PAXQuery: A Massively Parallel XQuery Processor

Jesús Camacho-Rodríguez, Dario Colazzo, Ioana Manolescu

PAXQuery: A Massively Parallel XQuery Processor

Jesús Camacho-Rodríguez
Université Paris-Sud & Inria, France
jesus.camacho-rodriguez@lri.fr

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

Ioana Manolescu
Inria & Université Paris-Sud, France
ioana.manolescu@inria.fr

ABSTRACT

We present a novel approach for parallelizing the execution of queries over XML documents, implemented within our system PAXQuery. We compile a rich subset of XQuery into plans expressed in the PArallelization ConTracts (PACT) programming model. These plans are then optimized and executed in parallel by the Stratosphere system. We demonstrate the efficiency and scalability of our approach through experiments on hundreds of GB of XML data.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems

General Terms

Design, Performance, Experimentation

1. INTRODUCTION

Increasingly large data volumes have lead to the apparition of massively parallel processing frameworks, such as MapReduce [7]. Its main advantage is to simplify parallel data processing and handle task allocation and fault tolerance transparently from the application.

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, more powerful abstractions have appeared to express massively parallel complex data processing, such as the *Parallelization Contracts* programming model [2] (or **PACT**, in short).

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. The PACT model is part of the open-source **Stratosphere** platform [18].

*This work was partially done when the author was with Université Paris-Sud and Inria.

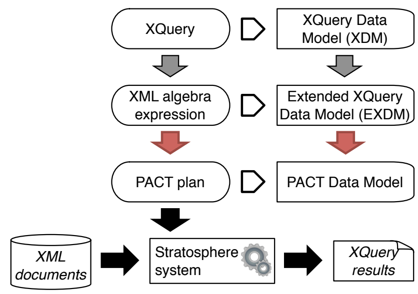

Figure 1: PAXQuery architecture overview.

In this work, we present PAXQuery, a massively parallel XQuery processor. 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. Inspired by other high-level data analytics languages that are compiled into parallel frameworks [3, 10, 16], PAXQuery translates XML queries 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, 12]. Further, we can rely on the Stratosphere platform for the optimization of the PACT plan and its automatic transformation into a data flow that is evaluated in parallel on top of the Hadoop Distributed File System (HDFS); these steps are explained in [2].

In the sequel, Section 2 describes PAXQuery architecture and main features in detail, and provides a beginning-to-end query translation example. Section 3 describes the experimental evaluation of our system. Section 4 discusses related work and Section 5 concludes.

2. PAXQUERY ARCHITECTURE

Our approach for implicit parallel evaluation of XML queries is to *translate* them into PACT plans as depicted in Figure 1. The central vertical stack traces the query translation steps from the top to the bottom, while at the right of each step we show the data models manipulated by that step. We present each step of the translation below.

2.1 Query language

PAXQuery supports a representative subset of XQuery [19], 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.

