Processing XML Queries and Updates on Map/Reduce Clusters

Nicole Bidoit Dario Colazzo Noor Malla Federico Ulliana
BD&OAK Team - Université Paris Sud - INRIA
{bidoit, colazzo, malla, fulliana}@lri.fr

Maurizio Nolè Carlo Sartiani
DIMIE - Università della Basilicata
{maunole, sartiani}@gmail.com

ABSTRACT

In this demo we will showcase a research prototype for processing queries and updates on large XML documents. The prototype is based on the idea of statically and dynamically partitioning the input document, so to distribute the computing load among the machines of a Map/Reduce cluster. Attendees will be able to run predefined queries and updates on documents conforming to the XMark schema, as well as to submit their own queries and updates.

Categories and Subject Descriptors

H.2 [**Database Management**]: Systems—*parallel databases*

Keywords

XML, Cloud Computing, Map/Reduce

1. INTRODUCTION

In the last few years cloud computing has attracted much attention from the database community. Indeed, cloud computing architectures like Google Map/Reduce [9] and Amazon EC2 proved to be very scalable and elastic, while allowing the programmer to write her own data analytics applications without worrying about interprocess communication, recovery from machine failures, and load balancing. Therefore, it is not surprising that cloud platforms are used by large companies like Yahoo!, Facebook, and Google to process and analyze huge amounts of data on a daily basis.

The advent of this novel paradigm is posing new challenges to the database community. Indeed, cloud computing applications might also be built upon *parallel databases*, that were introduced nearly two decades ago to manage huge amounts of data in a very scalable way. These systems are very robust and very efficient, but for the following reasons their adoption is still very limited: (i) they are very expensive; (ii) their installation, set up, and maintenance are very complex; and, (iii) they require clusters of high-end servers, which are more expensive than cloud computing clusters.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright is held by the author/owner(s).
EDBT'13, Mar 18-22 2013, Genoa, Italy
ACM 978-1-4503-1597-5/13/03

It should also be observed that at least one big company (Amazon) reported severe scalability and robustness issues with parallel databases and preferred to drop their previous DBMS-based infrastructure in favor of a simpler, but much more scalable in-house platform (see Werner Vogels's keynote at VLDB 2007).

Demo Contribution. In this demo we will demonstrate a system that is able to process both queries and updates on very large XML documents. As observed in [7], such very large documents are generated and processed in several contexts, in particular in those involving scientific data and logs.

Our system supports a large fragment of XQuery [5] and XUF (XQuery Update Facility) [13]. The system exploits dynamic and static partitioning to distribute the processing load among the machines of a Map/Reduce cluster. The proposed technique applies when queries and updates are *iterative*, i.e., they iterate the same query/update operations on a sequence of subtrees of the input document. From our experience many real world queries and updates actually meet this property.

Our partitioning technique is schema-less, as the presence of a user-supplied schema is not required; indeed, this technique only relies on path information extracted from the input query/update.

Our system uses Hadoop Map/Reduce [2] as cloud infrastructure and Qizx-open [3] as query/update engine, and required only a few modifications to the query engine to introduce full compression support. To boost the I/O performance across the distributed file system, our system uses EXI [15] compression at each stage of the computation, from data partitioning to query/update execution.

1.1 Related Systems

The partitioning technique employed by our system resembles that of [6], where an horizontal partitioning technique has been proposed in order to ensure parallel execution of single XPath queries. The partitioning technique proposed in that work can be performed only on the main-memory representation of the input document, and, as a consequence, is not suitable for very large XML documents.

In [11] a *vertical* partitioning technique has been proposed still with the aim of parallel and distributed execution of XPath queries. The technique can handle very large documents, but, unlike our system, requires the use of schema information on the input document. Both techniques [6, 11] require strong interventions inside a query engine, while

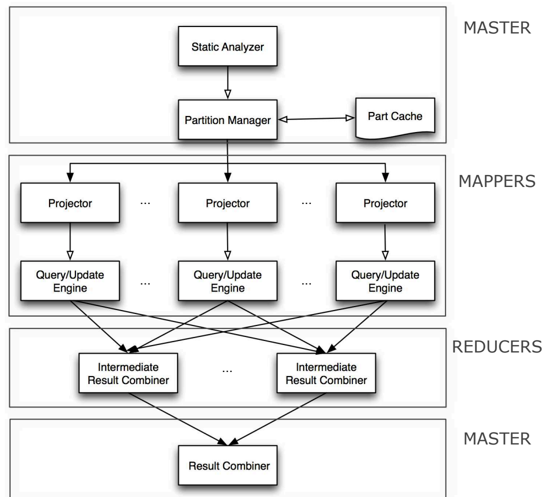

Figure 1: System architecture.

our system required only a minor extension of Qizx-open to support EXI compression.

