PAXQuery: Efficient Parallel Processing of Complex XQuery

Jesús Camacho-Rodríguez, Dario Colazzo, Ioana Manolescu

PAXQuery: Efficient Parallel Processing of Complex XQuery

Jesús Camacho-Rodríguez, Dario Colazzo, and Ioana Manolescu

Abstract—Increasing volumes of data are being produced and exchanged over the Web, in particular in tree-structured formats such as XML or JSON. This leads to a need of highly scalable algorithms and tools for processing such data, capable to take advantage of massively parallel processing platforms.
This work considers the problem of efficiently parallelizing the execution of complex nested data processing, expressed in XQuery. We provide novel algorithms showing how to translate such queries into PACT, a recent framework generalizing MapReduce in particular by supporting many-input tasks. We present the first formal translation of complex XQuery algebraic expressions into PACT plans, and demonstrate experimentally the efficiency and scalability of our approach.

Index Terms—XQuery processing, XQuery parallelization, XML data management.

1 Introduction

To scale data processing up to very large data volumes, platforms are increasingly relying on *implicit parallel* frameworks [8], [19], [48]. The main advantage of using such frameworks is that processing is distributed across many sites without the application having to explicitly handle data fragmentation, fragment placement etc.

By far the most widely adopted framework, MapReduce [19] features a very simple processing model consisting of two operations, *Map* which distributes processing over sets of (key, value) pairs, and *Reduce* which processes the sets of results computed by *Map* for each distinct key. However, the simplicity of this processing model makes complex computations hard to express. Therefore, high-level data analytics languages such as Pig [36], Hive [45] or Jaql [11], that are translated (compiled) into MapReduce programs, have emerged. Still, complex processing translates to large and complex MapReduce programs, which may miss parallelization opportunities and thus execute inefficiently.

Recently, more powerful abstractions for implicitly parallel data processing have emerged, such as the *Resilient Distributed Datasets* [48] or *Parallelization Contracts* [8] (**PACT**, in short). In particular, 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 lies at the core of the **Stratosphere** platform [44], which can read data from and write data to the Hadoop Distributed File System (HDFS) [3].

- *Jesús Camacho-Rodríguez and Ioana Manolescu are with Inria and Université Paris-Sud. Bâtiment 650 (PCRI), 91405 Orsay Cedex, France. E-mail: jesus.camacho_rodriguez@inria.fr, ioana.manolescu@inria.fr.*
- *Dario Colazzo is with Université Paris-Dauphine. Place du Marchal de Lattre de Tassigny, 75775 Paris Cedex, France. E-mail: dario.colazzo@dauphine.fr.*

In this work, we are interested in the *implicit parallelization of XQuery* [40], the W3C's standard query language for XML data. The language has been recently enhanced with features geared towards XML analytics [21], such as explicit grouping. Given a very large collection of documents, evaluating an XQuery query that *navigates over these documents and also joins results from different documents* raises performance challenges, which may be addressed by parallelism. In contrast with prior work [12], [18], [27], we are interested in *implicit parallelism*, which does not require the application (or the user) to partition the XML input nor the query across many nodes.

The contributions of this work are the following:

1) We present a novel *methodology for massively parallel evaluation of XQuery*, based on PACT and previous research in algebraic XQuery optimization.
2) We provide a *translation algorithm* from the algebraic operators required by a large powerful fragment of XQuery into operators of the PACT parallel framework. This enables parallel XQuery evaluation *without requiring data or query partitioning effort from the application*.
   Toward this goal, we first translate XML data instances (trees with identity) into PACT nested records, to ensure XML query results are returned after the PACT manipulations of nested records.
   Second, we bridge the gap between the XQuery algebra, and in particular, many flavors of joins [20], [31], [32] going beyond simple conjunctive equality joins, and PACT operators which (like MapReduce) are fundamentally designed around the equality of key values in their inputs.
   Our translation of complex joins into PACT is of interest beyond the XQuery context, as it may enable compiling other high-level languages [11], [36], [45] into PACT to take advantage of its efficiency.
3) We fully implemented our translation technique into our

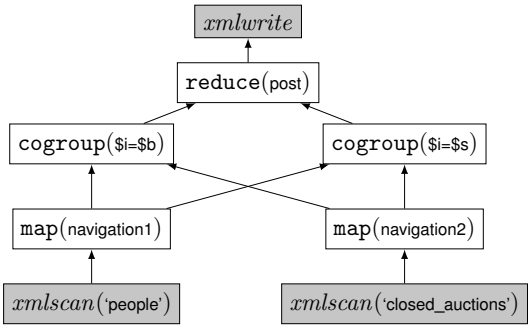


Fig. 1. Outline of the PACT program generated by PAXQuery for the XQuery in Example 1.

PAXQuery platform. We present experiments demonstrating that our translation approach (i) effectively parallelizes XQuery evaluation taking advantage of the PACT framework, and (ii) scales well beyond alternative approaches for implicitly parallel XQuery evaluation, in particular as soon as joins across documents are present in the workload.

The remainder of the paper is organized as follows. Section 2 introduces the problem by means of an example. Section 3 provides background on XML, XQuery, and the PACT model. Section 4 overviews our complete solution and characterizes the XQuery algebras targeted by our translation. Section 5 presents the translation algorithm from XQuery plans to PACT, at the core of this work. Section 6 describes our experimental evaluation. Section 7 discusses related work and then we conclude.

