PAXQuery: Parallel Analytical XML Processing

Jesús Camacho-Rodríguez, Dario Colazzo, Ioana Manolescu, Juan A. M. Naranjo

PAXQuery: Parallel Analytical XML Processing

Jesús Camacho-Rodríguez[*]
Hortonworks Inc.
USA
jcamachorodriguez@hortonworks.com

Dario Colazzo[*]
Université Paris-Dauphine
France
dario.colazzo@dauphine.fr

Ioana Manolescu
Juan A. M. Naranjo
INRIA & Université Paris-Sud
France
ioana.manolescu@inria.fr
juan-alvaro.munoz-naranjo@inria.fr

ABSTRACT

XQuery is a general-purpose programming language for processing semi-structured data, and as such, it is very expressive. As a consequence, optimizing and parallelizing complex analytics XQuery queries is still an open, challenging problem.

We demonstrate PAXQuery, a novel system that parallelizes the execution of XQuery queries over large collections of XML documents. PAXQuery compiles a rich subset of XQuery into plans expressed in the PArallelization ConTracts (PACT) programming model. Thanks to this translation, the resulting plans are optimized and executed in a massively parallel fashion by the Apache Flink system. The result is a scalable system capable of querying massive amounts of XML data very efficiently, as proved by the experimental results we outline.

1. INTRODUCTION

Over the last years, the Hadoop Distributed File System (HDFS) has gained popularity as an inexpensive solution to store huge volumes of heterogenous data. MapReduce [8] is arguably the most widely adopted model to analyze data stored in HDFS; its main advantage is that data processing is distributed across many sites without the application having to explicitly handle data fragmentation, fragment placement, etc.

While the simplicity of MapReduce is an advantage, it is also a limitation, since large data processing tasks are represented by complex programs consisting of many *Map* and *Reduce* tasks. In particular, since these tasks are conceptually very simple, one often needs to write programs comprising many successive tasks, which limits parallelism. To overcome this problem, recent efforts have focused on proposing more expressive dataflow abstractions for massively parallel analytics data processing [10, 22].

The *PArallelization ConTracts* programming model [3] (**PACT**, in short) is one of the proposals that pushes the idea of MapReduce further. In a nutshell, PACT generalizes MapReduce by (i) manipulating records with any number of fields, instead of (key, value) pairs, (ii) enabling the *definition of custom parallel operators* by means of second-order functions, and (iii) allowing one parallel operator to receive as input the outputs of *several* other such operators. Due to its declarative nature, a PACT program can have multiple physical execution plans with varying performance. At compile time, the compiler choses an optimal strategy (plan) that maximizes parallelisation opportunities, and thus efficiency. The PACT model is implemented within the open-source Apache Flink data processing engine[1] [10].

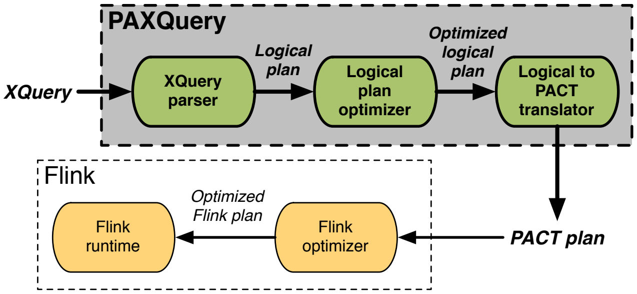


Figure 1: PAXQuery architecture overview.

Optimizing and parallelizing complex analytics queries on semi-structured data is extremely challenging and still a widely under-explored topic. For instance, given a very large collection of XML documents, evaluating a query that *navigates over these documents and also joins results from different documents* raises performance challenges, which may be addressed by parallelism.

Our demonstration features the PAXQuery system [6], a massively parallel processor of XML queries. Inspired by other high-level data analytics languages that are compiled to MapReduce and other dataflow abstractions (e.g. Pig [17] or Hive [23]), PAXQuery proposes a layered architecture to efficiently translate XQuery [24] into PACT plans. The main advantage of this approach is *implicit parallelism*: neither the application nor the user need to partition the XML input or the query across nodes. This contrasts with prior work [4, 7, 13]. Thus, we can rely on the Flink platform for the optimization of the PACT plan and its automatic transformation into a dataflow that is evaluated in parallel on top of HDFS; these steps are explained in [3].

In the sequel, Section 2 describes the PAXQuery architecture, and provides a beginning-to-end query translation example. Section 3 presents experimental results confirming the interest of PAXQuery' parallel XQuery processing approach. Section 4 describes the demonstration scenario and Section 5 concludes.

[*] Part of this work was performed while the authors were with Université Paris-Sud and INRIA. The work was partially supported by the KIC EIT ICT Labs Europa activity 12115.