A recent work [8] proposes new efficient algorithms for the distributed evaluation of XPath queries. This work uses horizontal-vertical partitioning, and assumes data have been statically partitioned according to some pre-existing technique. Another recent work [7] proposes an Hadoop-based architecture for processing multiple twig-patterns on a very large XML document. This system is able to deal with a subset of XPath 1.0 queries, and adopts static partitioning: the input document is statically partitioned into several blocks and some path information is added to blocks to avoid loss of structural information. Differently, our system supports both dynamic and static partitioning, and, importantly, supports mixed workloads containing both XQuery queries and updates.

2. SYSTEM ARCHITECTURE

In this Section we describe the overall architecture of our system, while more details about partitioning, query/update processing and result combination will be given in the next section.

The basic idea of our system is to dynamically and/or statically partition the input data to leverage on the parallelism of a Map/Reduce cluster and to increase the scalability. The architecture of our system is shown in Figure 1.

When a user submits a query or an update to the system, the Static Analyzer parses the input query or update, and extracts relevant information for partitioning the input document D. This information is passed to the Partition Manager, which verifies if D has already been partitioned; in that case, as a single document can be partitioned in multiple ways, the Partition Manager checks if there exists a partition that is still valid (i.e., D has not been updated or externally modified after partitioning), and that it is compatible with the submitted query or update. Parts are stored in the distributed file system, so to be globally available.

If no existing partition can be reused, D is dynamically partitioned according to the partitioning scheme described in Section 3. To overlap partitioning and query/update execution, as soon as a sufficiently large number of parts becomes available, Map/Reduce jobs are asynchronously submitted to the cluster. Parts are encoded as EXI (Efficient XML Interchange) files[1]; this allows the system to significantly reduce the storage space required for parts and, most importantly, to cut network costs. Parts are directly encoded in EXI format during the partition process through the streaming encoder of EXIficient [1].

If, instead, an existing partition can be reused, which is the most common case, the Partition Manager assigns parts to each mapper and launches a Map/Reduce job. Part assignment is performed by taking locality information into account, so to further improve the I/O throughput.

Each mapper works independently on each assigned part. In the case of a query, each part is also *projected*, in order to eliminate all unnecessary elements or attributes from the part; projection is performed according to the path-based projection scheme described in [4] and returns an EXI file. Projected parts reside in the local file system of the mapper and do not survive query execution. In the case of updates, the system ignores projection for the sake of simplifying the global result reconstruction from the updated parts.

After optional projection, the mapper executes the query or the update on each assigned part by invoking Qizx-open, a main-memory query engine. Results returned by Qizx-open are stored in the distributed file system.

Query/update results produced by mappers are combined into a single file in two phases. In the first phase, reducers perform a preliminary result combination, which is then refined by the Result Combiner.

3. PROCESSING QUERIES AND UPDATES

In this section we describe the class of queries and updates that our system supports, and illustrate in more details the data partitioning, query/update processing, and result fusion techniques we use.

3.1 Iterative Queries and Updates

Our system supports the execution of *iterative* XQuery queries and updates, i.e., queries and updates that i) use forward XPath axes, and ii) first select a sequence of subtrees of the input document, and then iterate some operation on each of the subtrees. Iterative queries and updates are widely used in practice, and a static analysis technique has been proposed to recognize them [4, 12].

As an example of iterative query, consider the following query on XMark documents (assume $\$auction$ is bound to the document node $doc(``xmark.xml'')$). The query iterates the same operation on each subtree selected by $\$auction/site//description$ and, hence, is iterative.

```
for $i in $auction/site//description
where contains(string(exactly-one($i)), "gold")
return $i/node()
```

This property is enjoyed by many real world queries: for instance, in the XMark benchmark 13 out of the 20 predefined queries are iterative[2]. Non iterative queries are typically those performing join operations on two independent sequences of nodes of the input documents. Notice that, however, iterative queries may perform join operations, as in the following case:

[1]EXI is a binary format, proposed by the W3C, for compressing and storing XML documents.

[2]Queries from Q_1 to Q_5, Q_{10}, and Q_{14} to Q_{20} are iterative.

```
for $i in $auction/site//description
     $x in $i//keyword
     $y in $i//listitem
where $x = $y
return $x
```

Iterative updates include the wide class of updates that modify a sequence of subtrees, and such that each `delete/rename/insert/replace` operation does not need data outside the current subtree. As an example of iterative update, consider the following one:

```
for $x in $auction/site/regions//item/location
where $x/text() = "United  States"
return (replace value of node $x with "USA")
```

This update iterates over *location* elements and replaces each occurrence of "United States" with "USA". As no information outside the subtrees rooted by *location* elements is required for processing the `replace` operation, the update is iterative.

