Andromeda: A System for Processing Queries and Updates on Big XML Documents

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

Andromeda: A System for Processing Queries and Updates on Big XML Documents

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. In this paper we present Andromeda, a system for processing queries and updates on large XML documents. The system 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.

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 [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 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.

Our Contribution In this paper we present Andromeda, a system that is able to process both queries and updates on very large XML documents, usually generated and processed in contexts involving scientific data and logs [2].

Our system 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. 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.

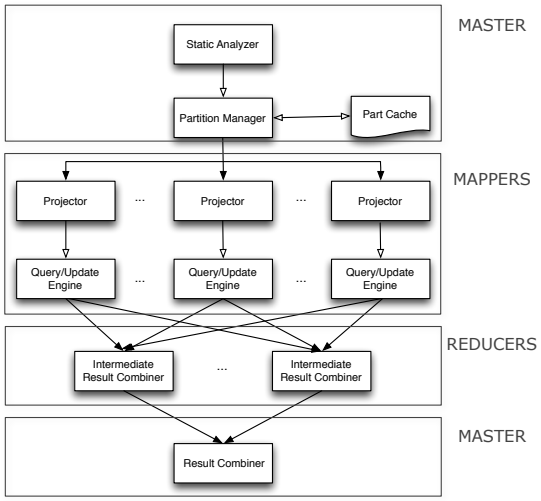


Fig. 1. System architecture.

Paper Outline The rest of the paper is structured as follows. Section 2 introduces the general architecture of Andromeda, while Section 3 details the query and update processing technique used by the system. In Section 4, then, we present an experimental evaluation validating the scalability properties of the system. Finally, in Sections 5 and 6, we discuss some related work and draw our conclusions.

2 System Architecture

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 scheme described in Section 3. During this process, parts are encoded as EXI (Efficient XML Interchange) files[6] through the streaming encoder of EXIficient [5]; this

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

allows the system to significantly reduce the storage space required for parts and, most importantly, to cut network costs.

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.

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 [6] 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 [7], 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

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 [6].

As an example of iterative query, consider the following query on XMark documents [8] (assume $\$auction$ is bound to the document node $doc(\text{“}xmark.xml''))$.

`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[7]. 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:

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

[7] Queries from Q_1 to Q_5, Q_{14}, and Q_{16} to Q_{20} are iterative.

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

As described in Section 2, the STATIC ANALYZER parses the input query/update to extract the 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 it 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 set of paths:

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

By analyzing this set of paths, 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. This 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.

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 split. Again, this indivisibility property is necessary in order to ensure semantics preservation once the update is distributed over the partition.

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 [9] 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).

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 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, and stores its updated version, 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, 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

Our experiments aim at i) proving the efficiency of the system in processing queries and updates on large documents, and ii) showing how the system scales with the document size and the number of nodes in the cluster.

4.1 Experimental Setup

We performed our experiments on a *multitenant* cluster running Hadoop 2.2 on RHEL Linux and Java 1.7. The cluster comprises 1 master node and 100 slave nodes connected through an InfiniBand network. To reduce issues related to independent system activities and other jobs in the cluster, we ran each experiment five times, discarded both the highest (worst) and the lowest (best) processing times, and reported the average processing time of the remaining runs.

4.2 Test Sets

We performed our experiments on two distinct datasets. The first dataset is dedicated to query experiments, and comprises five XMark [8] 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.