[1] Flink was known as Stratosphere prior to becoming an Apache incubator project in July 2014.

http://dx.doi.org/10.1145/2723372.2735374

2. PAXQUERY ARCHITECTURE

Our approach for implicit parallel evaluation of XML queries is to *translate* them into PACT plans as depicted in Figure 1. Then, PAXQuery sends the PACT plan to the Flink platform, which optimizes it, and turns it into a data flow that is evaluated in parallel, as explained in [3].

The experiments we present in this work demonstrate not only that PAXQuery (unlike competitor systems [4, 7, 13]) parallelizes XQuery evaluation without any user intervention, but it is also more efficient than comparable implementations based on single-site processors running in parallel. This is because of its reliance on a highly parallel back-end (Flink), which is enabled by our algebraic translation methodology.

Below, we describe the steps taken by PAXQuery to translate an input XQuery query into an efficient PACT plan.

2.1 XQuery parser

Query language PAXQuery supports a representative fragment of XQuery [24], the W3C's standard query language for XML data, which has been recently enhanced with strongly requested features geared towards XML analytics. In particular, our goal was to cover (*i*) the main navigational mechanisms of XQuery, and (*ii*) its key constructs to express analytical style queries e.g. aggregation, explicit grouping, and rich comparison predicates. The grammar of the covered XQuery fragment can be found in [6].

Example. The following XQuery (over collections of documents extracted from an XMark [21] document) *lists the maximum price per postal code of any item bought or sold in France*:

```
let $pc := collection('people'),
    $cc := collection('auctions')
for $p in $pc//person[//country/text()='FR'],
    $i in $p/@id, $z in $p/zipcode/text()
let $r :=
  for $c in $cc//closed_auction, $b in $c/buyer/@person,
      $s in $c/seller/@person, $pr in $c/price
  where $i = $b or $i = $s
  return $pr
group by $z
return <res zip={$z}>{max($r)}</res>
```

The above query shows some of the main features of our XQuery fragment. It includes FLWR expressions, which are powerful enough to express complex operations like *iterating over sequences, joining multiple documents, and performing grouping*. XPath paths start from the root of each document in a *collection* of XML documents, or from the bindings of a previously introduced *variable*; the XPath fragment we support corresponds to $\text{XPath}^{\{/,//,[]\}}$ [15]. Our fragment supports rich predicates expressed in disjunctive normal form (DNF), including value- and node identity-based comparisons. Other XQuery constructs such as *if* or *switch* expressions can be integrated into our proposal in a straightforward manner.

From XQuery to logical plans In a first step, PAXQuery represents the XQuery query as an *equivalent algebraic expression* [6]. XQuery translation into algebraic formalisms has been extensively studied [1, 5, 9, 14, 20].

A significant source of XQuery complexity comes from *nesting*: an XQuery expression can be nested in almost any position within another. In particular, nested queries challenge the processor, as straightforward translation into nested plans leads to poor performance. Effective translation techniques represent nested XQuery as *unnested plans relying on joining and grouping*. Depending on the query shape, such *decorrelating* joins may be *nested* and/or *outer* joins.

We introduce our algebraic representation of XQuery by means of our running example, relying on the algebra of [14].

Example (*continuation*). The algebraic plan corresponding to the XQuery introduced previously is shown in Figure 2.a.

The XML *scan* operators take as input the `'people'` (respectively `'auctions'`) collection of XML documents and create a tuple out of each document in the collection.

XQuery may perform *navigation*, which, in a nutshell, *binds variables to the result of path traversals*. Navigation is commonly represented through *tree patterns*, whose nodes carry the labels appearing in the paths, and where some *target nodes* are also annotated with names of variables to be bound, e.g. `$pc`, `$i`, etc. Our algebra uses a *navigation* operator parameterized by an *extended tree pattern* (ETP) supporting multiple returning nodes, child and descendant axis, and nested and optional edges [6]. This allows *consolidating as many navigation operations as possible in a query, within a single navigation tree pattern*; in particular in a *navigation performed outside of the* `for` *clauses*, which leads to more efficient matching against XML documents [1].

The operator *navigation*(e_1) concatenates each input tuple successively with all @id attributes (variable `$i`), and text values of `country` (`$x1`) and `zipcode` (`$z`) elements resulting from the embeddings of e_1 in the value bound to `$pc`. Observe that the variable `$x1` did not appear in the original query; in fact, it is created by PAXQuery to hold the value needed for the *selection* operator above it. The operator *navigation*(e_2) is generated in a similar fashion.

Above the previous operators, we find a *nested left outer join* on a disjunctive predicate, which brings together people and the auctions they participated in, either as buyers *or* sellers. Observe that the join is *outer*, i.e., all people are kept in the output, even if they did not participate in any auction.

