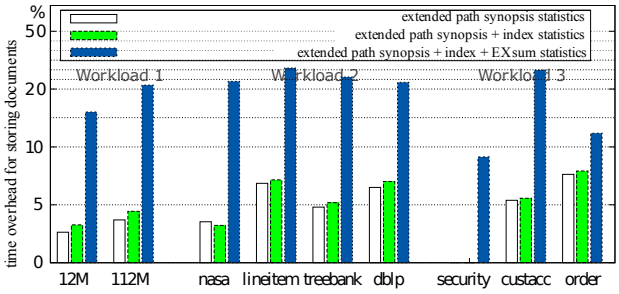

Fig. 5. On-the-fly statistics maintenance: overhead analysis

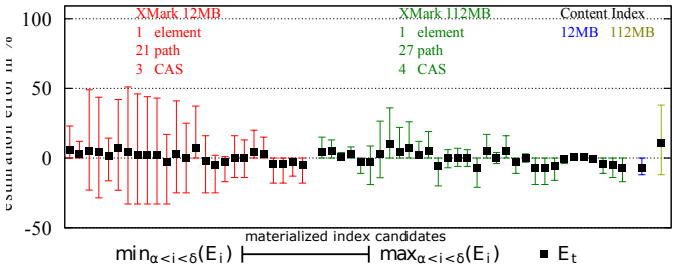

Fig. 6. Estimating index characteristics: error margins

and, note, this overhead occurs only once for each document while mapping it to the DB storage representation. Thus, the overhead caused by the path synopsis extension is negligible compared to the potential future gain. In contrast, the overhead of EXsum [20] is much higher (column 3) and would not pay off for autonomous indexing.

C. Accuracy

Because index tuning heavily depends on cost estimation accuracy, the statistics available need to be properly used to estimate the index shape and, thus, its access cost. In Figure 6, we used our index size estimation from III-B to evaluate and compare all materialized index candidates generated for a subset of 10 different XMark queries in a workload. The total error E_t is the weighted sum of the cardinality error E_c, page number error E_p, height error E_h, and size error E_s (using the ratios of $\frac{estimated}{real}$ values):

$$E_t = \alpha * E_c + \beta * E_p + \gamma * E_h + \delta * E_s, \sum_{\alpha}^{\delta} = 1$$

To augment cost-based query optimization, the weights are adjusted to their expected cost impact. The cardinality error α and the height error γ are weighted by 0.3 each, because the query optimizer's cost estimation heavily depends on them. The remaining weight of 0.4 is equally distributed.

The results in Figure 6 reveal that index estimation is fairly accurate in almost all cases (black dots equal $E_t \leq 12\%$). The minimum and maximum estimation error is shown via the error bars; however, the larger the workload (document size), the lower the min, max, and weighted error. Although content indexes were not used in the experiment, we show on the right side their estimation error, too. Apparently, compared to the tiny overhead caused by statistics gathering, the accuracy of index estimation seems to pay off.

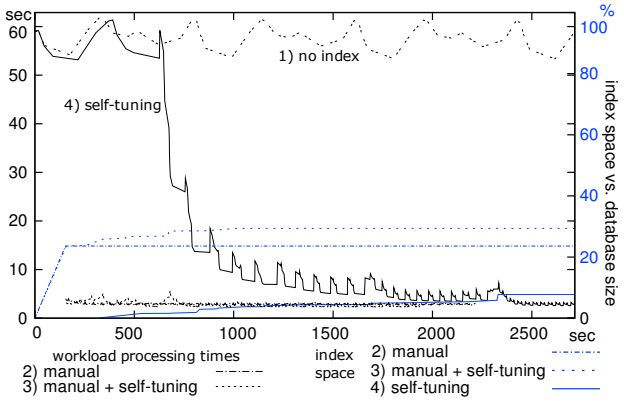

Fig. 7. Workload processing times of query set 1 (XMark).