The following update, instead, is not iterative, as deletion is made according to some condition depending from data outside the deleted subtrees:

```
if $auction//description//text() = "word"
then delete nodes $auction/site/regions/australia//item
```

In this case, the update accesses all *description* elements (in the `if` clause), but deletes nodes in a distinct fragment of the input document.

3.2 Data Partitioning

As described in Section 2, the Static Analyzer parses the input query/update to extract information required for checking the property of Section 3.1 and for partitioning the input data. This information, which is passed to the Partition Manager, is essentially the set of paths used in the query/update, enriched with details about bound variables, and guides the partitioning process.

To illustrate, consider the following iterative query:

```
for $x in /a,
     $y in $x/b
where $y/c/d
return <res> $y/c/e </res>
```

For this query the Static Analyzer extracts the following path set:

$$\{ \quad /a\{for\ x\}, /a\{for\ x\}/b\{for\ y\}, /a\{for\ x\}/b\{for\ y\}/c/d, /a\{for\ x\}/b\{for\ y\}/c/e \ \}$$

By analyzing this path set, the Static Analyzer derives that $/a/b$ is the path on which the query iterates; this path is called *partitioning path* and is used during the partitioning process to identify *indivisible* subtrees, i.e., subtrees that cannot be split among multiple parts. In particular, if a node matches this path, then the whole subtree is kept in the current part; subtrees rooted at nodes outside subtrees selected by the partitioning path can be split across consecutive parts. A possible input XML tree for the above query together with a possible partition $\{t'_1, t'_2\}$ are proposed in Figure 2. The above mentioned indivisibility property is necessary to ensure that query result on the input document is equal to the ordered concatenation of query results on each part. It is easy to verify that this does not hold for the partition $\{t''_1, t''_2, t''_3\}$ in Figure 2, where the first b subtree has been split.

In the case of updates, the system must distinguish between *simple updates*, i.e., updates consisting of a single `delete/rename/insert/replace` operation without `for`-iterations, and update containing iterations. In the first case, the Static Analyzer extracts paths selecting target nodes of the update operations, and considers these paths as partitioning paths. In the second case, the partitioning path is computed as for queries. Composite updates are treated by summing the partitioning paths of each update. As happens for queries, partitioning paths are used to recognise subtrees that should not be divided. Again, this indivisibility property is necessary in order to ensure semantics preservation once the update is distributed over the partition.

To illustrate, consider the following XQuery update performing both a deletion and a renaming:[3]

```
delete nodes //f;
for $x in //c insert <n/> as last into $x
```

For this update we have two partitioning paths $//c$ and $//f$. By using these two partitioning paths, a possible sound partition is $\{t'_1, t'_2\}$ in Figure 2 (target subtrees, that is subtrees selected by partitioning paths, are not split). Note that the partition $\{t''_1, t''_2, t''_3\}$ is unsound for the update since the first subtree rooted in a c node is split: executing the update on this partition entails two insertions for this subtree instead of one.

When a document is partitioned for the first time, the Partition Manager uses the partitioning paths to perform the actual partitioning. The Partition Manager also computes a DataGuide [10] for an input document D. The DataGuide is later used to verify the compatibility of a newly issued query/update with an existing partition, by verifying that the indivisible subtrees identified by the partition paths of the new query/update are already indivisible in an existing partition.

For both queries and updates, the Partition Manager ensures that each part in the partition does not exceed the memory capacity of the main-memory query engine by ending the current part and creating a new one when the size of the current part exceeds a given threshold (if this happens during the visit of an indivisible subtree, then the part is terminated only after the subtree has been totally parsed).

Also, for both queries and updates, artificial tags are added during partitioning to ensure each generated part is well-formed and rooted (so that the query/update engine can process it). For instance, in XML documents corresponding to the partition $\{t'_1, t'_2\}$ of Figure 2, the closing a tag for $\{t'_1\}$ root and the opening a tag for $\{t'_2\}$ root are both artificial (they were not present in the input document t).

3.3 Query/Update Processing

Once the Static Analyzer has extracted path information from the input query/update, and the Partition Manager has found an existing partition or created a new one for processing the query/update, parts are assigned to mappers for query/update processing.

When processing a query, each mapper receives not only the address on the distributed file system of each assigned part, but also the path set extracted by the Static Analyzer. This set is used to project the parts, i.e., to remove

[3]Recall that W3C adopts a snapshot semantics for updates, with no side-effects: first a list of update operations is determined on the same document, and then these operations are executed on the document.

