Queries and Updates on Big XML Documents (Extended Abstract)

Nicole Bidoit, Dario Colazzo, Carlo Sartiani, Alessandro Solimando, Federico Ulliana

Queries and Updates on Big XML Documents (Extended Abstract)

Nicole Bidoit[1], Dario Colazzo[2], Carlo Sartiani[3], Alessandro Solimando[4], and Federico Ulliana[5]

[1] BD&OAK Team - Université Paris Sud -INRIA
[2] LAMSADE - Université Paris Dauphine
[3] DIMIE - Università della Basilicata
[4] DIBRIS - Università di Genova
[5] LIRMM - Université Montpellier 2

Abstract. We present here Andromeda, a system for processing queries and updates on big XML documents. The system exploits static and dynamic partitioning of the input document, so to distribute the computing load among the machines of a Map/Reduce cluster.

1 Introduction

In the last few years cloud computing has attracted much attention from the database community. Indeed, architectures like Google Map/Reduce [1] and Amazon EC2 proved to be very scalable and elastic, while allowing the programmer to write her own data analytics applications without worrying about communication, machine failures, and load balancing issues. 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 to manage huge amounts of data in a very scalable way. These systems are robust and efficient, but their adoption is still very limited due to cost and maintenance issues.

In this paper we present Andromeda, a system able to process both queries and updates on very large XML documents, that are usually generated and processed in contexts involving scientific data and logs [2]. Andromeda supports a large fragment of XQuery [3] and XUF (XQuery Update Facility) [4]. The system exploits dynamic and static partitioning to distribute the processing load among the machines of a Map/Reduce cluster.

2 System Architecture

The basic idea of our system is to dynamically and/or statically partition the input data so 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.

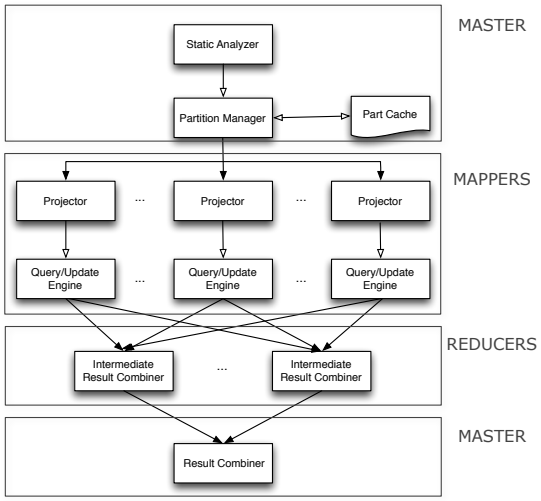


Fig. 1. System architecture.

When a user submits a query or an update to the system, the Static Analyzer extracts from it the information required 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, given that document partitioning is not unique, 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 input query or update.

If reuse is not possible, D is dynamically partitioned according to the partitioning scheme described in Section 3. Parts are encoded as EXI (Efficient XML Interchange) files [5] during the partition process through the streaming encoder of EXIficient [6]; this allows the system to significantly reduce the storage space required for parts and, most importantly, to cut network costs. Parts are stored in the distributed file system, so to be globally available.

If, instead, an existing partition can be reused, which is the most common case, the Partition Manager assigns parts to mappers and launches a Map/Reduce job. 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 [7]. Projected parts reside in the local file system of the mapper as EXI files 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 [8], and stores results in the distributed file system. These results are, then, combined into a single file by means of a two-phase process involving reducers and the Result Combiner.

3 Processing Queries and Updates

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 [7].

As an example of iterative query, consider the following query on XMark documents [9] (assume $\$auction$ is bound to the document node $doc(\text{“}xmark.xml\text{”})$).

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

The query iterates the same operation on each subtree selected by $\$auction/site//description$ and, hence, is iterative.

This property is enjoyed by many real world queries: for instance, in the XMark benchmark 11 out of the 20 predefined queries are iterative. Non iterative queries are typically those performing join operations on two independent sequences of nodes of the input documents.

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.

3.2 Data Partitioning

The partitioning process is driven by the set of paths used in the input query/update, enriched with details about bound variables.

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:

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

By analyzing this path set, the Static Analyzer derives that $/a/b$ is the path on which the query iterates; this path, called *partitioning path*, 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. This property is necessary to ensure that the query result on the input document is equal to the ordered concatenation of query results on each part.

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.

When an input document D is partitioned for the first time, the Partition Manager uses the partitioning paths to perform the actual partitioning and also computes a DataGuide [10] for 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).

3.3 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 elements and attributes not necessary for the query; 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. Therefore, the local instance of Qizx-open just executes the update on the original part and stores the EXI-encoded updated part in the distributed file system.