Then, we *group* the tuples coming from the previous operator by the value of their `zipcode`, and the result of the *aggregation* function *max* is calculated and concatenated to each of these tuples.

Finally, the XML construction operator is responsible for transforming a collection of tuples to XML. *For each tuple* in its input, *construct*(L) builds *one XML tree for each construction tree pattern* in the list L; more details can be found in [6].

2.2 Logical plan optimizer

After building a logical plan, PAXQuery optimizes it by using rewriting rules that create semantically equivalent alternative plans. The goal of these transformations is preparing the plan for the translation into a more efficient PACT plan. PAXQuery implements well-studied plan transformation rules [19], e.g. *push down projections*, *push down selections*, etc. We illustrate some of them next.

Example (*continuation*). The plan obtained after applying our transformation rules is shown in Figure 2.b (for readability reasons, *projection* operators have been omitted). First, observe that the *navigation* is integrated within the *scan* operator (*NavScan*), so the tuples resulting from the embeddings of the ETPs can be extracted as we read the XML documents. Further, note that the selection on the people whose `country` is France has been pushed into the ETP e_1. Finally, the *group-by* and *aggregation* operators have been rewritten into a single one that represents both operations.

2.3 Logical to PACT translator

The PACT model [3] is a generalization of MapReduce. A PACT plan is a DAGs of *implicitly parallel operators*, that are optimized and translated into *explicit parallel data flows* by Flink.

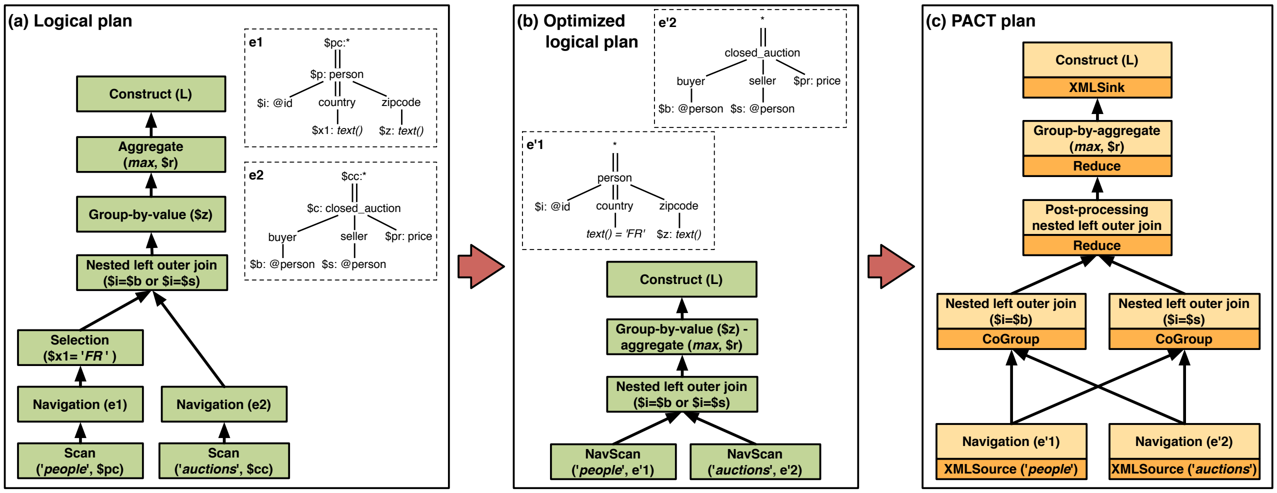


Figure 2: Sample translation from a logical plan to PACT.

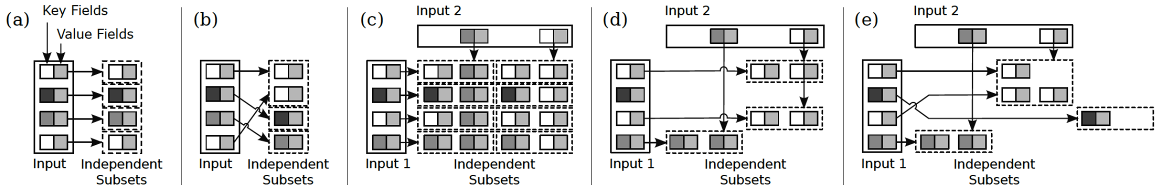


Figure 3: (a) Map, (b) Reduce, (c) Cross, (d) Match, and (e) CoGroup parallelization contracts.[2]

Data model. PACT plans manipulate *records* of the form $r = ((f_1, f_2, \ldots, f_n), (i_1, i_2, \ldots, i_k))$ where $1 \leq k \leq n$. The first component $(f_1, f_2, \ldots, f_n)$ is an ordered sequence of *fields* f_i; in turn, a field f_i is either an atomic value (string) or a ordered sequence $(r'_1, \ldots, r'_m)$ of records. On the other hand, the second component $(i_1, i_2, \ldots, i_k)$ is an ordered, possibly empty, sequence of record positions in $[1 \ldots n]$ indicating the *key* fields for the record. The *key of a record* r, denoted by $r.key$, is the list of all the key fields $(f_{i_1}, f_{i_2}, \ldots, f_{i_k})$.