Example. The following XQuery *extracts the name of users, and the items of their auctions (if any)*:

```
let $pc := collection('people'),
    $cc := collection('closed_auctions')
for $p in $pc/site/people/person, $i in $p/@id
let $n := $p/name
let $r :=
  for $c in $cc//closed_auction, $b in $c/buyer/@person,
      $s in $c/seller/@person
  let $a := $c/itemref
  where $i = $b or $i = $s
  return $a
return <res>{$n,$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*; we support the XPath$^{\{/,//,[]\}}$ fragment [15]. In addition, our subset supports rich predicates expressed in disjunctive normal form (DNF), supporting value- and node identity-based comparisons.

2.2 XML query algebra

Our approach is based on the representation of the XQuery query as an *equivalent algebraic expression*, on which multiple optimizations can be applied. XQuery translation into different algebra formalisms and the optimization of resulting expressions have been extensively studied [5, 8, 17].

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 optimizer, as straightforward translation into nested plans leads to very poor performance. Effective optimization techniques translate nested XQuery into *unnested plans relying on joining and grouping* [8, 13, 14]. Depending on the query shape, such *decorrelating* joins may be *nested* and/or *outer* joins.

PAXQuery uses the algebra presented in [13], and translation from XQuery to this algebra is outlined in [1]. However, we can easily adapt to any XML query algebra used by existing engines that satisfies the following two assumptions: (*i*) the algebra is tuple-oriented (potentially using nested tuples), and (*ii*) the algebra is rich enough to support decorrelated (unnested) plans even for nested XQuery; in particular we consider that the query plan has been unnested [14] before we start translating it into PACT.

2.3 Algebraic representation of XQuery

We introduce our algebraic representation of XQuery by the following example (see [13] for details):

Example (*continuation*). The algebra plan for the XQuery introduced above is shown in Figure 2. The schemas of the tuples produced by each operator are denoted by S_i. We discuss the operators starting from the leaves.

The XML *scan* operators take as input the 'people' (respectively 'closed_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

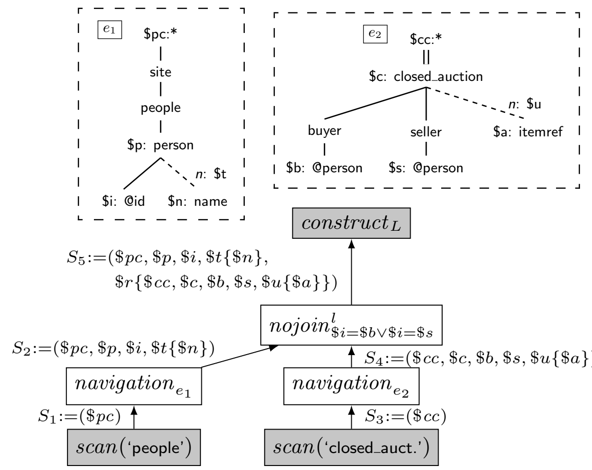

Figure 2: Logical plan for the example query.

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. The algebra we consider allows for *consolidating as many navigation operations from the same query as possible within a single navigation tree pattern*, and in particular *navigation performed outside of the* `for` *clauses*. Large navigation patterns lead to more efficient query execution, since patterns can be matched very efficiently against XML documents [6]. 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.

Consider the ETP e_1 in Figure 2. The operator $navigation_{e_1}$ concatenates each input tuple successively with all `@id` attributes (variable `$i`) resulting from the embeddings of e_1 in the value bound to `$pc`. The node labeled $n:name is (*i*) *optional* and (*ii*) *nested* with respect to its parent node $p:person, since by XQuery semantics: (*i*) if a given `$p` lacks a name, `$p` still contributes to the query result; (*ii*) all names for a given `$p` are bound into a single sequence. Observe that all `name` elements (variable `$n`) are *nested* into variable `$t`, which did not appear in the original query; in fact, `$t` is created by the XQuery to algebra translation to hold the nested collection with all values bound to `$n`. The operator $navigation_{e_2}$ is generated in a similar fashion. Therefore, in the previous query, ETPs e_1 and e_2 correspond to the following fragment:

```
for $p in $pc/site/people/person, $i in $p/@id
let $n := $p/name
let $r :=
  for $c in $cc//closed_auction, $b in $c/buyer/@person,
      $s in $c/seller/@person
  let $a := $c/itemref
```

Above the *navigation* operators in Figure 2, we have a nested left outer join ($nojoin^l_\rho$) on a disjunctive predicate ρ, which brings together people and the auctions they participated in, either as buyers *or* sellers. Observe that as the join is *outer*, all people are kept in the output, even if they did not participate in any auction.

Finally, the XML construction ($construct_L$) 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 attached to the operator. In our example, L contains a single pattern that generates for each tuple an XML tree consisting of elements of the form `<res>{$n,$r}</res>`.

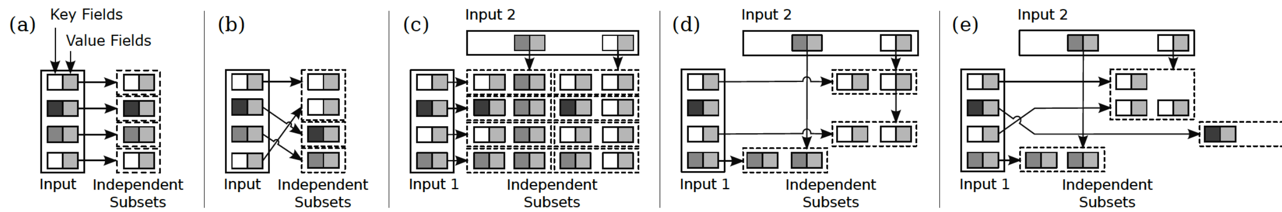

Figure 3: (a) `Map`, (b) `Reduce`, (c) `Cross`, (d) `Match`, and (e) `CoGroup` parallelization contracts.[1]

2.4 PACT model

The PACT model [2] is a generalization of MapReduce, based on the concept of parallel data processing operators. PACT plans are DAGs of *implicit parallel operators*, that are optimized and translated into *explicit parallel data flows* by Stratosphere.

Data model. PACT plans manipulate *nested 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. 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. Each of the key fields must be an atomic value. The *key of a record* r is the concatenation 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; a function parameterizing the data source specifies how to structure the data into records. In turn, data is output into 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*; and (*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).

Although the PACT model allows creating custom parallelization contracts, a set of them for the most common cases is built-in: `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 key value. The `Cross`, `Match`, and `CoGroup` contracts are used to define binary operators. The `Cross` contract forms a group from every pair of records in its two inputs (essentially, it produces 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 depending on the key value of the record.

2.5 From algebra expressions to PACT plans

We describe now how PAXQuery translates XML algebra expressions into PACT plans. First, out of nested tuples containing XML data instances (trees with identity), we create PACT nested records. Second, we translate XQuery algebraic expressions into PACT plans. Details about the former are omitted in this presentation, for space reason, while we focus in the latter.

[1]Figure reproduced from [11].

Table 1: Algebra to PACT overview.

Algebra operators		PACT operators (#)
Scan		`Source` (1)
Construct		`Sink` (1)
Navigation		`Map` (1)
Group-by		`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	`CoGroup` (1)
Disjunctive equi-join (n conjunctions)	Inner	`Match` (n)
	Outer	`CoGroup` (n) & `Reduce` (1)
	Nested outer	`CoGroup` (n) & `Reduce` (1)
Theta-join	Inner	`Cross` (1)
	Outer	`Cross` (1) & `Reduce` (1)
	Nested outer	`Cross` (1) & `Reduce` (1)

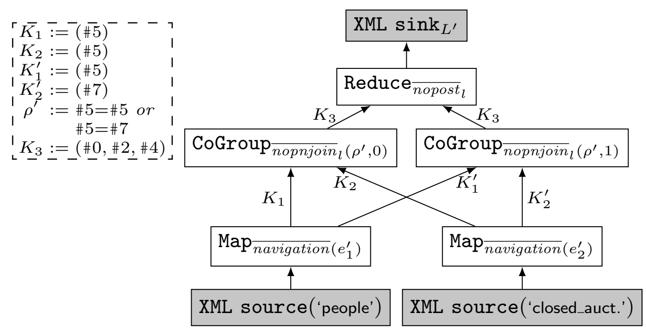

Figure 4: PACT plan corresponding to the logical expression in Figure 2.

Table 1 depicts the supported algebra operators and the contracts used by the PACT operators resulting from our translation. First, observe that the *scan*, respectively *construct*, functionality is integrated into a `source`, respectively `sink`, in the PACT plan. In turn, unary operators use `Map` and `Reduce` contracts depending on their semantics; the implementation of their UFs is in most of the cases straightforward. Finally, the translation of the binary operators is more complex, as they have 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 this more elaborated translation with the following example.

Example (*continuation*). Consider the algebra plan in Figure 2. PAXQuery generates the PACT program shown in Figure 4; each K_i contains the positions of the key fields in records. The `XML source` operators scan (in parallel) the respective collections and transform each document into a

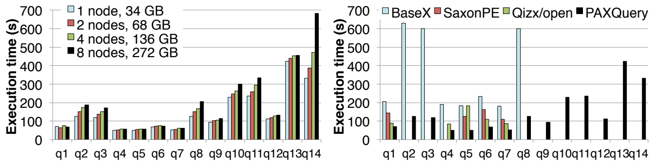

Figure 5: PAXQuery scalability evaluation (left) and comparison with centralized processors (right).