2 Motivation

Example 1. Consider the following XQuery that 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>
```

◇

We would like to evaluate this query over two large collections of documents (concerning people, respectively closed auctions) stored in HDFS. Evaluating the query in a massively parallel fashion as previously proposed, e.g., in [27] requires the programmer to explicitly insert parallelization primitives in the query, which requires time and advanced expertise. Alternatively, one could partition the XML data, as in [12], [18], and run the query as such. This also requires human input (potentially different for each query); moreover, for complex XQuery queries like the one in Example 1, it also requires manual decomposition of the query into (i) "embarrassingly parallel" subqueries which can be directly run in parallel over many documents, and (ii) a "recomposition" query that applies the remaining query operations.

In contrast, given this query, PAXQuery generates *in a fully automated fashion* the PACT program shown in Figure 1. We outline here its functioning while on purpose omitting details, which will be introduced later on. The $xmlscan$('people') and $xmlscan$('closed_auctions') operators scan (in parallel) the respective collections and transform each document into a record. Next, the `map` operators navigate in parallel within the records thus obtained, following the query's XPath expressions, and bind the query variables. The next operators in the PACT plan (`cogroup`) go beyond MapReduce. In a nutshell, a `cogroup` can be seen as a `reduce` operator on multiple inputs: it groups together records from all inputs sharing the same key value, and then it applies a user-defined function on each group. In this example, the functions are actually quite complex (we explain them in Section 5). The difficulty they have to solve is to correctly express (i) the *disjunction* in the where clause of the query, and (ii) the *outerjoin* semantics frequent in XQuery: in this example, a `<res>` element must be output even for people with no auctions. The output of both `cogroup` operators is received by the `reduce`, which builds join results between people and closed_auctions, while the last $xmlstore$ builds and returns XML results.

This approach enables us to take advantage of the Stratosphere platform [44] in order to automatically parallelize complex XML processing, expressed in a rich dialect of XQuery. In contrast, *state-of-the-art solutions require partitioning, among nodes and by hand, the query and/or the data*. Moreover, using PACT gives PAXQuery a performance advantage over MapReduce-based systems, because PACT's more expressive massively parallel operators allow more efficient query implementations.

3 Background

In the following, we provide background on the XML data model and XQuery dialect we target (Section 3.1), and the PACT programming model used by Stratosphere (Section 3.2).

3.1 XML and XQuery fragment

XML data. We view XML data as a forest of ordered, node-labeled, unranked trees, as outlined by the simple grammar:

Tree	d	::=	$\underline{s_i} \mid l_i[f]$
Forest	f	::=	$() \mid f,f \mid d$

A tree d is either a text node ($\underline{s_i}$), or an element node having the label l_i and a forest of children; in accordance with the W3C's XML data model, each node is endowed with a unique identity, which we materialize through the $_i$ index. A forest f is a sequence of XML trees; () denotes the empty forest. For the sake of presentation we omitted attributes in our grammar.

XQuery dialect. We consider a representative subset of the XQuery 3.0 language [40]. Our goal was to cover (i) the main navigating features of XQuery, and (ii) key constructs to express analytical style queries, e.g., aggregation, explicit grouping, or rich comparison predicates. However, extensions to support other XQuery constructs, e.g., *if* or *switch* expressions, can be integrated into our proposal in a straightforward manner. The full presentation of our XQuery dialect, including the grammar, can be found in Appendix A.

Q_1
```
let $ic := collection('items')
let $i := $ic/site/regions//item
return count($i)
```

Q_2
```
let $ic := collection('items')
for $i in $ic/site/regions//item
let $l := $i/location/text()
group by $l
return <res><name>{$l}</name>
             <num>{count($i)}</num></res>
```

Q_3
```
let $pc := collection('people'),
    $cc := collection('closed_auctions')
for $p in $pc/site/people/person, $i in $p/@id
let $n := $p/name/text()
let $a :=
  for $t in $cc/site/closed_auctions/closed_auction,
      $b in $t/buyer/@person
  where $b = $i
  return $t
return <item person="{$n}">{count($a)}</item>
```

Fig. 2. Sample queries expressed in our XQuery grammar.

Figure 2 provides three sample queries. A path starts from the root of each document in a *collection* found at URI *Uri*, or from the root of one *document* at URI *Uri*, or from the bindings of a previously introduced *variable*. The path expression dialect *Path* belongs to the XPath$^{\{/,//,[]\}}$ language [34]. We support two different types of comparators in predicates: (*ValCmp*) to compare atomic values, and (*NodeCmp*) to compare nodes by their identity. Finally, the *group by* clause groups tuples based on variable values.

In Figure 2, queries Q_1 and Q_2 use only one collection of documents while query Q_3 joins two collections. Further, Q_2 and Q_3 construct new XML elements while Q_1 returns the result of an aggregation over nodes from the input documents.

3.2 PACT framework

The PACT model [8] 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.

We introduce below the PACT data model and formalize the semantics of its operators.

Data model. PACT plans manipulate *record*s of the form:

$$r = ((f_1, f_2, \ldots, f_n), (i_1, i_2, \ldots, i_k))$$

where $1 \leq k \leq n$ and:

- $(f_1, f_2, \ldots, f_n)$ is an ordered sequence of *fields* f_i. In turn, a field f_i is either an atomic value (string) or an ordered sequence $(r'_1, \ldots, r'_m)$ of records.
- $(i_1, i_2, \ldots, i_k)$ 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 $f_{i_1}, f_{i_2}, \ldots, f_{i_k}$. We denote by $r[i]$ and $r.key$ the field i and the key of record r, respectively. A $\perp$-record is a record whose fields consist of *null* ($\perp$) values. Finally, $\mathcal{R}$ denotes the infinite domain of records.

Path indexes are needed to describe navigation through records. A path index pi obeys the grammar $pi := j.pi \mid \epsilon$, with $j \geq 0$. *Navigation through* r *along a path index* $j.pi$ first selects $r[j]$. If pi is empty (ϵ), then we are at the target field.

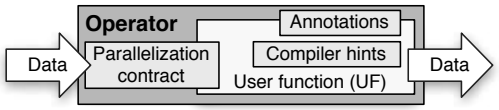


Fig. 3. PACT operator outline.

Otherwise, if $r[j]$ is a list of records (the field at position j is nested), pi navigation is performed on each record.

Data sources and sinks are, respectively, the starting and terminal nodes of a PACT plan. The input data is stored in files; the function parameterizing data source operators 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.

Semantics. Operators are data processing nodes in a PACT plan. Each operator manipulates bags of records; we write $\{\{r_1, r_2, \ldots, r_n\}\}$ to indicate a bag of n records. From now on, for simplicity, we will call a PACT operator simply a PACT, whenever this does not cause confusion. As Figure 3 shows, a PACT consists of (i) a *parallelization contract*, (ii) a *user function* (UF in short) and (iii) optional *annotations* and *compiler hints* characterizing the UF behaviour. We describe these next.

1) **Parallelization contract.** A PACT can have $k \geq 1$ inputs, each of which is a finite bag of records. The contract determines how input records are organized into *groups*.
2) **User function.** The UF is executed independently over each bag of records created by the parallelization contract, therefore these executions can take place in parallel. For each input bag of records, the UF returns a bag of records.
3) Annotations and/or compiler hints may be used to enable optimizations (with no impact on the semantics), thus we do not discuss them further.

The semantics of the PACT op given as input k bags of records $I_1, \ldots, I_k$, with $I_i \subset \mathcal{R}$, $1 \leq i \leq k$, and having the parallelization contract c and the user function f is:

$$op(I_1, \ldots, I_k) = \bigcup_{s \in c(I_1, \ldots, I_k)} f(s)$$

In the above, c builds bags of records by grouping the input records belonging to bags $I_1, \ldots, I_k$; f is invoked on each bag produced by c, and the resulting bags are unioned.

Predefined contracts. Although the PACT model allows creating custom parallelization contracts, a set of them for the most common cases is built-in:

- *Map* has a single input, and builds a singleton for each input record. Formally, given the bag $I_1 \subset \mathcal{R}$ of records, Map is defined as:
$$c_{\texttt{map}}(I_1) = \{\{\{r\} \mid r \in I_1\}\}$$
- *Reduce* also has a single input and groups together all records that share the same key. Given a bag of input records I_1:
$$c_{\texttt{rd}}(I_1) = \{s = \{\{r_1, r_2, \ldots, r_m\}\} \mid r_1, r_2, \ldots, r_m \in I_1 \text{ and } r_1.key = r_2.key = \ldots = r_m.key \text{ and } \nexists r' \in I \setminus s \text{ such that } (r'.key = r_1.key)\}$$
- *Cross* builds the cartesian product of two inputs.
- *Match* builds all pairs of records from its two inputs, which share the same key. Thus, given $I_1, I_2 \subset \mathcal{R}$:

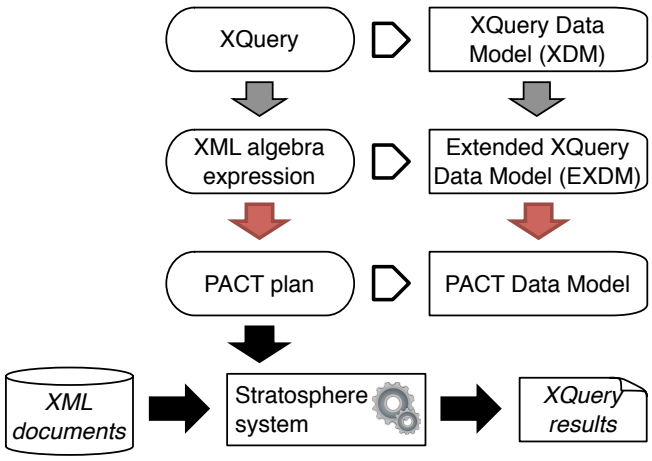


Fig. 4. Translation process overview.

$$c_{\texttt{mt}}(I_1, I_2) = \{\{(r_1, r_2) \mid r_1 \in I_1, r_2 \in I_2 \text{ and } r_1.key = r_2.key\}\}$$

- *CoGroup* can be seen as a "Reduce on two inputs"; it groups the records from both inputs, sharing the same key value. Formally, given $I_1, I_2 \subset \mathcal{R}$:

$$c_{\texttt{cg}}(I_1, I_2) = \{s = \{\{r_{11}, \ldots, r_{1m}, r_{21}, \ldots, r_{2j}\}\} \mid r_{11}, \ldots, r_{1m} \in I_1 \text{ and } r_{21}, \ldots, r_{2j} \in I_2 \text{ and } \forall r, r' \in s : r.key = r'.key \text{ and } \nexists r'' \in (I_1 \cup I_2) \setminus s \text{ such that } r''.key = r_{11}.key\}$$

4 Outline

Our approach for implicit parallel XQuery evaluation is to *translate* XQuery into PACT plans as depicted in Figure 4. 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.

First, the XQuery query is represented as an *algebraic expression*, on which multiple optimizations can be applied. XQuery translation into different algebra formalisms and the subsequent optimization of resulting expressions have been extensively studied [9], [14], [39], [49]. In Section 4.1, we characterize the class of XML algebras over which our translation technique can be applied, while we present the nested-tuple data model and algebra used by our work in Section 4.2.

Second, *the XQuery logical expression is translated into a PACT plan*; we explain this step in detail in Section 5.

Finally, the Stratosphere platform receives the PACT plan, optimizes it, and turns it into a data flow that is evaluated in parallel; these steps are explained in [8].

4.1 Assumptions on the XQuery algebra

Numerous logical algebras have been proposed for XQuery [9], [20], [31], [39]. While the language has a functional flavor, most algebras decompose the processing of a query into *operators*, such as: *navigation* (or *tree pattern matching*), which given a path (or tree pattern) query, extracts from a document *tuples* of nodes matching it; *selection*; *projection*; *join* etc.

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. For instance, in Figure 2, Q_3 contains a nested subquery `for $t ... return $t` (shown indented in the figure); let us call it Q_4 and write $Q_3 = e(Q_4)$. A naïve algebraic expression of such a query would evaluate Q_4 once per result of e in order to compute Q_3 results, which is typically inefficient.

Efficient optimization techniques translate nested XQuery into *unnested plans relying on joining and grouping* [20], [31], [32]. Thus, a smarter method to represent such query is to connect the sub-plans of Q_4 and e with a *join* in the plan of Q_3; the join condition in this example is `$b=$i`. Depending on the query shape, such *decorrelating* joins may be *nested* and/or *outer*.

Our goal is to complement existing engines, which translate from XQuery to an internal algebra, by an efficient compilation of this algebra into an implicit parallel framework such as PACT. This enables plugging a highly parallel back-end to an XQuery engine to improve its scalability. Accordingly, we aim to adapt to any XML query algebra satisfying the following two assumptions:

- The algebra is tuple-oriented (potentially using nested tuples).
- 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 before we start translating it into PACT.

Three observations are in order here.

First, to express complex queries without nesting, the algebra may include *any type of joins (conjunctive/disjunctive, value or identity-based, possibly nested, possibly outer), as well as grouping*; accordingly, we must be able to translate all such operators into PACT.

Second, a tuple-based algebra for XQuery provides *border* operators for (i) creating tuples from XML trees, in leaf operators of the algebraic plan; (ii) constructing XML trees out of tuples, at the top of the algebraic plan, so that XML results can be returned.

Finally, we require no optimization but unnesting [32] to be applied on the XML algebraic plan before translating it to PACT; however, any optimization *may* be applied before (and orthogonal to) our translation.

4.2 Algebra and data model

In the sequel, we present our work based on the algebra in [31]. We describe the nested tuple data model manipulated by this algebra, then present its operators.

Nested tuples data model for XML. The data model extends the W3C's XPath/XQuery data model with *nested tuples* to facilitate describing algebraic operations.

Formally, a tuple t is a list of *variable-value* pairs:

$$((\$V_1, v_1), (\$V_2, v_2), \ldots, (\$V_k, v_k))$$

where the variable names $\$V_i$ are all distinct, and each value v_i is either (i) an *item*, which can be an XML node, atomic value or $\perp$, or (ii) a *homogeneous* collection of tuples (see below).

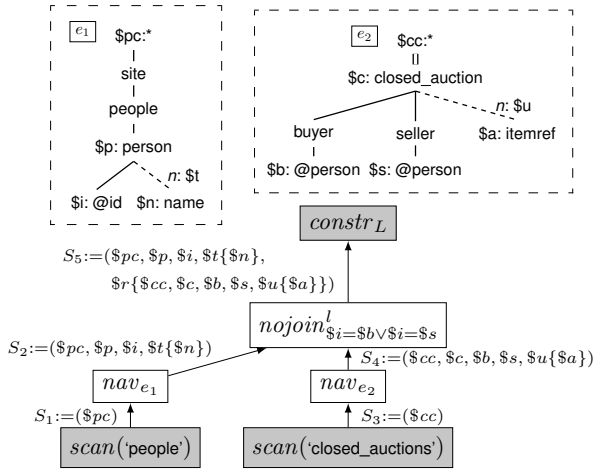


Fig. 5. Sample logical plan for the query in Example 1.

Three flavours of *collections* are considered, namely: *lists*, *bags* and *sets*, denoted as $(t_1, t_2, \ldots, t_n)$, $\{\{t_1, t_2, \ldots, t_n\}\}$, and $\{t_1, t_2, \ldots, t_n\}$, respectively.

Tuple schemas are needed for our discussion. The schema S of a tuple t is a set of pairs $\{(\$V_1, S_1), \ldots, (\$V_n, S_n)\}$ where each S_i is the schema of the value of the variable $\$V_i$. We use `val` to denote the type of (any) atomic value, and `node` to denote an XML node type. Further, a collection of values has the schema $C\{S\}$ where C is `list`, `bag`, or `set`, depending on the kind of collection, and S is the schema of all values in the collection i.e., only *homogeneous* collections are considered.

The *concatenation* of two tuples t_1 and t_2 is denoted by $t_1 + t_2$.

Algebraic representation of XQuery. In the following, we introduce the translation process and the main operators by example. A methodology for translating our XQuery dialect into the algebra we consider was described in [6], and detailed through examples in [30]. The complete list of algebra operators and their semantics can be found in Appendix B.

Example 1 (*continuation*). The algebraic plan corresponding to the XQuery introduced in Section 2 is shown in Figure 5. For simplicity, we omit the variable types in the operators schema and only show the variable names. We discuss the operators starting from the leaves.

The XML *scan* operators take as input the 'people' (respectively 'closed_auctions') XML forests and create a tuple out of each tree in them. XML scan is one of the *border* operators.

XPath and 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. The algebra we consider allows to *consolidate 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* [6], [20], [33]. Large navigation patterns lead to more efficient query execution, since patterns can be matched very efficiently against XML documents; for instance, if the pattern only uses *child and descendant* edges, it can be matched in a single pass over the input [16]. In the spirit of *generalized tree patterns* [17], *annotated tree patterns* [37], or *XML access modules* [5], we assume a navigation (nav) 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 5. 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, it will still contribute to the query result; (ii) if a given `$p` has several names, `let` binds them all into a single node collection. The operator nav_{e_1} concatenates each tuple successively with all `@id` attributes (variable `$i`) and `name` elements (variable `$n`) resulting from the embeddings of e_1 in the value bound to `$pc`. Observe that variable `$n` is *nested* into variable `$t`, which did not appear in the original query; in fact, `$t` is created by the XQuery to algebra translation to hold the nested collection with values bound to `$n`. The operator nav_{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 nav operators in Figure 5, we find a nested join ($nojoin^l_\rho$) on a disjunctive predicate ρ, which selects those people that appear as buyers *or* sellers in an auction.

Finally, the XML construction ($constr_L$) is the *border* operator responsible for transforming a collection of tuples to XML forests [23], [43]. The information on how to build the XML forest is specified by a list L of *construction tree patterns* (CTPs in short), attached to the $constr$ operator. *For each tuple* in its input, $constr_L$ builds *one XML tree for each CTP in* L [31]. In our example, L contains a single CTP that generates for each tuple an XML tree consisting of elements of the form `<res>{$n,$r}</res>`. We omit further details here; the interested reader may find them in Appendix B. ◇

Full operator set. We briefly comment below on the rest of operators that are handled by our translation.

The rest of unary operators are very close to their known counterparts in nested relational algebra. These are *flatten* ($flat_p$) which unnests tuples, *selection* (sel_ρ) based on a predicate ρ, *projection* ($proj_V$), *aggregation* ($agg_{p,a,\$r}$) computing the usual aggregates over (nested) records, and value-based *duplicate elimination* ($dupelim_V$). One operator that is slightly different is group-by ($grp_{G_{id},G_v,\$r}$). In order to conform to XML semantics, the operator may group by identity based on the variables in G_{id}, and/or by value on the variables in G_v [20], [31].

Binary operators include the usual cartesian product ($prod$), join ($join_\rho$), outer join ($ojoin^l_\rho$) and nested outer join ($nojoin^l_\rho$).

$$\frac{v_i \to r_i \qquad i = 1 \ldots n}{((\$V_1, v_1), \ldots (\$V_n, v_n)) \to r_1 + \ldots + r_n} \quad \text{(TUPLE)}$$

$$\frac{v ::\texttt{node}}{v \to (id(v), v)} \quad \text{(XMLNODE)}$$

$$\frac{v ::\texttt{val}}{v \to (v)} \quad \text{(ATOMICVALUE)}$$

$$\frac{v :: C\{S\} \qquad v \equiv [t_1, t_2, \ldots, t_m] \qquad t_i \to r_i \qquad i = 1 \ldots m}{v \to (\ (r_1, \ldots, r_m)\ )} \quad \text{(COLLVALUE)}$$

Fig. 6. Data model translation rules.

5 XML ALGEBRA TO PACT

Within the global approach depicted in Figure 4, this section describes our contribution: translating (i) from the Extended XQuery Data Model (or EXDM, in short) into the PACT Data Model (Section 5.1) and (ii) from algebraic expressions into PACT plans (Section 5.2). The most complex technical issues are raised by the latter.

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 (i) the *parallelization contract* (and possibly its corresponding *key fields*), and (ii) the *user function*, which together determine the PACT behavior. The hardest to translate are those algebraic operators whose input *cannot* be fragmented based on conjunctive key equalities (e.g., disjunctive joins) . This is because all massively parallel operators in PACT are based on key equality comparisons [8].

Translation rules. As in [39], we use deduction rules to specify our translation. In a nutshell, a deduction rule describes *how the translation is performed when some conditions are met over the input*. Our rules rely on *translation judgments*, noted as J, J_i, and are of the form:

$$\frac{cond \quad J_1 \ \ldots \ J_n}{J}$$

stating that the translation J (conclusion) is recursively made in terms of translations $J_1 \ \ldots \ J_n$ (premises) when the (optional) condition $cond$ holds. The translation judgments J_i are optional; their absence denotes that the rule handles the "fixpoint" (start of the recursive translation).

5.1 Translating XML tuples into PACT records

Rules for translating instances of EXDM into those of PACT rely on translation judgments of the form $\boxed{t \to r}$, or: "*the EXDM instance t translates into the PACT record r*".

The translation rules appear in Figure 6, where $+$ denotes record concatenation. *Rules produce records whose key fields are not set yet*; as we will see in Section 5.2, the keys are filled in by the translation.

Rule (TUPLE) produces a record from a tuple: it translates each tuple value, and then builds the output record r by concatenating the results according to tuple order.

There are three rules that can be triggered by rule (TUPLE). First, rule (XMLNODE) translates an XML node into a record with two fields: the first one contains the XML ID, while the second is the text serialization of the XML tree rooted at the node. In turn, rule (ATOMICVALUE) translates an XML value. Finally, rule (COLLVALUE) translates a tuple collection into a single-field record that contains the nested collection of records corresponding to the tuples in the input.

TABLE 1
Auxiliary functions details.

Signature	Description
$S; V \mapsto_{id} F$	Given the variable paths V bound to XML nodes according to S, returns the index path positions F in S-records corresponding to the XML node IDs.
$S; V \mapsto_{v} F$	Given a list of variable paths V bound to XML nodes, atomic values or collections, according to S, returns the index path positions F of the values of those variables in S-records.
$S; V \mapsto_{id,v} F$	"Union" of the two previous functions.
$S; L \mapsto L'$	Given a list of CTPs L, returns the CTPs L' where variables are replaced with corresponding fields in S-records.
$S; e \mapsto e'$	Given an ETP e whose root is a variable in S, builds a new ETP e' rooted with the corresponding field position in S-records.
$S; \rho \mapsto \rho'$	As above (replace ETPs with predicates).
$S_1, S_2; \rho \mapsto \rho'$	Given a predicate ρ referencing variables in tuples in S_1 and S_2, generates a new predicate ρ' referencing field positions in S_1- and S_2-records.

5.2 Translating algebraic expressions to PACT

Rules for translating an algebraic expression into a PACT plan are based on judgments of the form $\boxed{\mathcal{A} \Rightarrow \mathcal{P}}$, or: "*$\mathcal{A}$ translates into a PACT plan $\mathcal{P}$*". All rules are defined recursively over the structure of their input $\mathcal{A}$; for instance, the translation of $\mathcal{A} = sel_{\rho}(\mathcal{A}')$ relies on the PACT plan $\mathcal{P}'$ resulting from the translation of the smaller expression $\mathcal{A}'$, and so on.

The specific behavior of each rule is encoded in the choice of the parallelization contracts (and corresponding keys) and the user functions, so this is what we comment on below.

Preliminaries. In the translation, we denote a PACT operator by its parallelization contract $\subset$, user function f and the list K of key field positions in the PACT input. In particular:

- a unary PACT is of the form $\subset_f^K$; if $K{=}\emptyset$, for simplicity we omit it and use just $\subset_f$.
- a binary PACT is of the form $\subset_f^{K_1,K_2}$, assuming that the key of the left input records consists of the fields K_1 and that of the right input records of K_2, respectively.

To keep track of attribute position through the translation, we use a set of *helper functions* associating to variables from S, the index positions of the corresponding fields in the PACT records. These functions are outlined in Table 1; we use the term S-records as a shortcut for records obtained by translating tuples that conform to schema S. The helper functions implementation details are quite straightforward.

$$\frac{\mathcal{A} \Rightarrow \mathcal{P} \quad S_{\mathcal{A}}; L \mapsto L'}{constr_L(\mathcal{A}) \Rightarrow xmlwrite_{L'}(\mathcal{P})} \quad \text{(CONSTRUCTION)}$$

$$\frac{}{scan(f) \Rightarrow xmlscan(f)} \quad \text{(SCAN)}$$

Fig. 7. Border operators translation rules.

$$\frac{\mathcal{A} \Rightarrow \mathcal{P} \quad S_{\mathcal{A}}; e \mapsto e' \quad f := \overline{nav}(e')}{nav_e(\mathcal{A}) \Rightarrow \mathtt{mp}_f(\mathcal{P})} \quad \text{(NAVIGATION)}$$

$$\frac{\mathcal{A} \Rightarrow \mathcal{P} \quad S_{\mathcal{A}}; G_{id} \mapsto_{id} G'_{id} \quad S_{\mathcal{A}}; G_v \mapsto_v G'_v \quad K := G'_{id} + G'_v \quad f := \overline{grp}(K)}{grp_{G_{id},G_v,\$r}(\mathcal{A}) \Rightarrow \mathtt{rd}_f^K(\mathcal{P})} \quad \text{(GROUP-BY)}$$

$$\frac{\mathcal{A} \Rightarrow \mathcal{P} \quad S_{\mathcal{A}}; p \mapsto_v pi \quad f := \overline{flat}(pi)}{flat_p(\mathcal{A}) \Rightarrow \mathtt{mp}_f(\mathcal{P})} \quad \text{(FLATTEN)}$$

$$\frac{\mathcal{A} \Rightarrow \mathcal{P} \quad S_{\mathcal{A}}; \rho \mapsto \rho' \quad f := \overline{sel}(\rho')}{sel_\rho(\mathcal{A}) \Rightarrow \mathtt{mp}_f(\mathcal{P})} \quad \text{(SELECTION)}$$

$$\frac{\mathcal{A} \Rightarrow \mathcal{P} \quad S_{\mathcal{A}}; V \mapsto_{id,v} V' \quad f := \overline{proj}(V')}{proj_V(\mathcal{A}) \Rightarrow \mathtt{mp}_f(\mathcal{P})} \quad \text{(PROJECTION)}$$

$$\frac{\begin{array}{c}\mathcal{A} \Rightarrow \mathcal{P} \\ S_{\mathcal{A}}; p \mapsto_v pi \\ \texttt{if} \quad p.length \neq 1 \\ \texttt{then} \quad f := \overline{agg}_n(pi, a) \quad U := \mathtt{mp}_f \\ \texttt{else} \quad K := \emptyset \quad f := \overline{agg}(pi, a) \quad U := \mathtt{rd}_f^K\end{array}}{agg_{p,a,\$r}(\mathcal{A}) \Rightarrow U(\mathcal{P})} \quad \text{(AGGREGATION)}$$

$$\frac{\mathcal{A} \Rightarrow \mathcal{P} \quad S_{\mathcal{A}}; V \mapsto_v K \quad f := \overline{dupelim}}{dupelim_V(\mathcal{A}) \Rightarrow \mathtt{rd}_f^K(\mathcal{P})} \quad \text{(DUPLELIM)}$$

Fig. 8. Unary operators translation rules.

5.2.1 Border operators translation

Figure 7 outlines the translation of border operators.

Rule (CONSTRUCTION) translates the logical $constr_L$ operator into a data sink that uses our output function $xmlwrite$. For each input record from $\mathcal{P}$, $xmlwrite$ generates XML content using the list of construction patterns in L' and writes the results to a file.

Rule (SCAN) translates the logical operator $scan_f$ into a data source built up by means of our input function $xmlscan$. For each XML document in f, $xmlscan$ returns a single-field record holding the content of the document.

5.2.2 Unary operators translation

Unary operators are translated by the rules in Figure 8.

Rule (NAVIGATION) uses an auxiliary judgment that translates the input ETP e into e' using $S_{\mathcal{A}}$. Navigation is applied over each record independently, and thus we use a PACT with a *Map* contract. The UF is $\overline{nav}$, which generates new records from the (possibly partial) embeddings of e' in each input record.

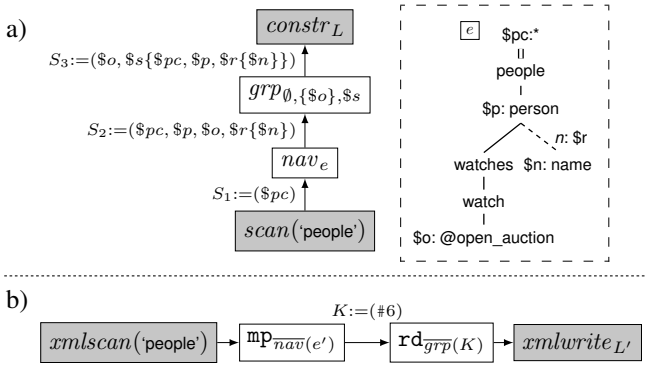


Fig. 9. Logical expression (a) and corresponding PACT plan (b) for the query in Example 2.

Rule (GROUP-BY) translates a group-by expression into a PACT with a *Reduce* contract, as the records need to be partitioned by the value of their grouping fields. The fields in K, which form the key used by the *Reduce* contract, are obtained appending G'_v to G'_{id}. K is also handed to the $\overline{grp}$ UF, which creates one record from each input collection of records. The new record contains the values for each field in K, and a new field which is the collection of the input records themselves.

Example 2. The following XQuery groups together the people that share interest in the same auctions:

```
let $pc := collection('people')
for $p in $pc//people/person,
    $o in $p/watches/watch/@open_auction
let $n := $p/name
group by $o
return <res><a>{$o}</a>{$n}</res>
```

The XML algebraic expression generated from this query is shown in Figure 9a. Using the judgments in Figure 8, the expression is translated into the PACT plan of Figure 9b. Observe that the grouping variable $\$o$ is translated into field position #6, used as key for the *Reduce* PACT. ◇

Rule (FLATTEN) translates a flatten expression into a *Map* PACT, that applies the flattening UF $\overline{flat}$ on each input record independently. The path pi to the nested collection is obtained from p using $S_{\mathcal{A}}$.

Rule (SELECTION) produces a *Map* PACT that applies the selection to each record produced by $\mathcal{P}$. Selection is performed by the $\overline{sel}$ UF, which uses the filtering condition ρ' obtained from ρ and $S_{\mathcal{A}}$.

Rule (PROJECTION) translates a projection expression into a PACT using a *Map* contract. The positions V' of the fields that should be kept by the projection are obtained from V using the schema $S_{\mathcal{A}}$.

The translation of (AGGREGATION) is interesting as it can use one PACT or another, depending on the path p to the variable being aggregated. If the variable is contained in a nested collection, i.e., $p.length \neq 1$, we produce a PACT with a *Map* contract; for each input record, the $\overline{agg}_n$ UF executes the aggregation operation a over the field pointed by pi and outputs a record with the aggregation results.

$$\frac{\mathcal{A}_1 \Rightarrow \mathcal{P}_1 \qquad \mathcal{A}_2 \Rightarrow \mathcal{P}_2 \qquad f := \overline{concat}}{prod(\mathcal{A}_1, \mathcal{A}_2) \Rightarrow \mathtt{cr}_f(\mathcal{P}_1, \mathcal{P}_2)} \quad \text{(CARTESIANPRODUCT)}$$

$$\frac{\mathcal{A}_1 \Rightarrow \mathcal{P}_1 \qquad \mathcal{A}_2 \Rightarrow \mathcal{P}_2 \qquad S_{\mathcal{A}_1}, S_{\mathcal{A}_2}; \rho \mapsto \rho' \qquad \rho' \rightharpoonup_l K_1 \qquad \rho' \rightharpoonup_r K_2 \qquad f := \overline{concat}}{join_\rho(\mathcal{A}_1, \mathcal{A}_2) \Rightarrow \mathtt{mt}_f^{K_1,K_2}(\mathcal{P}_1, \mathcal{P}_2)} \quad (\wedge \text{ JOIN}_=)$$

$$\frac{\mathcal{A}_1 \Rightarrow \mathcal{P}_1 \qquad \mathcal{A}_2 \Rightarrow \mathcal{P}_2 \qquad S_{\mathcal{A}_1}, S_{\mathcal{A}_2}; \rho \mapsto \rho' \qquad \rho' \rightharpoonup_l K_1 \qquad \rho' \rightharpoonup_r K_2 \qquad f := \overline{oconcat}_l}{ojoin^l_\rho(\mathcal{A}_1, \mathcal{A}_2) \Rightarrow \mathtt{cg}_f^{K_1,K_2}(\mathcal{P}_1, \mathcal{P}_2)} \quad (\text{LO} \wedge \text{ JOIN}_=)$$

$$\frac{\mathcal{A}_1 \Rightarrow \mathcal{P}_1 \qquad \mathcal{A}_2 \Rightarrow \mathcal{P}_2 \qquad S_{\mathcal{A}_1}, S_{\mathcal{A}_2}; \rho \mapsto \rho' \qquad \rho' \rightharpoonup_l K_1 \qquad \rho' \rightharpoonup_r K_2 \qquad f := \overline{noconcat}_l}{nojoin^l_\rho(\mathcal{A}_1, \mathcal{A}_2) \Rightarrow \mathtt{cg}_f^{K_1,K_2}(\mathcal{P}_1, \mathcal{P}_2)} \quad (\text{NLO} \wedge \text{ JOIN}_=)$$

Fig. 10. Cartesian product and conjunctive equi-join translation rules.

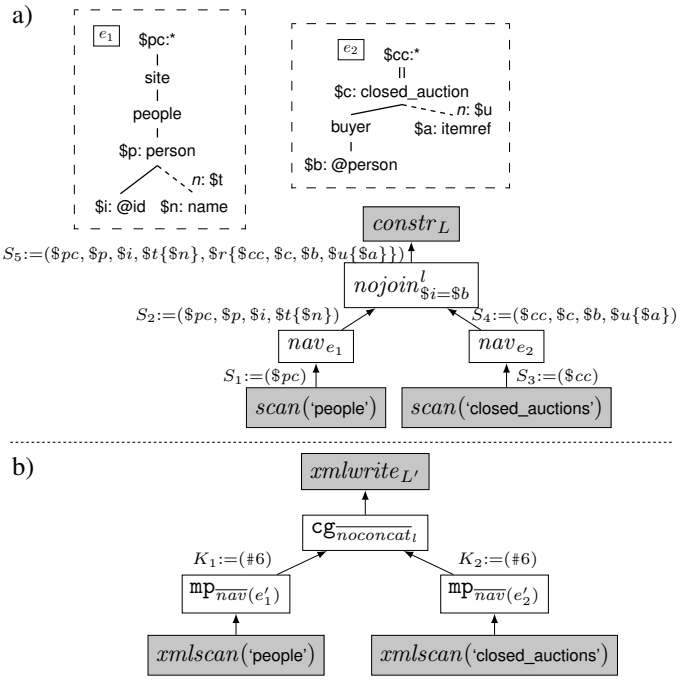


Fig. 11. Logical expression (a) and corresponding PACT plan (b) for the query in Example 3.

Otherwise, if the aggregation is executed on the complete input collection, we use a *Reduce* contract wrapping the input in a single group. The $\overline{agg}$ UF creates an output record having (i) a field with a nested collection of all input records and (ii) a field with the result of executing the aggregation a over the field pointed by pi.

Finally, rule (DUPLELIM) translates a duplicate elimination expression into a PACT with a *Reduce* contract. Each group handed to the UF holds the bag of records containing the same values in the fields pointed by K; the duplicate elimination UF, denoted by $\overline{dupelim}$, outputs only one record from the group.

5.2.3 Binary operators translation

The rules are depicted in Figure 10; we assume that the inputs $\mathcal{A}_1$ and $\mathcal{A}_2$ of the algebraic binary operator translate into the PACT plans $\mathcal{P}_1$ and $\mathcal{P}_2$.

a) Cartesian product. This operator requires the simple *concatenation* UF, taking as input a pair of records, and outputting their concatenation: $\overline{concat}(r_1, r_2) = r_1 + r_2$.

Rule (CARTESIANPRODUCT) translates a cartesian product into a *Cross* PACT with a $\overline{concat}$ UF.

b) Joins with conjunctive equality predicates. This family comprises joins on equality predicates, which can be simple (natural) equi-joins, or outer joins (without loss of generality we focus on left outer joins).

b.1) Conjunctive equi-join. The conjunctive equi-join operator is translated by rule ($\wedge$ JOIN$_=$), as follows. First, the predicate ρ over $\mathcal{A}_1$ and $\mathcal{A}_2$ translates into a predicate ρ' over records produced by $\mathcal{P}_1$ and $\mathcal{P}_2$. Then, the list of fields pointed by the left ($\rightharpoonup_l$), resp. right ($\rightharpoonup_r$) of the condition ρ' are extracted, and finally they are used as the keys of the generated *Match* PACT.

b.2) Left outer conjunctive equi-join. In the rule (LO $\wedge$ JOIN$_=$), the output PACT is a *CoGroup* whose keys are taken from the fields of the translated join predicate ρ'. The CoGroup contract groups the records produced by $\mathcal{P}_1$ and $\mathcal{P}_2$ sharing the same key. Then, the $\overline{oconcat}_l$ UF that we describe next is applied over each group, to produce the expected result.

Definition 1 ($\overline{oconcat}_l$): The left outer concatenation UF, $\overline{oconcat}_l$, of two record bags $\{\{r_1, \ldots, r_x\}\}$ and $\{\{r'_1, \ldots, r'_y\}\}$ is defined as:

- If $y \neq 0$, the cartesian product of the two bags.
- Otherwise, $\{\{r_1 + \perp', \ldots, r_x + \perp'\}\}$ i.e., concatenate each left input record with a $\perp$-record having the schema (structure) of the right records. $\diamond$

b.3) Nested left outer conjunctive equi-join. Similar to the non-nested case, rule (NLO $\wedge$ JOIN$_=$) translates the nested left outer conjunctive equi-join into a *CoGroup* PACT whose key is extracted from ρ'. However, we need a different UF in order to generate the desired right-hand side nested records, and we define it below.

Definition 2 ($\overline{noconcat}_l$): The nested left outer concatenation UF, $\overline{noconcat}_l$, of the bags $\{\{r_1, \ldots, r_x\}\}$ and $\{\{r'_1, \ldots, r'_y\}\}$ is defined as:

- If $y \neq 0$, $\{\{r_1 + (r'_1, \ldots, r'_y), \ldots, r_x + (r'_1, \ldots, r'_y)\}\}$ i.e., nest the right set as a new field concatenated to each record from the left.
- Otherwise, $\{\{r_1 + (\perp'), \ldots, r_x + (\perp')\}\}$ i.e., add to each left record a field with a list containing a $\perp$-record conforming to the schema of the right records. $\diamond$

Example 3. The following XQuery extracts the name of users and the items that they bought (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,
```