3.4 Result Combination

Result combination works a bit differently for queries and updates. 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, untouched parts must be merged with updated parts; this process requires the system to read all parts and drop artificial tags introduced by the data partitioning technique.

4 Experimental Evaluation

4.1 Experimental Setup

We performed our tests on a 100-node cluster running Hadoop 2.2 on RHEL and Java 1.7. To reduce issues related to independent system activities, we ran each experiment five times, discarded both the highest and the lowest processing times, and reported the average processing time of the remaining runs.

4.2 Datasets

We performed our experiments on two distinct datasets. The first dataset is dedicated to query experiments, and comprises five XMark [9] XML documents obtained by running the XMark data generator with factors 100, 150, 200, 250, and 300, respectively; the resulting documents have approximate sizes ranging from 10GB to 32GB. The second dataset is used for update tests and contains ten XMark documents whose size ranges approximately from 1GB to 10GB.

4.3 Evaluating Queries

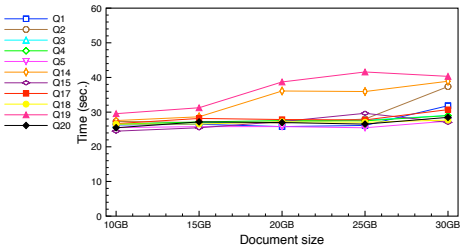
In these experiments we tested the performance and the scalability of our system when processing queries. In the first test we selected the iterative fragment of the XMark benchmark query set (i.e., queries Q_1, Q_2, Q_3, Q_4, Q_5, Q_{14}, Q_{15}, Q_{17}, Q_{18}, Q_{19}, and Q_{20}) and processed each query individually on the documents of the first data set; in this experiment we used parts of size 100000000 bytes. As illustrated in Figure 2(a), the evaluation time is only partially affected by the size of the input document; indeed, Andromeda filters out parts that do not structurally match the input query, and processes the query only on those parts that may give a contribution to the result; hence, even for large documents, the number of machines actually used by the system is below the cluster size.

Partitioning results for exemplifying queries Q_1, Q_2, Q_5, and Q_{14} are reported in Table 1. As it can be easily observed, the partitioning time grows linearly with the size of the input document and the number of used parts is only a small fraction of the total number of parts, with the only notable exception of

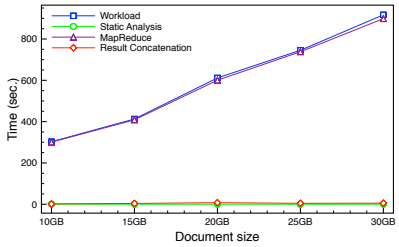
query Q_{14}, which is not very selective. This explains why the processing time of queries Q_{14} and Q_{19}, that uses exactly the same partitioning scheme of query Q_{14}, is bigger than that of the remaining queries.

Table 1. Partitioning time (sec.), generated parts, and used parts (%).

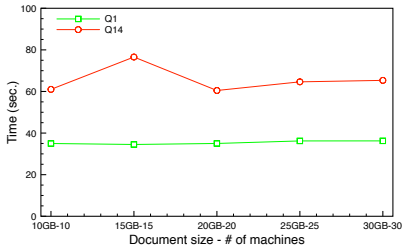
Size	Q_1			Q_2			Q_5			Q_{14}		
	Time	Gen.	Used	Time	Gen.	Used	Time	Gen.	Used	Time	Gen.	Used
10GB	851.686	142	9.1%	706.45	138	22.4%	813.448	144	11.8%	810.014	138	42.7%
15GB	1148	214	8.8%	1060	207	22.7%	1243	217	11.9%	1250	208	42.7%
20GB	1564	285	9.1%	1461	277	22.3%	1666	290	12%	1700	277	42.5%
25GB	2007	357	8.9%	1808	347	22.1%	2215	363	11.8%	2299	347	42.6%
30GB	2391	429	8.8%	2147	417	22.3%	2526	436	11.9%	2534	417	42.4%



(a) Single query experiment.



(b) Query workload experiment.



(c) Query horizontal scalability.

Fig. 2. Query experiments.

Processing workloads In this experiment we evaluated the performance of our system when processing a workload comprising all the queries of the iterative fragment of XMark. As shown in Figure 2(b), workload processing time grows linearly with the size of the input document. This is implied by the fact that, even on smaller documents, the parallel execution of the queries in the workload involves the use of all the machines in the cluster.

Horizontal scalability: changing cluster size In our last experiment on queries we evaluated the horizontal scalability of the system when processing queries Q_1 and Q_{14}: we chose these queries as they are representative of high selectivity (Q_1)

and low selectivity (Q_{14}) queries; Q_{14} also contains a *full-text* predicate that is quite stressful for XQuery engines. In particular, we increased the cluster size as the size of the input document increases, by adding 1 machine per Gigabyte. As shown in Figure 2(c), the system scales beautifully on query Q_1, that exploits only a modest number of machines. Surprisingly enough, we got a similar result for query Q_{14} too. This shows that, even when fully loaded, the system scales well and can efficiently process complex iterative queries.