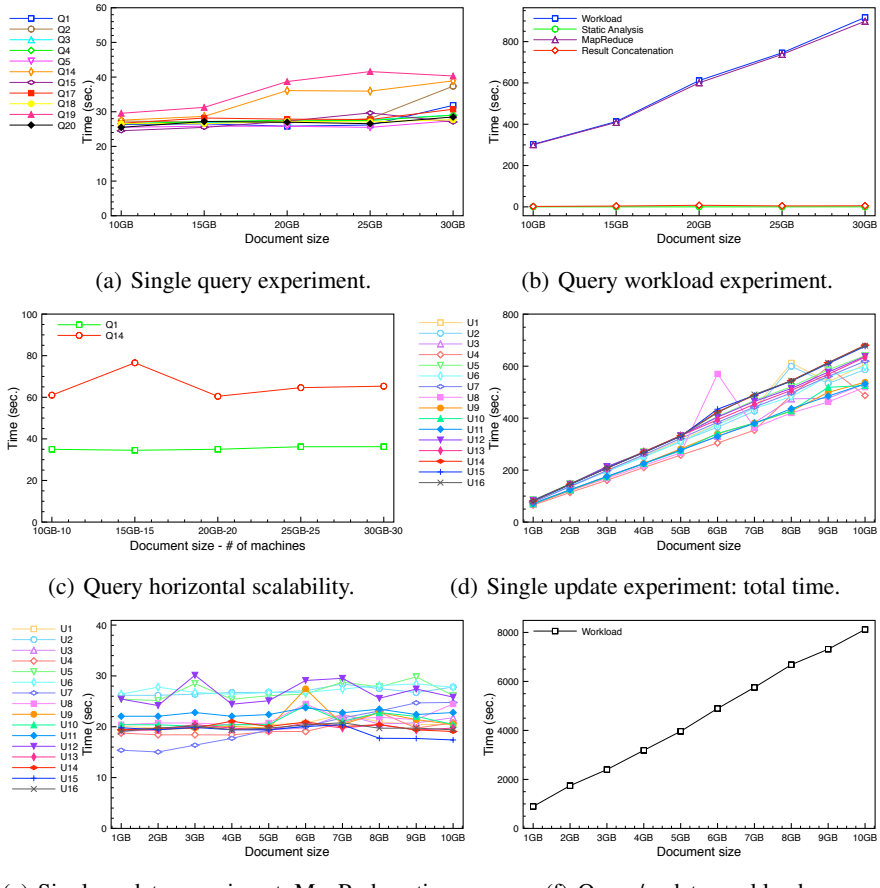
4.3 Evaluating Queries

In our first battery of 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. The results we obtained are shown in Figure 2(a). This graph indicates that the evaluation time is only partially affected by the size of the input document: indeed, given an input query Q, Andromeda filters out parts that do not structurally match Q, and processes Q 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 time for exemplifying queries Q_1, Q_2, Q_5, and Q_{14} is reported in Table 1, together with the number of *generated* parts and the percentage of *used* parts. As we mentioned before, unused parts are discarded. 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.

(d) Single update experiment: total time.

(e) Single update experiment: MapReduce time.

(f) Query/update workload.

Fig. 2. Experiments.

Table 2. Workload: partitioning time (sec.), generated parts, and map input records.

Size	Time	Gen.	Map input records
10GB	694.342	120	1309
15GB	1070106	181	1980
20GB	1424.379	241	2618
25GB	1876.539	302	3289
30GB	2138.638	362	3938

Processing Workloads In our third experiment we evaluated the performance of our system when processing a workload comprising all the queries of the iterative fragment of XMark. The results we collected are shown in Figure 2(b) and Table 2.

In Figure 2(b) we reported the total workload processing time. It is worthy to note that 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, as confirmed by Table 2, which reports the partitioning time, the number of generated parts, and the number of map input records (parts to process) for each input document: as shown in this table, even on the 10GB document the cluster is fully exploited.

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. The results of this experiment are reported in Figure 2(c). As expected, the system scales beautifully on query Q_1, as this 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 these updates return a new document. Figure 2(d) 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; more details on this combination process can be found in [10]. To validate this claim we reported in Figure 2(e) the update processing time without part concatenation; as it can be observed, in this case update processing exposes a behaviour 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, hence queries can be evaluated simultaneously, while updates have to be processed individually. Queries and updates are selected by respecting a 80:20 ratio, hence the workload contains 16 queries and 4 updates. The composition of the workload we considered is reported below:

$$W = (U_2, U_{12}, [Q_{18}, Q_{17}, Q_3, Q_1, Q_{18}],\\ U_4, U_{14}, [Q_{15}, Q_5, Q_2, Q_{17}, Q_{15}, Q_{15}, Q_{20}, Q_{10}, Q_1, Q_5, Q_{18}])$$