$$\frac{\begin{array}{c}\mathcal{A}_1 \Rightarrow \mathcal{P}_1 \qquad \mathcal{A}_2 \Rightarrow \mathcal{P}_2 \\ S_{\mathcal{A}_1}, S_{\mathcal{A}_2}; \rho \mapsto \rho' \qquad \rho' \equiv \rho'_1 \vee \ldots \vee \rho'_n \\ \rho'_k \rightarrow_l K_{1k} \qquad \rho'_k \rightarrow_r K_{2k} \\ f_k := \overline{pnjoin}(\rho', k{-}1) \qquad k = 1 \ldots n \\ U := \{\mathtt{mt}_{f_1}^{K_{11},K_{21}}, \ldots, \mathtt{mt}_{f_n}^{K_{1n},K_{2n}}\}\end{array}}{join_\rho(\mathcal{A}_1, \mathcal{A}_2) \Rightarrow U(\mathcal{P}_1, \mathcal{P}_2)} \quad (\vee \text{ JOIN}_=)$$

$$\frac{\begin{array}{c}\mathcal{A}_1 \Rightarrow \mathcal{P}_1 \qquad \mathcal{A}_2 \Rightarrow \mathcal{P}_2 \\ S_{\mathcal{A}_1}, S_{\mathcal{A}_2}; \rho \mapsto \rho' \qquad \rho' \equiv \rho'_1 \vee \ldots \vee \rho'_n \\ \rho'_k \rightarrow_l K_{1k} \qquad \rho'_k \rightarrow_r K_{2k} \\ f_k := \overline{nopnjoin}_l(\rho', k{-}1) \qquad k = 1 \ldots n \\ U := \{\mathtt{cg}_{f_1}^{K_{11},K_{21}}, \ldots, \mathtt{cg}_{f_n}^{K_{1n},K_{2n}}\} \\ S_{\mathcal{A}_1} \rightsquigarrow K \qquad f' := \overline{opost}_{l\vee}\end{array}}{ojoin_\rho^l(\mathcal{A}_1, \mathcal{A}_2) \Rightarrow \mathtt{rd}_{f'}^K(U(\mathcal{P}_1, \mathcal{P}_2))} \quad (\text{LO} \vee \text{ JOIN}_=)$$

$$\frac{\begin{array}{c}\mathcal{A}_1 \Rightarrow \mathcal{P}_1 \qquad \mathcal{A}_2 \Rightarrow \mathcal{P}_2 \\ S_{\mathcal{A}_1}, S_{\mathcal{A}_2}; \rho \mapsto \rho' \qquad \rho' \equiv \rho'_1 \vee \ldots \vee \rho'_n \\ \rho'_k \rightarrow_l K_{1k} \qquad \rho'_k \rightarrow_r K_{2k} \\ f_k := \overline{nopnjoin}_l(\rho', k{-}1) \qquad k = 1 \ldots n \\ U := \{\mathtt{cg}_{f_1}^{K_{11},K_{21}}, \ldots, \mathtt{cg}_{f_n}^{K_{1n},K_{2n}}\} \\ S_{\mathcal{A}_1} \rightsquigarrow K \qquad f' := \overline{nopost}_{l\vee}\end{array}}{nojoin_\rho^l(\mathcal{A}_1, \mathcal{A}_2) \Rightarrow \mathtt{rd}_{f'}^K(U(\mathcal{P}_1, \mathcal{P}_2))} \quad (\text{NLO} \vee \text{ JOIN}_=)$$

Fig. 12. Disjunctive equi-join translation rules.

```
      $b in $c/buyer/@person
  let $a := $c/itemref
  where $i = $b
  return $a
return <res>{$n,$r}</res>
```

The query translates into the algebraic expression depicted in Figure 11a, while the corresponding PACT plan is shown in Figure 11b.

Rule (NLO $\wedge$ JOIN$_=$) translates the nested left outer conjunctive equi-join into a PACT with a *CoGroup* contract that groups together all records having the same values in the fields corresponding to $\$i$ (K_1) and $\$b$ (K_2), and applies our $\overline{noconcat}_l$ UF on them. ◇

c) Joins with disjunctive equality predicates. Translating joins with disjunctive equality predicates is harder. The reason is that PACT contracts are centered around *equality* of record fields, and thus *inherently not suited* to disjunctive semantics. To solve this mismatch, our translation relies on using more than one PACT for each operator, as we explain below.

c.1) Disjunctive equi-join. In rule ($\vee$ JOIN$_=$), the predicate ρ' is generated from ρ using $S_{\mathcal{A}_1}$ and $S_{\mathcal{A}_2}$. Then, for each conjunctive predicate ρ'_k in ρ', we create a *Match* whose keys are the fields participating in ρ'_k. Observe that the UFs of these *Match* operators should guarantee that no erroneous duplicates are generated when the evaluation of more than one conjunctive predicates ρ'_i, ρ'_j, $i \neq j$ is true for a certain record. To that purpose, we define the new UF $\overline{pnjoin}$ below, parameterized by k and performing a partial negative join.