record. Next, the Map operators apply the $\overline{navigation}$ UF in parallel on the records thus obtained, following the query's XPath expressions. 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 $\overline{nopnjoin_l}(\rho', k)$ 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 $\overline{nopost_l}$ UF associated to it is applied to produce the final result for the join. Finally, the XML sink builds and returns XML results.

Clearly, complex joins such as the one contained in the example could be translated using a single Cross operator instead of multiple CoGroup. However, this would be less efficient and scale poorly (number of comparisons quadratic in the input size), as our experiments demonstrate.

2.6 Optimization and execution by Stratosphere

In the last step, PAXQuery sends the PACT plan to the Stratosphere platform, which optimizes it, and turns it into a data flow that is evaluated in parallel, as explained in [2].

3. IMPLEMENTATION AND EXPERIMENTS

PAXQuery has been implemented in Java 1.6, and it relies on the Stratosphere platform [18] for the execution. We present here some experimental results we obtained with it.

PAXQuery scalability was studied by fixing a set of 14 queries, generating 11.000 documents (34GB) per node, and varying 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 5 (left) shows the response times for each query. Our results show 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 to translate an inequality join; this highlights the interest of the more efficient operators (especially CoGroup) used in the other plans.

Figure 5 compares PAXQuery with XQuery processors in a centralized setting (one single node, 34 GB of data). None of the competing processors was able to evaluate any of our queries with joins across documents (q_9-q_{14}), as they run out of memory or out of time, highlighting the need for efficient parallel platforms for evaluating such queries on such large document collections.

4. RELATED WORK

MRQL [9] proposes a simple SQL-like XML query language implemented through a few operators directly compilable into MapReduce. MRQL queries may be nested, however, its dialect does not allow for expressing the rich join flavours that we use. Further, the XML navigation supported by MRQL is limited to XPath, in contrast to our richer navigation based on tree patterns with multiple returning nodes, and nested and optional edges. ChuQL [12] is an XQuery extension that exposes the MapReduce framework to the developer in order to distribute computations among XQuery engines; this leaves the parallelization work to the programmer, in contrast with our implicit query parallelization approach.

5. CONCLUSION AND FUTURE WORK

We have presented the PAXQuery approach for the implicit parallelization of XQuery through translation to PACT. Our experiments demonstrate the efficiency and scalability of PAXQuery. It is our plan to open-source the system in the near future. In the future, we contemplate the integration of indexing techniques into PAXQuery, and reusing intermediary results in the PACT framework to enable efficient multiple-query processing.

Acknowledgements. This work has been partially funded by the KIC EIT ICT Labs activity 12115.

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] 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.

[3] K. S. Beyer, V. Ercegovac, R. Gemulla, A. Balmin, M. Y. Eltabakh, C.-C. Kanne, F. Özcan, and E. J. Shekita. Jaql: A Scripting Language for Large Scale Semistructured Data Analysis. *PVLDB*, 2011.

[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] Y. Chen, S. B. Davidson, and Y. Zheng. An efficient XPath query processor for XML streams. In *ICDE*, 2006.

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

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

[9] L. Fegaras, C. Li, U. Gupta, and J. Philip. XML Query Optimization in Map-Reduce. In *WebDB*, 2011.

[10] A. Heise, A. Rheinländer, M. Leich, U. Leser, and F. Naumann. Meteor/Sopremo: An Extensible Query Language and Operator Model . In *BIGDATA*, 2012.

[11] 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.

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

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

[14] N. May, S. Helmer, and G. Moerkotte. Strategies for query unnesting in XML databases. *TODS*, 2006.

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

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

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

[18] Stratosphere Platform. http://www.stratosphere.eu/.

[19] XQuery 3.0: An XML Query Language, October 2013.