Processing model. *Data sources and sinks* are, respectively, the starting and terminal nodes of a PACT plan. The input data is stored in files, e.g. in HDFS; a function parameterizing data source operators specifies how to structure the data into records. In turn, results can be output to files, with the destination and format similarly controlled by an output function.

The rest of data processing nodes in a PACT plan are *operators*. An operator manipulates bags of records. Its semantics is defined by (i) a *parallelization contract*, which determines how input records are organized into *groups*; (ii) a *user function* (or UF) that is executed independently over each bag (group) of records created by the parallelization contract (these executions can take place in parallel); and (iii) optional *annotations* that may enable further optimizations by Flink.

A set of the most common parallelization contracts is built in Flink: Map, Reduce, Cross, Match, and CoGroup (see Figure 3). The Map contract forms an individual group for every input record. The Reduce contract forms a group for every unique value of the key attribute in the input data set, and the group contains all records with that particular key value. The Cross contract builds the Cartesian product of the two input bags. The Match contract forms a group from every pair of records in its two inputs, only if the records have the same value for the key attribute. Finally, the CoGroup contract forms a group for every value of the key attribute (from the domains of both inputs), and places each record in the appropriate group.

Table 1: Algebra to PACT overview.

Algebra operators		PACT operators (#)
Scan / NavScan		Source (1)
Construct		Sink (1)
Navigation		Map (1)
Group-by / Group-by-aggregate		Reduce (1)
Flatten		Map (1)
Selection		Map (1)
Projection		Map (1)
Aggregation (on nested field)		Map (1)
Aggregation (on top-level field)		Reduce (1)
Duplicate elimination		Reduce (1)
Cartesian product		Cross (1)
Conjunctive equi-join	Inner	Match (1)
	Outer	CoGroup (1)
	Nested outer / Nested outer-aggr.	CoGroup (1)
Disjunctive equi-join (n conjunctions)	Inner	Match (n)
	Outer	CoGroup (n) & Reduce (1)
	Nested outer / Nested outer-aggr.	CoGroup (n) & Reduce (1)
Inequi-join	Inner	Cross (1)
	Outer	Cross (1) & Reduce (1)
	Nested outer Nested outer-aggr.	Cross (1) & Reduce (1)

[2] Figure reproduced from [12] with authorization.

From logical plans to PACT plans XQuery algebraic plans are translated into PACT plans recursively, operator by operator; for each XQuery operator, the translation outputs one or several PACT operators for which we need to choose the *parallelization contract* (and possibly its corresponding *key fields*), and the *user function*, which together determine the PACT behavior. Further, we annotate the PACT operators to take fully advantage of Flink's optimizer. More details about these steps can be found in [6].

Table 1 provides an overview on the algebra operators and the contracts used by the PACT operators resulting from our translation. Observe that the translation of the binary operators is the most complex, as it has to deal efficiently with the *nested* and/or *outer* nature of some joins, which may result in multiple operators at the PACT level. We illustrate PAXQuery translation to PACT with the following example.

Example (*continuation*). For the algebra plan in Figure 2.b, PAXQuery generates the PACT program shown in Figure 2.c; the key fields for each operator are omitted for readability. The XML source operators scan (in parallel) the respective collections and apply the *navigation* UF over each document to create records. The nested outer join is translated into two CoGroup operators and a post-processing Reduce. The core difficulty to address by our translation is to correctly express (i) the *disjunction* in the where clause of the query, and (ii) the *outerjoin* semantics (recall that in this example a <res> element must be output even for people with no auctions). The main feature of the *nested left outer join* UF associated to each CoGroup is to guarantee that no erroneous duplicates are generated when the parallel evaluation of more than one conjunctive predicate is true for a certain record. The Reduce operator groups all the results of the previous CoGroup operators having the same left hand-side record, and then the *post-processing* UF associated to it is applied to produce the final result for the join. The following Reduce groups together the records with the same zipcode and calculates the *aggregation* function over the price in each of them. Finally, the XML sink builds and returns XML results.

3. PAXQUERY SCALABILITY AND ALTERNATIVES

PAXQuery is implemented in Java 1.6, and runs on top of the Flink platform [10] supporting PACT. We describe an extended experimental evaluation in [6]; below, we borrow from that article the two most significant experiments, related on one hand to the platform scalability, and on the other to the comparison between PAXQuery and alternative massively parallel XQuery architectures.