Definition 3 ($\overline{pnjoin}$): Let $\rho' = \rho'_1 \vee \rho'_2 \vee \ldots \vee \rho'_n$ and k be an integer, with $0 \leq k < n$. Given two records r_1, r_2, the $\overline{pnjoin}(\rho', k)$ UF evaluates $\rho'_1 \vee \rho'_2 \vee \ldots \vee \rho'_k$ over r_1, r_2, and outputs $r_1{+}r_2$ if they evaluate to false. ◇

Note that the UF ensures correct multiplicity of each record in the result.

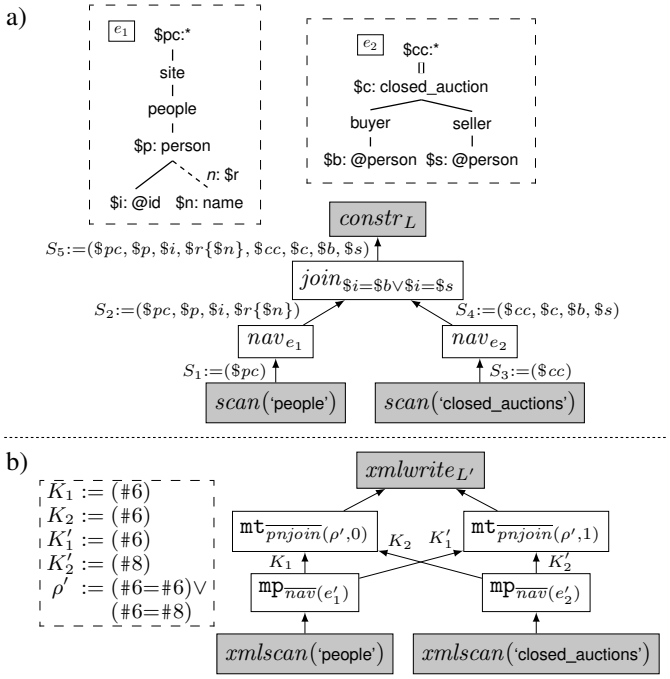


Fig. 13. Logical expression (a) and corresponding PACT plan (b) for the query in Example 4.

Example 4. The following XQuery extracts the names of users involved in at least one auction, either as buyers or sellers:

```
let $pc := collection('people'),
    $cc := collection('closed_auctions')
for $p in $pc/site/people/person, $i in $p/@id,
    $c in $cc//closed_auction,
    $b in $c/buyer/@person,
    $s in $c/seller/@person
let $n := $p/name
where $i = $b or $i = $s
return <res>{$n}</res>
```

Rule ($\vee$ JOIN$_=$) translates the disjunctive equi-join into two PACTs with *Match* contracts, one per disjunction. Observe that two distinct values (0 and 1) of k are used in the $\overline{pnjoin}$ UFs to prevent spurious duplicates, one for the predicate `$i=$b` and one for `$i=$s`. ◇

c.2) (Nested) left outer disjunctive equi-join. The translation of the plain and nested variants of the outer disjunctive equi-join, described by the (LO $\vee$ JOIN$_=$) and (NLO $\vee$ JOIN$_=$) rules respectively, are very similar; as illustrated next, the main difference resides in the different post-processing operations they adopt. The translation of these two operators is challenging because we want to ensure parallel evaluation of each conjunctive join predicate in the disjunction, and at the same time we need to:

1) *Avoid the generation of duplicate records*. We adopt a non trivial variation of the technique used previously for disjunctive equi-join.
2) *Recognise records generated by the left hand-side expression which do not join any record coming from the right-hand side expression*. We use the XML node identifiers in each left hand-side record to identify it uniquely, so that, after the parallel evaluation of each

conjunction, a Reduce post-processing PACT groups all resulting combinations having the same left hand-side record; if none of such combinations exists, the left hand-side record representing a group is concatenated to a (nested) $\perp$-record conforming to the right input schema, and the resulting record is output; otherwise the output record(s) are generated from the combinations.

In the first step, we must evaluate in parallel the joins related to predicates ρ'_i. A PACT with a *CoGroup* contract is built *for each conjunctive predicate* ρ'_k. Each such PACT groups together all records that share the same value in the fields pointed by ρ'_k, then applies the $\overline{nopnjoin_l}$ UF (see below) on each group, with the goal of avoiding erroneous duplicates in the result; the UF is more complex than $\overline{pnjoin}$ though, as it has to handle the disjunction and the nesting. $\overline{nopnjoin_l}$ is parameterized by k, as we will use it once for each conjunction ρ'_k. Furthermore, $\overline{nopnjoin_l}$ takes as input *two* bags of records and is defined as follows, along the lines of $\overline{pnjoin}$.

Definition 4 ($\overline{nopnjoin_l}$): Let $\rho' = \rho'_1 \vee \rho'_2 \vee \ldots \vee \rho'_n$ be a predicate where each ρ'_i is conjunctive. Given two input bags $\{\{r_1, \ldots, r_x\}\}$ and $\{\{r'_1, \ldots, r'_y\}\}$, the $\overline{nopnjoin_l}(\rho', k)$ UF is defined as follows:

- If the second input is empty ($y = 0$), return $\{\{r_1 + (\perp'), \ldots, r_x + (\perp')\}\}$ i.e., concatenate every left input record with a field containing a nested list of one $\perp$-record conforming to the schema of the right input.
- Otherwise, for each left input record r_i:
  1) create an empty list c_i;
  2) for each $r'_{j,1\leq j\leq y}$, evaluate $\rho'_1 \vee \rho'_2 \vee \ldots \vee \rho'_k$ over r_i and r'_j, and add r'_j to c_i *if the result is false*;
  3) if c_i is empty, then insert into c_i a $\perp$-record with the schema of the right input;
  4) output r_i concatenated with a new field whose value is c_i. $\diamond$

The second PACT produced by the (LO $\vee$ JOIN$_=$) and (NLO $\vee$ JOIN$_=$) rules uses a *Reduce* contract, taking as input the outputs of all the *CoGroup* operators; its key consists of the XML node identifiers in each left hand-side record (we denote by $\rightsquigarrow$ the extraction of these fields from the schema). This amounts to grouping together the records originated from the same left input record.

Depending on the join flavor though, this last PACT uses a different UF. For the plain (non-nested) join (LO $\vee$ JOIN$_=$), we use the $\overline{opost}_{l\vee}$ UF producing records with an unnested right side. For the nested join (NLO $\vee$ JOIN$_=$), on the other hand, the $\overline{nopost}_{l\vee}$ UF is used to produce nested records. Due to space constraints, we omit the definition of these UFs here and delegate their details to Appendix C.

Example 1 (*continuation*). Our algorithms translate the algebraic expression shown in Figure 5 into the PACT plan depicted in Figure 14; observe that it is the same PACT plan that was shown in less detail in Figure 1.

Rule (NLO $\vee$ JOIN$_=$) translates the nested left outer disjunctive equi-join into (i) two PACTs with *CoGroup* contracts, one for each disjunction, and (ii) a PACT with a *Reduce* contract that groups together records originating from the same

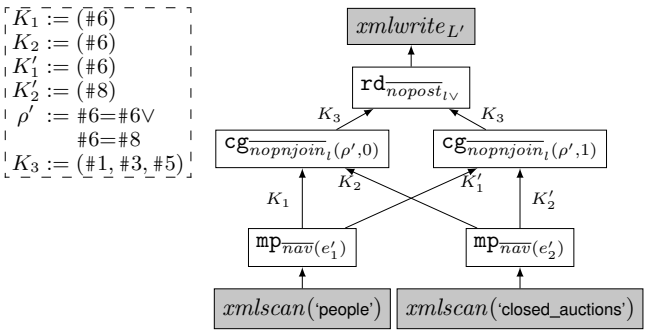


Fig. 14. PACT plan corresponding to the logical expression in Figure 5.

left-hand side record, i.e., K_3 holds field positions $\#1, \#3, \#5$, which contain the XML node identifiers of $\$pc, \$p, \$i$, respectively. $\diamond$

d) Joins on inequalities. Our XQuery subset also supports joins with inequality conditions. In this case, the translation uses *Cross* contracts. Further, just like for joins with disjunctive predicates, the non-nested and nested outer variants of the joins on inequalities require more than one PACT. The corresponding translation rules can be found in Appendix D.

Syntactically complex translation vs. performance Clearly, complex joins such as those considered in **c)** could be translated into a single *Cross* PACT over the pairs of records as in **d)**. However, this would be less efficient and scale poorly (number of comparisons quadratic in the input size), as our experiments will demonstrate.

6 Experimental evaluation

We implemented our PAXQuery translation approach in Java 1.6, and relied on the Stratosphere platform [44] supporting PACT. We first describe the experimental setup, and then present our results.

Experimental setup. The experiments run in a cluster of 8 nodes on an 1GB Ethernet. Each node has 2 $\times$ 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 Stratosphere 0.2.1; it stores the XML data in HDFS 1.1.2.

XML data. We used XMark [42] 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 simply stored in HDFS (which replicates them three times), that is, *we do not control the distribution/allocation of documents over the nodes*.

XML queries. We used a subset of XMark queries from our XQuery fragment, and added queries with features supported by our dialect but absent from the original XMark, e.g., joins on disjunctive predicates; all queries are detailed in Appendix E.

Table 2 outlines the queries: the collection(s) that each query carries over, the corresponding XML algebraic operators and

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 auct.*	Navigation (1)	Map (1)
q_5	*closed auct.*	Navigation (1)	Map (1)
q_6	*users*	Navigation (1)	Map (1)
q_7	*closed auct.*	Navigation (1) Aggregation (2)	Map (2) Reduce (1)
q_8	*items*	Navigation (1) Aggregation (2)	Map (2) Reduce (1)
q_9	*users* *closed auct.*	Navigation (2) Projection (1) Group-by/aggregation (1) Conj. equi-join (1)	Map (3) Reduce (1) Match (1)
q_{10}	*users* *items* *closed auct.*	Navigation (3) Projection (2) NLO conj. equi-join (2)	Map (5) CoGroup (2)
q_{11}	*users*	Navigation (2) Projection (1) Dup. elim. (1) NLO conj. equi-join (1)	Map (3) Reduce (1) CoGroup (1)
q_{12}	*users* *closed auct.*	Navigation (2) Projection (1) NLO conj. equi-join/ aggregation (1)	Map (3) CoGroup (1)
q_{13}	*users* *closed auct.*	Navigation (2) Projection (1) NLO disj. equi-join (1)	Map (3) Reduce (2) CoGroup (2)
q_{14}	*users* *open auct.*	Navigation (2) Projection (1) NLO inequi-join (1)	Map (3) Reduce (2) Cross (1)

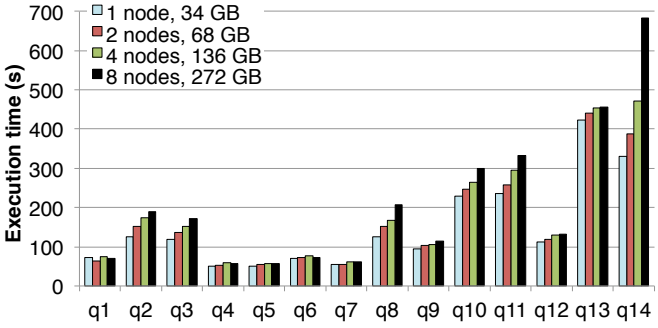


Fig. 15. PAXQuery scalability evaluation.

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*.

6.1 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 15 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; each translates into a *Map* PACT, thus their response time follows the size of the input. These queries scale up well; we see a moderate overhead in Figure 15 as the data volume and number of nodes increases.

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 the rules in Figure 12). 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.

TABLE 3
Query evaluation time (1 node, 34GB).