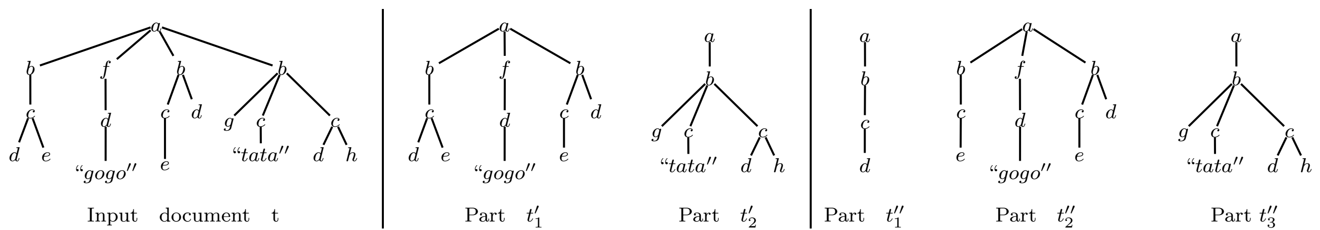

Figure 2: Sound and unsound partitions

elements and attributes not necessary for the query. While original parts are stored in the distributed file system, projected parts are stored in the local file system of the mapper and do not survive query execution. The input query is executed on each projected part by a local instance of Qizx-open, which exports the results, encoded in XML format, to the distributed file system.

When processing an update, instead, projection cannot be applied, as each fragment of a given input part is necessary. As a consequence, the local instance of Qizx-open just executes the update on the original part, discards artificial tags, and stores the updated part, encoded in EXI format, in the distributed file system.

3.4 Result Combination

Result combination works a bit differently for queries and updates. Indeed, partial results of a query can be simply concatenated together, while partial results of an update must be merged.

The combination of partial query results is performed in two steps. In the first step, each reducer receives a set of consecutive part results, which are then combined through high-speed Java NIO channels; the Result Combiner, finally, links together the combined part results produced by the reducers. In the case of updates, combination through Java NIO channels is replaced by EXI file merging.

4. DEMO OVERVIEW

At the demonstration, attendees will be able to pose queries and updates on several very large documents from the XMark benchmark [14]. Attendees will be able to submit either predefined queries and updates or write their own queries/updates and submit them to the system.

Attendees will be able to run any of the iterative queries from the benchmark on instances of the data at different scale factors loaded to a small remote cluster running our system. In addition to the queries from the benchmark, they will also be able to submit predefined updates.

We propose to display an interface that lists all the predefined queries and updates from the benchmark. Once a query or an update have been selected, their submission will take the user to a console that shows relevant information about the status of the cluster. Once the query or the update completes, the user will be able to inspect the processing times for each of the processing phases (i.e., static analysis, data partitioning, data projection, query/update execution, and result combination), and to compare the load of each phase. The user will also be able to run predefined workloads consisting of multiple queries and/or updates.

Through the same interface, attendees will be able to submit their own queries or updates.

5. REFERENCES

[1] Exificient. `http://exificient.sourceforge.net`.

[2] Hadoop. `http://hadoop.apache.org/`.

[3] Qizx-open. `http://www.xmlmind.com/qizxopen/`.

[4] N. Bidoit, D. Colazzo, N. Malla, and C. Sartiani. Partitioning XML documents for iterative queries. In *IDEAS*, 2012.

[5] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML Query Language (Second Edition). Technical report, World Wide Web Consortium, Dec. 2010. W3C Recommendation.

[6] R. Bordawekar, L. Lim, and O. Shmueli. Parallelization of XPath queries using multi-core processors: challenges and experiences. In *EDBT*, 2009.

[7] H. Choi, K.-H. Lee, S.-H. Kim, Y.-J. Lee, and B. Moon. HadoopXML: A suite for parallel processing of massive XML data with multiple twig pattern queries. In *CIKM*, 2012.

[8] G. Cong, W. Fan, A. Kementsietsidis, J. Li, and X. Liu. Partial evaluation for distributed XPath query processing and beyond. *ACM TODS*, 37(4):43, 2012.

[9] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150. USENIX Association, 2004.

[10] R. Goldman and J. Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In *VLDB*, 1997.

[11] P. Kling, M. T. Özsu, and K. Daudjee. Generating efficient execution plans for vertically partitioned XML databases. *PVLDB*, 4(1), 2010.

[12] N. Malla. *Partitioning XML data, towards distributed and parallel management*. PhD thesis, Université Paris Sud, 2012.

[13] J. Robie, D. Chamberlin, M. Dyck, D. Florescu, J. Melton, and J. Siméon. XQuery Update Facility 1.0. Technical report, World Wide Web Consortium, Mar. 2011. W3C Recommendation.

[14] A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A benchmark for XML data management. In *VLDB*, 2002.

[15] J. Schneider and T. Kamiya. Efficient XML Interchange (EXI) Format 1.0. Technical report, World Wide Web Consortium, Mar. 2011. W3C Recommendation.