Experimental setup. The experiments run in a cluster of 8 nodes on an 1GB Ethernet. Each node has 2 × 2.93GHz Quad Core Xeon CPUs, 16GB RAM and two 600GB SATA hard disks and runs Linux CentOS 6.4. PAXQuery is built on top of Flink 0.2.1; it stores the XML data in HDFS 1.1.2.

XML data. We used XMark [21] data; to study queries joining several documents, we used the split option of the XMark generator to create four collections of XML documents, each containing a specific type of XMark subtrees: *users* (10% of the dataset size), *items* (50%), *open auctions* (25%) and *closed auctions* (15%). We used datasets of up to **272GB** as detailed below. All documents are stored into HDFS, which replicates them three times and distributes them across the nodes.

XML queries. We used a subset of XMark queries from our XQuery fragment, and added queries with features from our dialect but absent from the original XMark, e.g. joins on disjunctive predicates.

Table 2 outlines the queries: the collection(s) that each query carries over, the corresponding XML algebraic operators and their numbers of occurrences, and the parallelization contracts used in the plan generated by our translation framework. *Queries q_9-q_{14} all involve value joins, which carry over thousands of documents arbitrarily distributed across the HDFS nodes.*

Table 2: Query details.

Query	Collections	Algebra operators (#)	Parallelization contracts (#)
q_1	*users*	Navigation (1)	Map (1)
q_2	*items*	Navigation (1)	Map (1)
q_3	*items*	Navigation (1)	Map (1)
q_4	*closed auctions*	Navigation (1)	Map (1)
q_5	*closed auctions*	Navigation (1)	Map (1)
q_6	*users*	Navigation (1)	Map (1)
q_7	*closed auctions*	Navigation (1) Aggregation (2)	Map (2) Reduce (1)
q_8	*items*	Navigation (1) Aggregation (2)	Map (2) Reduce (1)
q_9	*users* *closed auctions*	Navigation (2) Projection (1) Group-by/aggregation (1) Conjunctive equi-join (1)	Map (3) Reduce (1) Match (1)
q_{10}	*users* *items* *closed auctions*	Navigation (3) Projection (2) NLO conjunctive equi-join (2)	Map (5) CoGroup (2)
q_{11}	*users*	Navigation (2) Projection (1) Dup. elim. (1) NLO conjunctive equi-join (1)	Map (3) Reduce (1) CoGroup (1)
q_{12}	*users* *closed auctions*	Navigation (2) Projection (1) NLO conjunctive equi-join/ aggregation (1)	Map (3) CoGroup (1)
q_{13}	*users* *closed auctions*	Navigation (2) Projection (1) NLO disjunctive equi-join (1)	Map (3) Reduce (2) CoGroup (2)
q_{14}	*users* *open auctions*	Navigation (2) Projection (1) NLO inequi-join (1)	Map (3) Reduce (2) Cross (1)

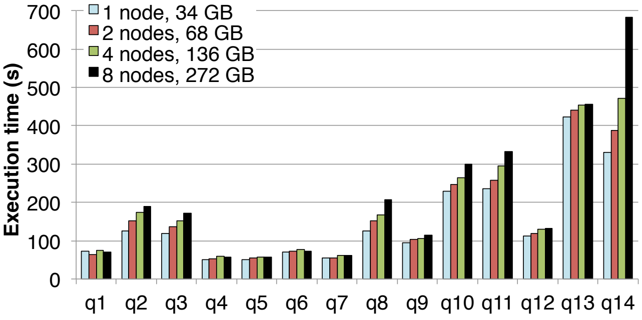


Figure 4: PAXQuery scalability evaluation.

3.1 PAXQuery scalability

Our first goal is to check that PAXQuery brings to XQuery evaluation the desired benefits of implicit parallelism. For this, we fixed a set of queries, generated 11,000 documents (**34GB**) per node, and varied the number of nodes from **1** to **2, 4, 8** respectively; the total dataset size increases accordingly in a linear fashion, up to **272GB**.

Figure 4 shows the response times for each query. Queries q_1-q_6 navigate in the input document according to a given navigation pattern of 5 to 14 nodes. The response time of these queries follows the size of the input, as each of them translates into a Map PACT. In Figure 4 we can see that these queries scale up well, with a moderate overhead as the data volume and number of nodes increases.