Query	Evaluation time (seconds)			
	BaseX	Saxon-PE	Qizx/open	PAXQuery
q_1	206	145	90	**72**
q_2	629	~~OOM~~	~~OOM~~	**125**
q_3	600	~~OOM~~	~~OOM~~	**120**
q_4	189	~~OOM~~	84	**51**
q_5	183	125	183	**51**
q_6	233	162	109	**70**
q_7	181	111	88	**54**
q_8	599	~~OOM~~	~~OOM~~	**126**
q_9	~~TO~~	~~OOM~~	~~OOM~~	**94**
q_{10}	~~OOM~~	~~OOM~~	~~OOM~~	**229**
q_{11}	~~TO~~	~~TO~~	~~TO~~	**236**
q_{12}	~~TO~~	~~OOM~~	~~OOM~~	**113**
q_{13}	~~TO~~	~~OOM~~	~~OOM~~	**424**
q_{14}	~~OOM~~	~~OOM~~	~~OOM~~	**331**

6.2 Comparison against other processors

To evaluate the performance of our processor against existing alternatives, we started by comparing it *on a single node* with other popular centralized XQuery processors. The purpose is to validate our choice of an XML algebra as outlined in Section 4.2 as input to our translation, by demonstrating that *single-site* query evaluation based on such an algebra is efficient. For this, we compare our processor with BaseX 7.7 [7], Saxon-PE 9.4 [41] and Qizx/open 4.1 [38], on a dataset of 11000 XML documents (**34GB**).

Table 3 shows the response times for each query and processor; the shortest time is shown in bold, while ~~OOM~~ stands for *out of memory*, and ~~TO~~ for *timeout* (above 2 hours). In Table 3, we identify two query groups. First, q_1-q_8 do not feature joins; while the performance varies across systems,

TABLE 4
Query evaluation time (8 nodes, 272GB).

Query	Evaluation time (seconds)		
	BaseX Hadoop-MR	BaseX Stratosphere-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**

only BaseX and PAXQuery are able to run all these queries. PAXQuery outperforms other systems because, compiled in PACT, it is able to exploit the multicore architecture.

In the second group, queries q_9-q_{14} join across the documents. None of the competing XQuery processors completes their evaluation, while PAXQuery executes them quite fast. For these, the usage of outer joins and multicore parallelization are key to this good performance behavior.

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 centralized system (the best performing in the experiment above) and used Hadoop-MapReduce on one hand, and Stratosphere-PACT on the other hand, to parallelize its execution.

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 [27], 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.

Table 4 shows the response times when running the query on the 8 nodes and 272GB; the shortest time is in bold. First, we notice that BaseX runs 2 to 5 times faster on Stratosphere than on Hadoop. This is due to Hadoop's checkpoints (writing intermediary results to disk) while Stratosphere 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 Stratosphere; 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 again. 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.

6.3 Conclusions of the experiments

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. [35].

Secondly, we have shown that PAXQuery outperforms competitor XQuery processors, whether centralized or distributed over Hadoop and Stratosphere. None of the competing processors was able to evaluate any of our queries with joins across documents on the data volumes we considered, highlighting the need for efficient parallel platforms for evaluating such queries.

7 Related Work

Massively parallel XML query processing. In this area, MRQL [22] proposes a simple SQL-like XML query language implemented through a few operators directly compilable into MapReduce. Like our XQuery fragment, MRQL queries may be nested, however, its dialect does not allow 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 [27] 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 implicitly parallel approach which does not expose the underlying parallelism at the query level.

HadoopXML [18] and the recent [12] process XML queries in Hadoop clusters by *explicitly fragmenting the input data* in a schema-driven, respectively, query-driven way, which is effective when querying one single huge document. In contrast, we focus on the frequent situation when no single document is too large for one node, but there are many documents whose global size is high, and queries may both

navigate and join over them. Further, we do not require any partitioning work from the application level.

After the wide acceptance of Hadoop, other parallel execution engines and programming abstractions conceived to run custom data intensive tasks over large data sets have been proposed: PACT [8], Dryad [25], Hyracks [15] or Spark [48]. Among these, the only effort at parallelizing XQuery is the ongoing VXQuery project [4], translating XQuery into the Algebricks algebra, which compiles into parallel plans executable by Hyracks. In contrast, PAXQuery translates into an implicit parallel logical model such as PACT. Thus, our algorithms do not need to address underlying parallelization issues such as data redistribution between computation steps etc. which [15] explicitly mentions.

XQuery processing in centralized settings has been thoroughly studied, in particular through algebras in [39], [20], [32], [31]. Our focus is on *extending the benefits of implicit large-scale parallelism to a complex XML algebra*, by formalizing its translation into the implicitly parallel PACT paradigm.

XML data management has also been studied from many other angles, e.g., on top of column stores [14], distributed with [28] or without [1] an explicit fragmentation specification, in P2P [29] etc. We focus on XQuery evaluation through the massively parallel PACT framework, which leads to specific translation difficulties we addressed.

Parallelizable nested languages. Recently, many high-level languages which translate into massively parallel frameworks have been proposed; some of them work with nested data and/or feature nesting in the language, thus somehow resemble XQuery.

Jaql [11] is a scripting language tailored to JSON data, which translates into MapReduce programs; Meteor [24], also for JSON, translates into PACT. None of these languages handles XQuery semantics exactly, since JSON does not feature node identity; the languages are also more limited, e.g., Jaql only supports equi-joins.

The Asterix Query Language [10], or AQL in short, is based on FLOWR expressions and resembles XQuery, but ignores node identity which is important in XQuery and which we support. Like VXQuery, AQL queries are translated into Algebricks; recall that unlike our translation, its compilation to the underlying Hyracks engine needs to deal with parallelization related issues.

Finally, other higher level languages that support nested data models and translate into parallel processing paradigms include Pig [36] or Hive [45]. Our XQuery fragment is more expressive, in particular supporting more types of joins. In addition, Pig only allows two levels of nesting in queries, which is a limitation. In contrast, we translate XQuery into unnested algebraic plans with (possibly nested, possibly outer) joins and grouping which we parallelize, leading to efficient execution even for (originally) nested queries.

Complex operations using implicit parallel models. The problem of evaluating complex operations through implicit parallelism is of independent interest. For instance, the execution of join operations using MapReduce has been studied extensively. Shortly after the first formal proposal to compute equi-joins on MapReduce [47], other studies extending it [13], [26] or focusing on the processing of specific join types such as multi-way joins [2], set-similarity joins [46], or θ-joins [35], appeared. PAXQuery is the first to translate a large family of joins (which can be used outside XQuery), into the more flexible PACT parallel framework.

8 Conclusion and Future Work

We have presented the PAXQuery approach for the implicit parallelization of XQuery, through the translation of an XQuery algebraic plan into a PACT parallel plan. We targeted a rich subset of XQuery 3.0 including recent additions such as explicit grouping, and demonstrated the efficiency and scalability of PAXQuery with experiments on collections of hundreds of GBs.

For future work, we contemplate the integration of indexing techniques into PAXQuery to improve query evaluation time. Further, we would like to explore reutilization of intermediary results in the PACT framework to enable efficient multiple-query processing.

References

[1] S. Abiteboul, A. Bonifati, G. Cobéna, I. Manolescu, and T. Milo, "Dynamic XML Documents with Distribution and Replication," in *SIGMOD*, 2003.

[2] F. N. Afrati and J. D. Ullman, "Optimizing joins in a map-reduce environment," in *EDBT*, 2010.

[3] "Apache Hadoop," http://hadoop.apache.org/.

[4] "Apache VXQuery," http://incubator.apache.org/vxquery/.

[5] A. Arion, V. Benzaken, and I. Manolescu, "XML Access Modules: Towards Physical Data Independence in XML Databases," in *XIME-P*, 2005.

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

[7] "BaseX," http://basex.org/products/xquery/.

[8] 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.

[9] C. Beeri and Y. Tzaban, "SAL: An Algebra for Semistructured Data and XML," in *WebDB*, 1999.

[10] A. Behm, V. R. Borkar, M. J. Carey, R. Grover, C. Li, N. Onose, R. Vernica, A. Deutsch, Y. Papakonstantinou, and V. J. Tsotras, "ASTERIX: Towards a Scalable, Semistructured Data Platform for Evolving-world Models," *Distributed and Parallel Databases*, 2011.

[11] 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.

[12] N. Bidoit, D. Colazzo, N. Malla, F. Ulliana, M. Nolè, and C. Sartiani, "Processing XML queries and updates on map/reduce clusters (demo)," in *EDBT*, 2013.

[13] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian, "A Comparison of Join Algorithms for Log Processing in MapReduce," in *SIGMOD*, 2010.

[14] 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.

[15] V. R. Borkar, M. J. Carey, R. Grover, N. Onose, and R. Vernica, "Hyracks: A flexible and extensible foundation for data-intensive computing," in *ICDE*, 2011.

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

[17] Z. Chen, H. V. Jagadish, L. Lakshmanan, and S. Paparizos, "From Tree Patterns to Generalized Tree Patterns: On Efficient Evaluation of XQuery," in *VLDB*, 2003.

[18] 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 (demo)," in *ACM CIKM*, 2012.

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

[20] A. Deutsch, Y. Papakonstantinou, and Y. Xu, "The NEXT Logical Framework for XQuery," in *VLDB*, 2004.
[21] A. Eisenberg, "XQuery 3.0 is nearing completion," *SIGMOD Record*, vol. 42, no. 3, 2013.
[22] L. Fegaras, C. Li, U. Gupta, and J. Philip, "XML Query Optimization in Map-Reduce," in *WebDB*, 2011.
[23] T. Fiebig and G. Moerkotte, "Algebraic XML Construction and its Optimization in Natix," *World Wide Web*, vol. 4, no. 3, 2001.
[24] A. Heise, A. Rheinländer, M. Leich, U. Leser, and F. Naumann, " Meteor/Sopremo: An Extensible Query Language and Operator Model ," in *BIGDATA*, 2012.
[25] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks," in *EuroSys*, 2007.
[26] D. Jiang, A. K. H. Tung, and G. Chen, "MAP-JOIN-REDUCE: Toward Scalable and Efficient Data Analysis on Large Clusters," *IEEE TKDE*, 2011.
[27] S. Khatchadourian, M. P. Consens, and J. Siméon, "Having a ChuQL at XML on the cloud," in *A. Mendelzon Int'l. Workshop*, 2011.
[28] P. Kling, M. T. Özsu, and K. Daudjee, "Generating Efficient Execution Plans for Vertically Partitioned XML Databases," *PVLDB*, 2010.
[29] G. Koloniari and E. Pitoura, "Peer-to-peer management of XML data: issues and research challenges," *SIGMOD Record*, vol. 34, no. 2, 2005.
[30] I. Manolescu and Y. Papakonstantinou, "XQuery Midflight: Emerging Database-Oriented Paradigms and a Classification of Research Advances," in *ICDE*, 2005.
[31] I. Manolescu, Y. Papakonstantinou, and V. Vassalos, "XML Tuple Algebra," in *Encyclopedia of Database Systems*, 2009.
[32] N. May, S. Helmer, and G. Moerkotte, "Strategies for query unnesting in XML databases," *TODS*, vol. 31, no. 3, 2006.
[33] P. Michiels, G. A. Mihaila, and J. Siméon, "Put a Tree Pattern in Your Algebra," in *ICDE*, 2007.
[34] G. Miklau and D. Suciu, "Containment and equivalence for an XPath fragment," in *PODS*, 2002.
[35] A. Okcan and M. Riedewald, "Processing theta-joins using MapReduce," in *SIGMOD*, 2011.
[36] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig Latin: A Not-So-Foreign Language for Data Processing," in *SIGMOD*, 2008.
[37] S. Paparizos, Y. Wu, L. V. S. Lakshmanan, and H. V. Jagadish, "Tree Logical Classes for Efficient Evaluation of XQuery," in *SIGMOD*, 2004.
[38] "Qizx/open," http://www.axyana.com/qizxopen/.
[39] C. Re, J. Siméon, and M. F. Fernández, "A Complete and Efficient Algebraic Compiler for XQuery," in *ICDE*, 2006.
[40] J. Robie, D. Chamberlin, M. Dyck, and J. Snelson, *XQuery 3.0: An XML Query Language*, W3C Proposed Recommendation, October 2013.
[41] "Saxon," http://www.saxonica.com/.
[42] 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.
[43] J. Shanmugasundaram, J. Kiernan, E. Shekita, C. Fan, and J. Funderburk, "Querying XML Views of Relational Data," in *VLDB*, 2001.
[44] "Stratosphere Platform," http://www.stratosphere.eu/.
[45] 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.
[46] R. Vernica, M. J. Carey, and C. Li, "Efficient Parallel Set-similarity Joins Using MapReduce," in *SIGMOD*, 2010.
[47] H.-c. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker, "Map-reduce-merge: Simplified Relational Data Processing on Large Clusters," in *SIGMOD*, 2007.
[48] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing," in *NSDI*, 2012.
[49] X. Zhang, B. Pielech, and E. A. Rundesnteiner, "Honey, I shrunk the XQuery!: an XML algebra optimization approach," in *WIDM*, 2002.