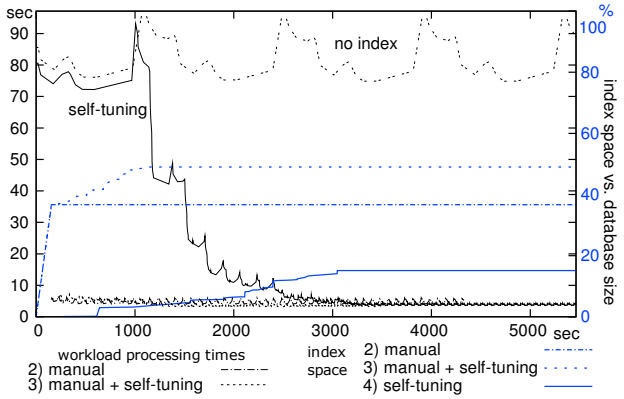

Fig. 8. Workload processing times of query set 2 (XMark).

D. Index Self-tuning Quality

In a further experiment, we want to identify the overall gain drawn from the autonomous index definition and selection. Therefore, we designed the following four scenarios

1) **No indexes**: Unmanaged system.
2) **Manual**: Content and element indexes created, because they are straightforward to define and usually deliver quite an acceptable performance.
3) **Manual + Self-tuning**: Additionally to the pre-selected content and element indexes, the system tunes the configuration autonomously.
4) **Self-tuning**: No indexes were pre-selected, optimization is due to autonomous index tuning.

and illustrate the average tuning gains for varying XMark workloads in Figure 7 and 8, respectively. Both scenarios reveal similar aspects, the manual version delivers quite a satisfactory performance under a constantly high index space overhead, whereas the self-tuning version continuously improves response times down to the same level (or even $\sim 10\%$ better!) as the manual versions, yet consuming clearly less index space. For each new index (noticeable as steps in the colored index space curves in Figure 7 and 8), the quality ratio $\frac{Performance\,Gain}{Overhead}$ decreases slowly, thereby proving the effectiveness of the cost-based index selection.

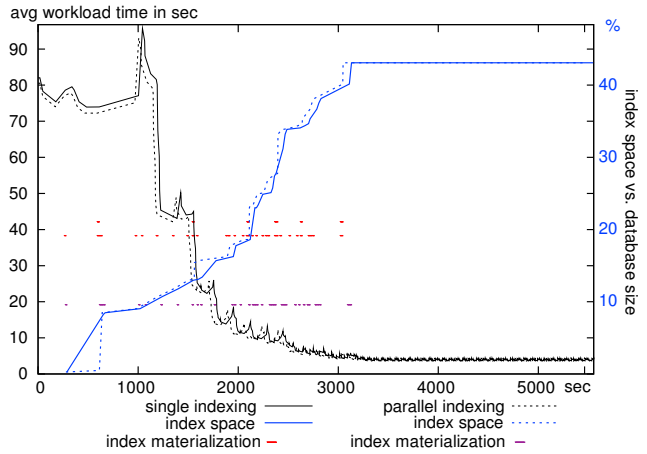

Fig. 9. Impact of parallel index creation

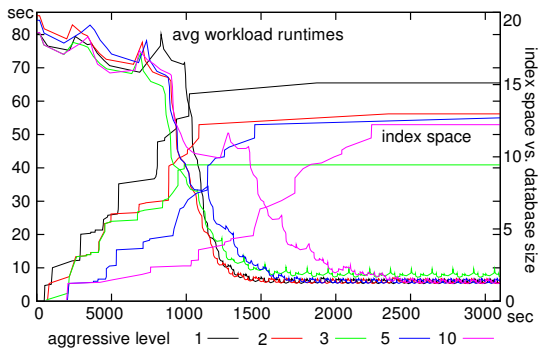

Fig. 10. Threshold / aggressive index building

The question, whether or not parallel indexing (in single-user mode) improves the tuning gain, is answered for some XMark-based benchmarks. Figure 9 tries to reveal correlated effects of our experiments: elapsed times (black), index space (colored), and decreasing runtimes for repetitive workloads for both configurations. The negligible margin of both pairs of curves confirms that 1) parallel indexing has hardly any (negative) influence on workload time and 2) that the growth of the index space is independent of whether or not performed sequentially or in parallel. The scattered bars (red) indicate the phases where additional indexes were created.

Playing around with the multi-programming level yields an upper limit of building four indexes in parallel; but, scan sharing may further increase this limit. In any case, only a couple of workload repetitions is usually necessary to reach for the index configuration a final (stable) state.