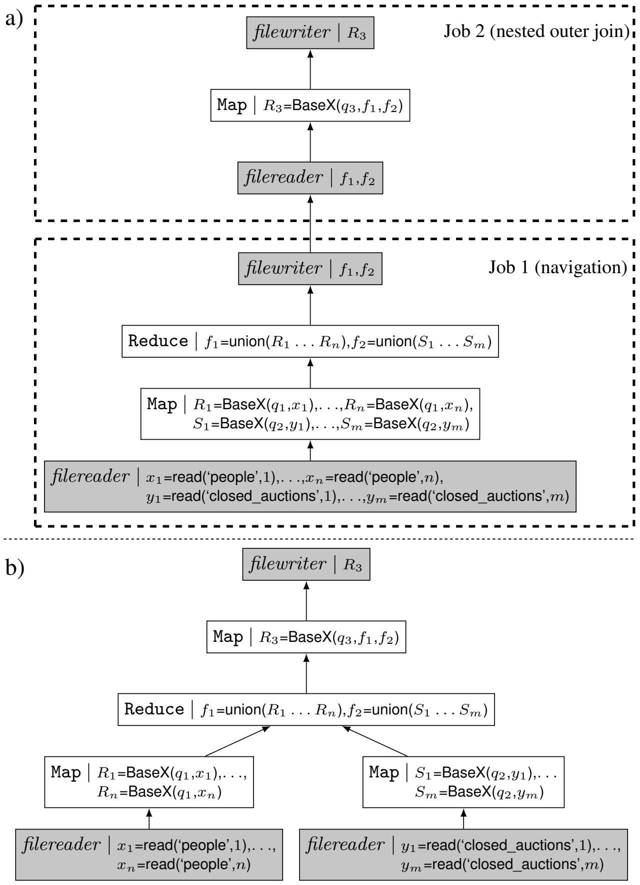

a)
filewriter | R_3
Job 2 (nested outer join)
Map | R_3=BaseX(q_3,f_1,f_2)
filereader | f_1,f_2
filewriter | f_1,f_2
Job 1 (navigation)
Reduce | f_1=union($R_1 \ldots R_n$),f_2=union($S_1 \ldots S_m$)
Map | R_1=BaseX(q_1,x_1),. . .,R_n=BaseX(q_1,x_n), S_1=BaseX(q_2,y_1),. . .,S_m=BaseX(q_2,y_m)
filereader | x_1=read('people',1),. . .,x_n=read('people',n), y_1=read('closed_auctions',1),. . .,y_m=read('closed_auctions',m)
b)
filewriter | R_3
Map | R_3=BaseX(q_3,f_1,f_2)
Reduce | f_1=union($R_1 \ldots R_n$),f_2=union($S_1 \ldots S_m$)
Map | R_1=BaseX(q_1,x_1),. . ., R_n=BaseX(q_1,x_n)
Map | S_1=BaseX(q_2,y_1),. . . S_m=BaseX(q_2,y_m)
filereader | x_1=read('people',1),. . ., x_n=read('people',n)
filereader | y_1=read('closed_auctions',1),. . ., y_m=read('closed_auctions',m)

Figure 5: Execution of XQuery using alternative architectures based on MapReduce (a) and PACT (b) for comparison with PAXQuery.

Queries q_7 and q_8 apply an aggregation over all the records generated by a navigation. For both queries, the navigation generates nested records and the aggregation consists on two steps. The first step goes over the nested fields in each input record, and thus it uses a Map contract. The second step is executed over the results of the first. Therefore, a Reduce contract that groups together all records coming from the previous operator is used. Since the running time is dominated by the Map PACTs which parallelize very well, q_7 and q_8 also scale up well.

Queries q_9-q_{12} involve conjunctive equi-joins over the collections. Query q_{13} executes a NLO disjunctive equi-join, while q_{14} applies a NLO inequi-join. We notice a very good scaleup for q_9-q_{13}, whose joins are translated in many PACTs (recall Table 1). In contrast, q_{14}, which translates into a Cross PACT, scales noticeably less well. *This validates the interest of translating disjunctive equi-joins into many PACTs* (as our rules do), *rather than into a single Cross*, since, despite parallelization, it fundamentally does not scale.

3.2 Comparison against other alternatives

We next compare our system with other *alternatives for implicitly parallel evaluation of XQuery*. As explained in the Introduction, no comparable system is available yet. Therefore, for our comparison, we picked the BaseX 7.7 [2] centralized processor and used Hadoop-MapReduce on one hand, and Flink-PACT on the other hand, to parallelize its execution.

Table 3: Query evaluation time (8 nodes, 272GB).

Query	Evaluation time (seconds)		
	BaseX Hadoop-MR	BaseX Flink-PACT	PAXQuery
q_1	465	**66**	70
q_2	773	282	**189**
q_3	762	243	**172**
q_4	244	72	**58**
q_5	237	72	**57**
q_6	488	**70**	73
q_7	245	74	**62**
q_8	576	237	**206**
q_9	OOM	OOM	**114**
q_{10}	OOM	OOM	**299**
q_{11}	OOM	OOM	**334**
q_{12}	OOM	OOM	**132**
q_{13}	OOM	OOM	**456**
q_{14}	OOM	OOM	**683**

We compare PAXQuery, relying on the XML algebra-to-PACT translation we described, with the following alternative architecture. We deployed BaseX on each node, and parallelized XQuery execution as follows:

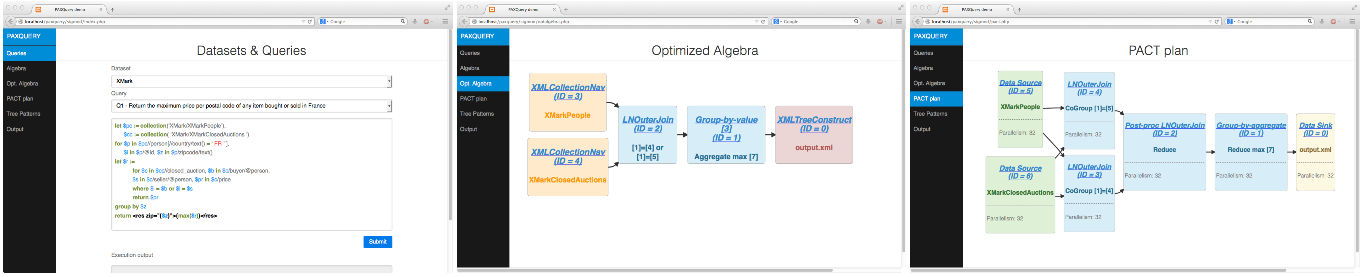
1. Manually decompose each query into a set of leaf subqueries performing just tree pattern navigation, followed by a recomposition subquery which applies (possibly nested, outer) joins over the results of the leaf subqueries;
2. Parallelize the evaluation of the leaf subqueries through one Map over all the documents, followed by one Reduce to union all the results. Moreover, if the recomposition query is not empty, start a new MapReduce job running the recomposition XQuery query over all the results thus obtained, in order to compute complete query results.

This alternative architecture is in-between ChuQL [13], where the query writer *explicitly* controls the choice of Map and Reduce keys, i.e., MapReduce is *visible at the query level*, and PAXQuery where parallelism is completely hidden. In this architecture, q_1-q_8 translate to one Map and one Reduce, whereas q_9-q_{14} feature joins which translates into a recomposition query and thus a second job. As we will illustrate with the following example, the manual decomposition takes a considerable effort.

Example (*continuation*). The MapReduce and PACT plans that execute the XQuery introduced in our example are depicted in Figure 5.

Observe that the MapReduce workflow contains two jobs. The *first job* creates a key/value pair out of each document in each collection, which contains the document's content. The pairs are correspondingly labeled so that they can be identified in the following steps. The Map user function uses BaseX to execute a navigation query on the content of each input pair; q_1 is equivalent to the tree pattern e'_1 in Figure 2.b, while q_2 is equivalent to e'_2. In turn, the Reduce gathers the pairs that originated from the same collection in a group, and applies a user function that unions the results for each of these collections, thus creating files f_1 and f_2 for collections 'people' and 'closed_auctions', respectively. Note that the Reduce operation is necessary because we execute a nested outer join between the results from both collections in the subsequent step. In the absence of our parallelization algorithms, BaseX needs a global view over the results from each collection.

The second job is Map-only. It reads the inputs from the first job, and then it uses BaseX to execute the nested outer join between the inputs. The result is then written to disk.



Datasets & Queries

Dataset: XMark

Query: Q1 - Return the maximum price per postal code of any item bought or sold in France

```
let $pc := collection('XMark/XMarkPeople'),
    $cc := collection( 'XMark/XMarkClosedAuctions ')
for $p in $pc//person[//country/text() = ' FR ' ],
    $i in $p/@id, $z in $p/zipcode/text()
let $r :=
        for $c in $cc//closed_auction, $b in $c/buyer/@person,
        $s in $c/seller/@person, $pr in $c/price
        where $i = $b or $i = $s
        return $pr
group by $z
return <res zip="{$z}">{max($r)}</res>
```

Figure 6: PAXQuery demonstration GUI.

The PACT plan is similar to the previous one, except for the fact that instead of being a linear workflow, it is a DAG of operators.

Table 3 shows the response times when running the query on the 8 nodes and 272GB; the shortest time is shown in bold, while OOM stands for *out of memory*. First, we notice that BaseX runs 2 to 5 times faster on Flink than on Hadoop. This is due to Hadoop's checkpoints (writing intermediary results to disk) while Flink currently does not, trading reliability for speed. For queries without joins (q_1-q_8), PAXQuery is faster for most queries than BaseX on Hadoop or Flink; this simply points out that our in-house tree pattern matching operator (physical implementation of nav) is more efficient than the one of BaseX. Queries with joins (q_9-q_{14}) fail in the competitor architecture. The reason is that intermediary join results grow too large and this leads to an out-of-memory error. PAXQuery evaluates such queries well, based on its massively parallel (outer) joins.