Query ::= *FLWRExpr*
FLWRExpr ::= *Initial Middle* Return*
Initial ::= *For* | *Let*
Middle ::= *Initial* | *Where* | *GroupBy*
For ::= for *ForBinding* (, *ForBinding*)*
ForBinding ::= *Var* in *PathExpr*
PathExpr ::= (distinct-values)? (collection(*Uri*) | doc(*Uri*) | *Var*) *Path*
Let ::= let *LetBinding* (, *LetBinding*)*
LetBinding ::= *Var* := (*FLWRExpr* | *AggrExpr* | *PathExpr*)
AggrExpr ::= (count | avg | max | min | sum) *Var*
Where ::= where *OrExpr*
OrExpr ::= *AndExpr* (or *AndExpr*)*
AndExpr ::= *BoolExpr* (and *BoolExpr*)*
BoolExpr ::= (not)? (*Pred* | *Contains* | *Empty*)
Pred ::= *Var* (*ValCmp* | *NodeCmp*) (*Var* | *C*)
Contains ::= contains (*Var*, *C*)
Empty ::= empty (*Var*)
GroupBy ::= group by *Var* (, *Var*)*
Return ::= return (*EleConst* | (*AggrExpr* | *Var*)+)
EleConst ::= <*EName Att** (/> | (> (*EleConst* | *AggrExpr* | *Var*)* </ *EName* >))
Att ::= *AName* = "(*AggrExpr* | *Var* | *AVal*)*"
Var ::= $*VarName*

Fig. 16. Grammar for the considered XQuery dialect.

APPENDIX A
XQUERY DIALECT

Figure 16 depicts the grammar for our XQuery dialect.

A query is a FLWR expression, which is a powerful abstraction that can be used for many purposes, including iterating over sequences, joining multiple documents, and performing grouping.

The initial clause of the expression is a *for* or *let*. The *for* clause iterates over the items in the sequence resulting from its associated expression, binding the variable to each item. In turn, a *let* clause binds each variable to the result of its associated expression, without iteration.

The bindings for *for* clauses are generated from an expression *PathExp*. A path is evaluated starting from the root of each document in a collection available at URI *Uri*, from the root of a single document available at URI *Uri*, or from the bindings of a previously introduced variable. *Path* corresponds to the navigational path used to locate nodes within trees. In particular, *Path* belongs to the XPath$^{\{/,//,[]\}}$ language [34]. In turn, the bindings for *let* clauses can be an expression *PathExp*, another FLWR expression or an aggregation expression *AggrExpr*.

The middle clauses (*for*, *let*, *where*, or *group by*) may appear multiple times and in any order. The *where* clause supports expressions formed with or and and, in *disjunctive normal form* (DNF). We support two different types of elementary comparators: (*ValCmp*) compares atomic values, while (*NodeCmp*) compares nodes by their identity or by their document order. The *group by* clause groups tuples based on the value of the variables specified in the clause.

Finally, the FLWR expression ends with a *return* clause. For each tuple of bindings, the clause builds an XML forest using an element construction expression *EleConst* or a list of variables *Var+*. When we use the element construction expression, the value in *AVal* follow the XML naming convention for attribute values, while *AName* and *EName* follow the restrictions associated to the XML node naming conventions.

$\mathcal{A}$	::= $constr_L$ (*Operator*)
Operator	::= *Scan* | *UnaryOp* | *BinaryOp*
Scan	::= $scan$
UnaryOp	::= (*Navigation* | *Group-By* | *Flatten* | *Selection* | *Projection* | *Aggregation* | *DupElim*) *Operator*
Navigation	::= nav_e
Selection	::= sel_ρ
Projection	::= $proj_V$
Group-By	::= $grp_{G_{id},G_v,\$r}$
Flatten	::= $flat_p$
Aggregation	::= $agg_{p,a,\$r}$
DupElim	::= $dupelim_V$
BinaryOp	::= (*CartProd* | *Join* | *LeftOuterJoin* | *NestedLeftOuterJoin*) *Operator*, *Operator*
CartProd	::= $prod$
Join	::= $join_\rho$
LeftOuterJoin	::= $ojoin^l_\rho$
NestedLeftOuterJoin	::= $nojoin^l_\rho$

Fig. 17. XML algebraic plan grammar.

APPENDIX B
XML ALGEBRA OPERATORS

This section provides details about algebraic operators used by the algebra considered in this work. In the following, we denote by $\mathcal{F}$ the domain of XML forests, and we denote by $\mathcal{T}$ the domain of tuples.

B.1 Border operators

XML Construction ($constr_L$). The input to the operator is a collection of tuples, and from each tuple an XML forest is created: $constr_L : \mathcal{T}^* \rightarrow \mathcal{F}^*$.

The information on how to build the XML forest is specified by a list L of *construction tree patterns* (CTPs in short), attached to the $constr$ operator. *For each tuple* in its input, $constr_L$ builds *one XML tree for each CTP in* L [31].

Formally, Construction Tree Patterns are defined as follows.

Definition 5 (Construction Tree Pattern): A Construction Tree Pattern is a tree $c = (V, E)$ such that each node $n \in V$ is labeled with (i) a valid XML element or attribute name, *or* (ii) a variable path p, which is $\$V_1.p'$ where p' is in turn a variable path.

If a node n is labeled with a variable path p, and a descendant n_{desc} of n is annotated with a variable path p_{desc}, then p is a prefix of p_{desc}.

Finally, we depict an *optional* construction subtree in c with a dashed edge. ◇

Without loss of generality, we will assume from now on that in all CTPs, the paths are valid wrt the schema of tuples in the input to the $constr$ operator.

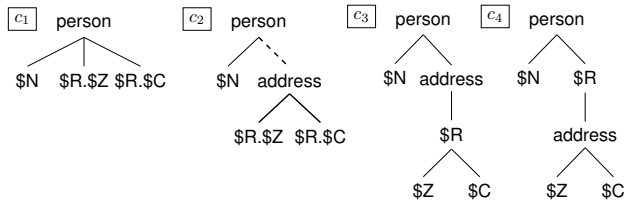
The semantics of $constr_L$ for an input collection of tuples T and a list of CTPs L is depicted in Algorithm 1. We use an XML forest f, initially empty, to gather the resulting XML. XML content is built out of each tuple $t \in T$ during a top-down, left-to right traversal of each CTP $c \in L$; $constr$ is called recursively following this order (lines 5-12). Observe that if an intermediary node in c is labeled with a variable path p, p is followed to extract a nested collection of tuples within t, which is in turn used as input for the subsequent $constr$ call (line 7). Thus, we can navigate over the nested collection of tuples to build the construction results. Afterwards, XML content for the current node r and its children (if any) is created and appended to f iff (i) r is not the root of an optional subtree, or (ii) r is the root of an optional subtree but following any variable path p_r labeling a leaf under r leads to a non-$\perp$ value within t (lines 13-24).

Algorithm 1: XML Construction

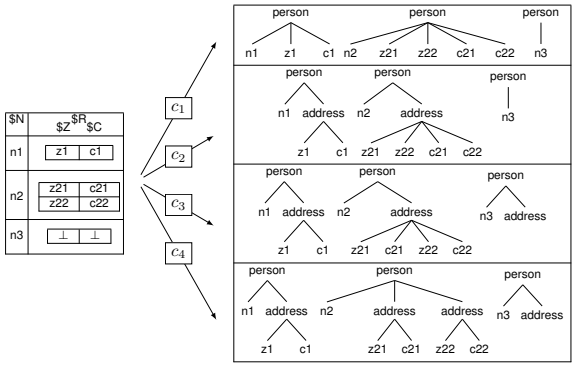
```
   Input  : Collection of tuples T, list of CTPs L
   Output: XML forest
1  f ← ()
2  for t ∈ T do
3      for c ∈ L do
4          r ← c.root; N ← children(r); f_n ← ()
5          for n ∈ N do
6              if n is not a leaf and n is labeled with a variable
               path p then
7                  T' ← collection of tuples obtained by
                   following p within t
8              else
9                  T' ← T
10             L' ← children(n)
11             f' ← constr_{L'}(T')
12             append f' to f_n
13         if r is not optional or any variable path p_r labeling a leaf
           under r leads to a non-⊥ value within t then
14             f_r ← ()
15             if r is labeled with an element (resp. attribute) name
               l then
16                 f_r ← new element/attribute labeled l
17             else
18                 if r is a leaf labeled with variable path p then
19                     f_r ← t_|p
20             if f_r is not () then
21                 add f_n as child of f_r
22             else
23                 f_r ← f_n
24             append f_r to f
25 output f; exit
```

In Figure 18a, we show four CTPs $c_1, \ldots, c_4$, while Figure 18b shows three nested tuples and the four different XML forests produced out of these three tuples by the operator $constr_{c_i, 1 \leq i \leq 4}$. Regardless of the construction pattern used, there are three trees in the forest, each built from one input tuple. The root of each tree is a newly created node labeled *person*, as dictated by each of the four c_is. Further, in each tree of the forest built for c_1, the children of the *person* node are deep copies[1] of the forests found in the $\$N$ attribute, respectively, in the nested $\$R.\$Z$ and $\$R.\$C$ attributes. Since in the third tuple the latter forests are empty, the third tree in the forest of c_1 only has a copy of n_3 as child. Observe that the same happens for c_2, as the subtree rooted at *address* is optional and thus it is only built if $\$R.\$Z$ or $\$R.\$C$ are not bound to $\perp$. Finally, note that when the CTP c_4 is used,

1. Following standard XQuery semantics [40], whenever an input node needs to be output under a new parent, a deep copy of the input node is used.

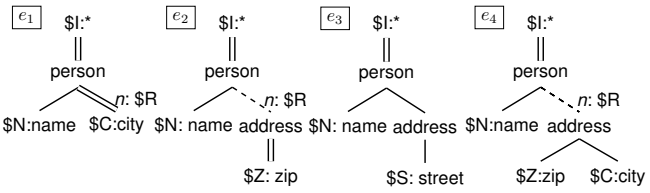


(a) Sample CTPs.

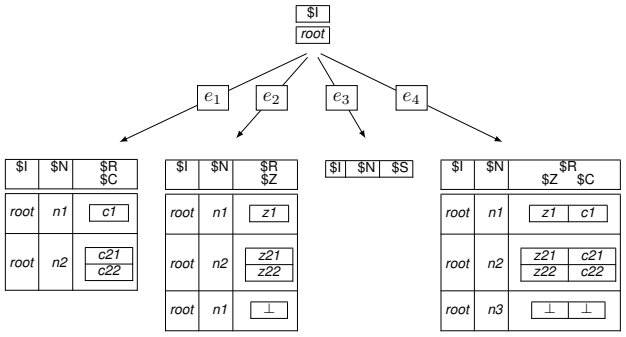


(b) Resulting XML forests after applying the CTPs to a collection of tuples.

Fig. 18. Sample CTPs and corresponding XML construction results.



(a) Sample ETPs.



(b) Resulting tuples after applying ETPs to a given tuple.

Fig. 19. Sample ETPs and corresponding navigation results.

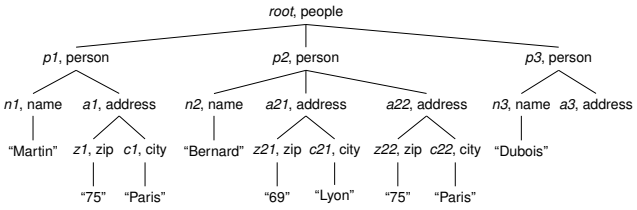


Fig. 20. Sample XML tree.

the root node in each tree has as children (copies of) the $\$N$ nodes, as well as a newly created *address* node having the $\$Z$ and $\$C$ forests as children.