In a final experiment, we explored the effects of "aggressiveness" when materializing candidates. Figure 10 shows throughput growth and index space consumption for various thresholds controlling how aggressive a positive cost-benefit ratio turns into an index materialization. Especially the containment problem is automatically addressed by analyzing several queries/workloads to exploit synergy effects of index candidates. This is clearly visible for decision levels > 3. In terms of space overhead, the moderate policies are the most efficient ones. However, nearly all setups increase their throughput in the same order. Thus, it is possible to adjust the threshold to specify whether fast adaptation to changing workloads or conservative space occupation is preferred.

VI. Conclusions

Self-tuning XML indexes may not only enhance manual index setups, but also outperform it. Tailor-made statistics causing little overhead are necessary to compare and figure out the complex alternatives in the XML index search space. In the future, we will combine all the presented tuning knobs (e.g., parallelism, aggressiveness) with major shifts in the workloads together with updates requiring index maintenance.

References

[1] G. Weikum, A. Moenkeberg, C. Hasse, and P. Zabback, "Self-tuning database technology and information services: from wishful thinking to viable engineering," in *VLDB Proc.*, 2002, pp. 20–31.

[2] S. Chaudhuri and V. Narasayya, "Self-tuning database systems: a decade of progress," in *VLDB Proc.*, 2007, pp. 3–14.

[3] D. C. Zilio, J. Rao, S. Lightstone, G. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden, "DB2 design advisor: integrated automatic physical database design," in *VLDB Proc.*, 2004, pp. 1087–1097.

[4] S. Chaudhuri and V. R. Narasayya, "AutoAdmin 'What-if' index analysis utility," in *SIGMOD Proc.*, 1998, pp. 367–378.

[5] K. Schmidt and T. Härder, "Usage-driven storage structures for native XML databases," in *IDEAS Proc.*, 2008, pp. 169–178.

[6] M. P. Haustein and T. Härder, "An efficient infrastructure for native transactional XML processing," *Data Knowl. Eng.*, vol. 61, no. 3, pp. 500–523, 2007.

[7] K. Runapongsa, J. M. Patel, R. Bordawekar, and S. Padmanabhan, "XIST: An XML index selection tool," in *XSym Proc.*, 2004, pp. 219–234.

[8] I. Elghandour, A. Aboulnaga, D. C. Zilio, F. Chiang, A. Balmin, K. S. Beyer, and C. Zuzarte, "An xml index advisor for db2," in *SIGMOD Proc.*, 2008, pp. 1267–1270.

[9] T. Milo and D. Suciu, "Index structures for path expressions," in *ICDT Proc.*, 1999, pp. 277–295.

[10] Q. Li and B. Moon, "Indexing and querying XML data for regular path expressions," in *VLDB Proc.*, 2001, pp. 361–370.

[11] T. Härder, C. Mathis, and K. Schmidt, "Comparison of complete and elementless native storage of XML documents," in *IDEAS Proc.*, 2007, pp. 102–113.

[12] T. Härder, M. P. Haustein, C. Mathis, and M. Wagner, "Node labeling schemes for dynamic XML documents reconsidered," *Data Knowl. Eng.*, vol. 60, no. 1, pp. 126–149, 2007.

[13] R. Goldman and J. Widom, "Dataguides: Enabling query formulation and optimization in semistructured databases," in *VLDB Proc.*, 1997, pp. 436–445.

[14] C. Mathis, A. Weiner, T. Härder, and C. R. F. Hoppen, "Xtccmp: Xquery compilation on xtc," in *VLDB Proc. (Demo Track)*, 2008.

[15] A. Schmidt, F. Waas, M. Kersten, M. J. Carey, I. Manolescu, and R. Busse, "XMark: a benchmark for XML data management," in *VLDB Proc.*, 2002, pp. 974–985.

[16] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[17] M. Lühring, K.-U. Sattler, K. Schmidt, and E. Schallehn, "Autonomous management of soft indexes," in *ICDE Workshops*, 2007, pp. 450–458.

[18] M. Nicola, I. Kogan, and B. Schiefer, "An XML transaction processing benchmark," in *SIGMOD Proc.*, 2007, pp. 937–948.

[19] G. Miklau, "XML data repository," www.cs.washington.edu/research/xmldatasets.

[20] J. de Aguiar Moraes Filho and T. Härder, "EXsum: an XML summarization framework," in *IDEAS Proc.*, 2008, pp. 139–148.