Our experiments demonstrate the efficiency of an XQuery processor built on top of PACT. First, our scalability evaluation has shown that the translation to PACT allows PAXQuery to parallelize every query execution step with no effort required to partition, redistribute data etc., and thus to scale out with the number of machines in a cluster. The only case where scale-up was not so good is q_{14} where we used a Cross (cartesian product) to translate an inequality join; an orthogonal optimization here would be to use a smarter dedicated join operator for such predicates, e.g. [16].

Second, PAXQuery outperformed an alternative architecture in which an XQuery processor runs on each node and Hadoop or Flink/Pact are used to parallelize *XPath navigation only* across the input. In such an architecture, the queries with joins across documents on the data volumes we considered could not complete, highlighting the need for parallel platforms supporting all XQuery processing steps, through an algebraic translation, such as PAXQuery.

4. DEMONSTRATION SCENARIO

For our demonstration, we upload multiple collections of XML datasets [11, 18] to our cluster, and we provide a selection of analytical queries that cover the most important aspects of the translation, e.g. rich tree pattern navigation, nested queries, aggregation, complex joins. The audience may also formulate their own queries.

Then, we showcase the different steps that PAXQuery takes to transform any input XQuery query into a PACT plan. Thus, for a given query, attendees are shown (*i*) the initial algebraic plan resulting from query parsing, (*ii*) an optimized algebraic plan as a result of applying rule-based optimization to the initial plan, (*iii*) the equivalent PACT plan, including the annotations over each PACT operator, and (*iv*) the final XML output. We have implemented a Web client for PAXQuery that visualizes all this information (see Figure 6). Further, we use the Flink monitoring tools to provide real-time information about the parallel execution in the cluster.

5. CONCLUSION

We demonstrate PAXQuery, a system that enables the parallelization of the execution of XML queries over large collections of XML documents. PAXQuery transforms an input XQuery query into an efficient PACT plan whose execution can be easily parallelized by the Flink platform; the feasability and performance improvements of this approach are proven by the experimental results provided. While PAXQuery's implementation is specific to XQuery, the concepts shown in this demonstration are applicable to other programming languages for semi-structured data, e.g. Jaql or JSONiq.

6. REFERENCES

[1] A. Arion, V. Benzaken, I. Manolescu, Y. Papakonstantinou, and R. Vijay. Algebra-Based identification of tree patterns in XQuery. In *FQAS*, 2006.

[2] BaseX. http://basex.org/products/xquery/.

[3] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke. Nephele/PACTs: a programming model and execution framework for web-scale analytical processing. In *SoCC*, 2010.

[4] N. Bidoit, D. Colazzo, N. Malla, F. Ulliana, M. Nolè, and C. Sartiani. Processing XML queries and updates on map/reduce clusters. In *EDBT*, 2013.

[5] P. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner. MonetDB/XQuery: a fast XQuery processor powered by a relational engine. In *SIGMOD*, 2006.

[6] J. Camacho-Rodríguez, D. Colazzo, and I. Manolescu. PAXQuery: Efficient Parallel Processing of Complex XQuery. *IEEE TKDE*, 2015.

[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] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.

[9] A. Deutsch, Y. Papakonstantinou, and Y. Xu. The NEXT Logical Framework for XQuery. In *VLDB*, 2004.

[10] Apache Flink. http://flink.incubator.apache.org/.

[11] GeoNames. http://www.geonames.org/.

[12] F. Hueske, M. Peters, M. J. Sax, A. Rheinländer, R. Bergmann, A. Krettek, and K. Tzoumas. Opening the Black Boxes in Data Flow Optimization. *PVLDB*, 2012.

[13] S. Khatchadourian, M. P. Consens, and J. Siméon. Having a ChuQL at XML on the cloud. In *AMW*, 2011.

[14] I. Manolescu, Y. Papakonstantinou, and V. Vassalos. XML Tuple Algebra. In *Encyclopedia of Database Systems*. 2009.

[15] G. Miklau and D. Suciu. Containment and equivalence for an XPath fragment. In *PODS*, 2002.

[16] A. Okcan and M. Riedewald. Processing theta-joins using MapReduce. In *SIGMOD*, 2011.

[17] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. In *SIGMOD*, 2008.

[18] Open Weather Map. http://openweathermap.org/.

[19] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, Inc., New York, NY, USA, 3 edition, 2003.

[20] C. Re, J. Siméon, and M. F. Fernández. A Complete and Efficient Algebraic Compiler for XQuery. In *ICDE*, 2006.

[21] 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.

[22] Apache Tez. http://tez.apache.org/.

[23] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Anthony, H. Liu, and R. Murthy. Hive - a PB scale data warehouse using Hadoop. In *ICDE*, 2010.

[24] XQuery 3.0: An XML Query Language, April 2014.