Figure 2(f) describes the behaviour of the system when processing the workload. As it can be observed, 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 Related Works

There exist only a few systems able to process queries on XML data in distributed and cloud environments, *e.g.*, ChuQL [11], MRQL [12], HadoopXML [2], PAXQuery [13], and VXQuery [14]. Among them, HadoopXML is the system that most closely resembles Andromeda as it can transparently process XPath queries on an Hadoop cluster. HadoopXML requires a preliminary document indexing phase, close to Andromeda partitioning phase. Despite these similarities, HadoopXML only supports XPath queries, and, unlike Andromeda, cannot process XQuery queries or XUF updates.

PAXQuery and VXQuery are systems for processing XQuery queries on collections of (relatively) small XML documents scattered across a cloud computing cluster. While very efficient even on small clusters, they were not designed to evaluate queries on big documents. MRQL is a query processing system that supports an SQL-like query language that can be used to query XML and JSON data; MRQL directly translates queries into Java code that can be executed on top of Hadoop or Spark. While more powerful than PigLatin, MRQL cannot process complex XQuery queries and does not support updates. ChuQL, finally, is a language embedding XQuery that allows the programmer to distribute XQuery queries over Map/Reduce clusters. The programmer has the duty to manage low-level details about query parallelization, while Andromeda completely hides the underlying processing environment.

To the best of our knowledge, there is no system supporting XUF updates on big XML documents.

6 Conclusions and Future Work

In this paper we described the architecture of Andromeda, and analyzed its performance and scalability. 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.

In the near future we want to extend Andromeda in several ways. First of all, we want to improve the partitioning technique, so to obtain new partitions from existing ones without the need of reading and parsing again the whole input document. Second, we want to explore new techniques for result fusion in order to lower its cost. Finally, we want to understand if and how Hadoop can be replaced with Apache Spark.

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 wen Chen, X., Lebanon, G., Wang, H., Zaki, M.J., eds.: 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. : (Exificient) `http://exificient.sourceforge.net`.
6. Bidoit, N., Colazzo, D., Malla, N., Sartiani, C.: Partitioning XML documents for iterative queries. In Desai, B.C., Pokorný, J., Bernardino, J., eds.: IDEAS, ACM (2012) 51–60
7. : (Qizx-open) `http://www.xmlmind.com/qizxopen/`.
8. 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
9. Goldman, R., Widom, J.: DataGuides: Enabling query formulation and optimization in semistructured databases. In: VLDB. (1997)
10. Malla, N.: Partitioning XML data, towards distributed and parallel management. PhD thesis, Université Paris Sud (2012)
11. Khatchadourian, S., Consens, M.P., Siméon, J.: Having a ChuQL at XML on the cloud. In Barceló, P., Tannen, V., eds.: Proceedings of the 5th Alberto Mendelzon International Workshop on Foundations of Data Management, Santiago, Chile, May 9-12, 2011. Volume 749 of CEUR Workshop Proceedings., CEUR-WS.org (2011)
12. Fegaras, L., Li, C., Gupta, U., Philip, J.: XML Query Optimization in Map-Reduce. In: WebDB. (2011)
13. Camacho-Rodríguez, J., Colazzo, D., Manolescu, I.: PAXQuery: A massively parallel XQuery processor. In Katsifodimos, A., Tzoumas, K., Babu, S., eds.: Proceedings of the Third Workshop on Data analytics in the Cloud, DanaC 2014, June 22, 2014, Snowbird, Utah, USA, In conjunction with ACM SIGMOD/PODS Conference, ACM (2014) 1–4
14. Jr., E.P.C., Westmann, T., Borkar, V.R., Carey, M.J., Tsotras, V.J.: Apache VXQuery: A scalable XQuery implementation. CoRR **abs/1504.00331** (2015)