4.4 Evaluating Updates

In our second battery of experiments we evaluated the performance of Andromeda when processing updates in different scenarios. We evaluated each update in a set of iterative updates against the documents in the second dataset of Section 4.2; in all tests we used parts of 100 millions of bytes (about 95 MB).

Scalability of update processing In our first test we analysed the behaviour of Andromeda when individually executing 16 iterative updates, all returning a new document. Figure 3(a) illustrates the total execution time for each update without partitioning time.

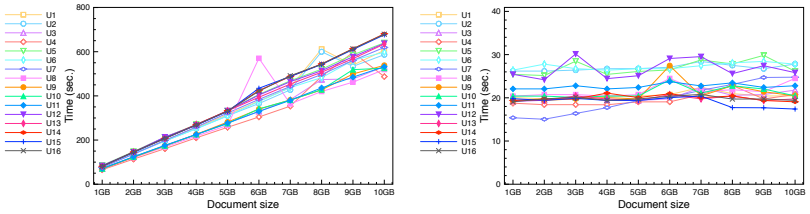
Unlike what happens for queries, update processing is deeply influenced by the input document size, as execution time grows linearly with it. This is motivated by the fact that the system must produce an updated document by combining the updated parts with the parts of the original document that were not touched by the update: this requires the system to traverse all the document parts. To validate this claim we reported in Figure 3(b) the update processing time without part concatenation; as it can be observed, in this case update processing exposes a behavior close to that shown on queries (see Figure 2(a)).

Processing mixed workloads In our second test we created a random query/update workload and analyzed the behaviour of the system when processing the workload on documents of increasing size. The workload comprises 20 expressions randomly chosen by an initialization script, that also chooses the execution order: queries and updates are executed according to the *reader/writer* semantics. Queries and updates are selected by respecting a 80:20 ratio, hence the workload contains 16 queries and 4 updates.

As it can be observed in Figure 3(c), the workload execution time grows linearly with the input size, despite the fact that 16 tasks out of 20 are queries. This is caused by the presence of updates, which not only require result concatenation, but also force the system to partition the updated document for processing the next task, hence making partition reuse much less effective.

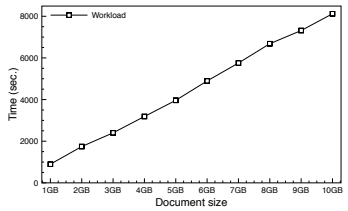
5 Conclusions

In this paper we analysed the performance and the scalability of Andromeda. This analysis confirms that Andromeda scales with the document size and the number of nodes in the cluster, and that it can efficiently process queries and updates on very large XML documents.



(a) Single update experiment: total time.

(b) Single update experiment: MapReduce time.



(c) Query/update workload.

Fig. 3. Update experiments.

References

1. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. In: OSDI, USENIX Association (2004) 137–150
2. Choi, H., Lee, K.H., Kim, S.H., Lee, Y.J., Moon, B.: HadoopXML: a suite for parallel processing of massive XML data with multiple twig pattern queries. In CIKM, ACM (2012) 2737–2739
3. Boag, S., Chamberlin, D., Fernández, M.F., Florescu, D., Robie, J., Siméon, J.: XQuery 1.0: An XML Query Language (Second Edition). Technical report, World Wide Web Consortium (2010) W3C Recommendation.
4. Robie, J., Chamberlin, D., Dyck, M., Florescu, D., Melton, J., Siméon, J.: XQuery Update Facility 1.0. Technical report, World Wide Web Consortium (2011) W3C Recommendation.
5. Schneider, J., Kamiya, T., Peintner, D., Kyusakov, R.: Efficient XML Interchange (EXI) Format 1.0 (Second Edition). Technical report, World Wide Web Consortium (2014) W3C Recommendation.
6. Exificient: `http://exificient.sourceforge.net`.
7. Bidoit, N., Colazzo, D., Malla, N., Sartiani, C.: Partitioning XML documents for iterative queries. In IDEAS, ACM (2012) 51–60
8. Qizx-open: `http://www.xmlmind.com/qizxopen/`.
9. Schmidt, A., Waas, F., Kersten, M.L., Carey, M.J., Manolescu, I., Busse, R.: XMark: A Benchmark for XML Data Management. In: VLDB, Morgan Kaufmann (2002) 974–985
10. Goldman, R., Widom, J.: DataGuides: Enabling query formulation and optimization in semistructured databases. In: VLDB. (1997)