Scan ($scan$). The scan operator takes as input an XML forest and creates a tuple out of each tree in the forest: $scan : \mathcal{F} \rightarrow \mathcal{T}^*$. The semantics of the scan operator whose input is an XML forest f is the following:

$$scan(f) = \{\{\langle(\$I, d.root)\rangle \mid d \in f\}\}$$

B.2 Unary operators

Navigation (nav_e). The operator is applied on a set of tuples and is parameterized by an *extended tree pattern* (ETP) [31].

In the following we introduce ETPs formally.

Definition 6 (Extended Tree Pattern): An Extended Tree Pattern is a tree $e = (V, E)$ where:

- The root $r \in V$ is labeled with a variable $\$I$.
- Each non-root node $n \in V$ are labeled with (i) an element/attribute name and (ii) optionally, a variable $\$V$.
- Each $e = (x, y) \in E$ is either a *child* edge from x to y, denoted by a single line, or a *descendant* edge from x to y, denoted by a double line. Further, *optional* edges are depicted with dashed lines, and *nested* edges are labeled with n. ◇

Figure 19a depicts some sample extended tree patterns.

Given an ETP e and an XML tree d, an *embedding* generates the tuple that results from binding the root variable of e to d and mapping the nodes of e to a collection of nodes in d. The variables of the binding tuples are ordered by the preorder traversal sequence of e. Note that if e contains *optional* edges, a mapping may be partial: nodes connected to the pattern by these edges may not be mapped, in which case the node takes the $\perp$ value.

We denote by $\varphi(e, d)$ all the embeddings from e to d. Then, we define the semantics of the navigation operator as:

$$nav_e(\mathcal{A}) = \bigcup_{t \in \mathcal{A}} \{\{t + t' \mid t' \in \varphi(e, t.\$I)\}\}$$

In other words, the navigation operator nav is parameterized by a tree pattern e, whose root is labeled with a variable $\$I$, that must appear in tuples returned by the input expression $\mathcal{A}$. The nav operator concatenates t successively with all tuples binding returned by $\varphi(e, t.\$I)$, for any tuple t returned by $\mathcal{A}$.

The semantics of the operator are illustrated with four examples in Figure 19b. Given a tuple with a variable $\$I$ bound to the XML tree shown in Figure 20, the navigation operator using e_1 extracts the *name* and *city* nodes of each person; observe that the variable $\$C$ is nested in $\$R$ and that the person without any *city* node does not generate any bindings. Instead, the navigation operator using e_4 generates bindings from all *person* nodes, as the subtree rooted at the *address* node is optional. The navigation result for ETPs e_2, e_3 is extracted in the similar fashion.

Group-By ($grp_{G_{id},G_v,\$r}$). The operator has three parameters: the set of group-by-id variables G_{id}, the set of group-by-value variables G_v and the result variable $\$r$.

Let $P(\mathcal{A}, G_{id}, G_v)$ be the set of tuple collections that results from partitioning the tuples output by $\mathcal{A}$, such that the tuples in a collection have id-equal values for the variables of G_{id} and equal values for the variables of G_v. For each collection

$p \in P(\mathcal{A}, G_{id}, G_v)$, let t^p_{id} (respectively, t^p_v) be the tuple consisting of the G_{id} (respectively, G_v) variables together with their values in p. Then, the semantics of group-by operator is defined as follows.

$$grp_{G_{id},G_v,\$r}(\mathcal{A}) = \{\{t^p_{id}+t^p_v+\langle(\$r, p)\rangle \mid p \in P(\mathcal{A}, G_{id}, G_v)\}\}$$

Each tuple in the output of grp contains: the variables of G_{id} and G_v with their values, and a newly introduced variable $\$r$, whose value is the group of input tuples whose G_{id} attributes are ID-equal, and whose G_v values are equal.

Flatten ($flat_p$). This operator unnests tuples in the collection referenced by p.

In the following, we introduce the semantics when $p.length = 1$; the other cases can be easily worked out using the same approach. For each input tuple $t \in \mathcal{A}$, let t' (respectively, t'') be the tuple containing the variables preceding (respectively, succeeding) p in t. Further, let t_i be each of tuples contained in the collection $t_{|p}$. Then, we formalize the semantics of the operator as follows:

$$flat_p(\mathcal{A}) = \{\{t' + t^p_i + t'' \mid t' + \langle(p, t_{|p})\rangle + t'' \in \mathcal{A} \text{ and } t^p_i \in t_{|p}\}\}$$

Selection (sel_ρ). The selection operator is defined in the usual way based on a boolean predicate ρ to be tested on a tuple t. Formally, a selection over a stream of tuples generated by $\mathcal{A}$ is defined as:

$$sel_\rho(\mathcal{A}) = \{\{t \mid t \in \mathcal{A} \text{ and } \rho(t) \text{ holds}\}\}$$

Projection ($proj_V$). The operator is defined by specifying a set of variable names $V = \{\$V_1, \$V_2, \ldots, \$V_k\}$ that are present in the schema of the input tuples and should be retained in the output tuples. More precisely:

$$proj_V(\mathcal{A}) = \{\{\langle(\$V_1, v_1), (\$V_2, v_2), \ldots, (\$V_k, v_k)\rangle \mid t \in \mathcal{A} \text{ and } \forall j \in \{1..k\}.\$V_j \in V \text{ and } (\$V_j, v_j) \in t\}\}$$

Aggregation ($agg_{p,a,\$r}$). The operator has three parameters: the variable path p that references the variable over whose bound values we will execute the aggregation, the aggregation operation a (recall that we support *count*, *avg*, *max*, *min* and *sum*), and the result variable $\$r$.

Let $A(t, p, a)$ be the result of applying aggregation operation a on the values bound to variable path p in tuple t.

If the path p refers to a variable in a immediate nested collection, i.e. $p.length = 2$, the semantics of the aggregation operator are defined as follows.

$$agg_{p,a,\$r}(\mathcal{A}) = \{\{t + t' \mid t \in \mathcal{A} \text{ and } t' = \langle(\$r, A(t, p, a))\rangle\}\}$$

The semantics of the aggregation with more levels of nesting, i.e. $p.length > 2$, is straightforward.

Finally, if we want to aggregate over a non-nested variable of the input tuples, i.e. $p.length = 1$, we proceed by nesting them under a new variable to produce the correct aggregation result. Thus, the semantics is defined as follows.

$$agg_{p,a,\$r}(\mathcal{A}) = \{\{agg_{p,a,\$r}(\{t\}) \mid t = \langle(\$N, (\mathcal{A}))\rangle\}\}$$

Duplicate elimination ($dupelim_V$). The operator is defined by specifying a set of variable names V that are present in the schema of the input tuples and whose bound value should be unique among the output tuples.

Recall $P(\mathcal{A}, G_{id}, G_v)$ that partitions the tuples output by $\mathcal{A}$, such that the tuples in a collection have id-equal values for the variables of G_{id} and equal values for the variables of G_v. Then, the semantics of the duplicate elimination operator is defined as follows.

$$dupelim_V(\mathcal{A}) = \{t_1 \mid \{\{t_1, \ldots, t_n\}\} \in P(\mathcal{A}, (), V)\}$$

B.3 Binary operators

The last family of operators are binary: they take in input two collections of tuples produced by the plans $\mathcal{A}_1$ and $\mathcal{A}_2$ respectively, and output a collection of tuples. We outline their semantics in the following.

Cartesian product ($prod$). The cartesian product is standard.

$$prod(\mathcal{A}_1, \mathcal{A}_2) = \{\{t_1+t_2 \mid t_1 \in \mathcal{A}_1, t_2 \in \mathcal{A}_2\}\}$$

Join ($join_\rho$). The join relies on a boolean join predicate $\rho(t_1, t_2)$, and is defined as follows.

$$join_\rho(\mathcal{A}_1, \mathcal{A}_2) = \{\{t_1+t_2 \mid t_1 \in \mathcal{A}_1, t_2 \in \mathcal{A}_2 \text{ and } \rho(t_1, t_2) \text{ holds}\}\}$$

As stated previously, the join predicate is expressed in disjunctive normal form (DNF).

Left outer join ($ojoin^l_\rho$). Given two streams of tuples produced by $\mathcal{A}_1, \mathcal{A}_2$ and a DNF predicate ρ, $ojoin^l_\rho(\mathcal{A}_1, \mathcal{A}_2)$ returns the pairs of tuples satisfying ρ, plus the tuples from the left input without a matching right tuple. Its semantics are defined as follows:

$$ojoin^l_\rho(\mathcal{A}_1, \mathcal{A}_2) = \{\{t_1+t_2 \mid t_1 \in \mathcal{A}_1, t_2 \in \mathcal{A}_2 \text{ and } \rho(t_1, t_2)\}\} \cup \{\{t_1 + \perp_{\mathcal{A}_2} \mid t_1 \in \mathcal{A}_1, \nexists t_2 \in \mathcal{A}_2 \text{ s.t. } \rho(t_1, t_2)\}\}$$

where $\perp_{\mathcal{A}_2}$ is a tuple having the schema of the tuples in $\mathcal{A}_2$ and $\perp$ values bound to its variables. As customary of left outer joins, the left tuples without a matching right tuple are concatenated to $\perp_{\mathcal{A}_2}$.

Nested left outer join ($nojoin^l_\rho$). The operator semantics are defined as:

$$nojoin^l_\rho(\mathcal{A}_1, \mathcal{A}_2) = \{\{t_1 + \langle(\$r, (t_{21}, t_{22}, \ldots, t_{2n}))\rangle \mid t_1 \in \mathcal{A}_1 \text{ and } t_{21}, t_{22}, \ldots, t_{2n} \in \mathcal{A}_2 \text{ and } \forall k \in \{1..n\},\ \rho(t_1, t_{2k}) \text{ holds } \}\} \cup \{\{t_1 + \langle(\$r, \perp_{\mathcal{A}_2})\rangle \mid t_1 \in \mathcal{A}_1, \nexists t_2 \in \mathcal{A}_2 \text{ such that } \rho(t_1, t_2) \text{ holds}\}\}$$

Thus, each tuple from the left input is paired with a new nested variable $\$r$, encapsulating all the matching tuples from the right-hand input. If the left tuple does not have a matching right tuple, $\$r$ must contain a tuple $\perp_{\mathcal{A}_2}$.

APPENDIX C
DISJUNCTIVE EQUI-JOINS POST-PROCESSING

The semantics of these UFs are introduced in the following.

Definition 7 ($\overline{opost}_{l\vee}$): Consider an input bag of records $\{\{r_1, \ldots, r_x\}\}$. Each record $r_{i,1\leq i\leq x}$ is separated in left and right side, i.e. $r_i = r^l_i + r^r_i$. Further, r^r_i contains a single field with a nested collection of records R_i. We denote by $\overline{opost}_{l\vee}$ the post-processing UF which:

- If all nested collections $R_{i,1\leq i\leq x}$ contain only $\perp$-records, it outputs a single record $r = r^l_i + \perp'$, where r^l_i is the left side of any input record and $\perp'$ is the $\perp$-record conforming to the signature of the records in R_i.
- Otherwise, it flattens the nested collections $R_{i,1\leq i\leq x}$ excluding $\perp$-records, and returns the result. $\diamond$

$$\frac{\mathcal{A}_1 \Rightarrow \mathcal{P}_1 \quad \mathcal{A}_2 \Rightarrow \mathcal{P}_2 \quad S_{\mathcal{A}_1}, S_{\mathcal{A}_2}; \rho \mapsto \rho' \quad f := \overline{pnjoin}(\rho')}{join_\rho(\mathcal{A}_1, \mathcal{A}_2) \Rightarrow \mathtt{cr}_f(\mathcal{P}_1, \mathcal{P}_2)} \quad \text{(INEQUI-JOIN)}$$

$$\frac{\mathcal{A}_1 \Rightarrow \mathcal{P}_1 \quad \mathcal{A}_2 \Rightarrow \mathcal{P}_2 \quad S_{\mathcal{A}_1}, S_{\mathcal{A}_2}; \rho \mapsto \rho' \quad f := \overline{ojoin}_l(\rho') \quad S_{\mathcal{A}_1} \rightsquigarrow K \quad f' := \overline{opost}_l}{ojoin^l_\rho(\mathcal{A}_1, \mathcal{A}_2) \Rightarrow \mathtt{rd}^K_{f'}(\mathtt{cr}_f(\mathcal{P}_1, \mathcal{P}_2))} \quad \text{(LO INEQUI-JOIN)}$$

$$\frac{\mathcal{A}_1 \Rightarrow \mathcal{P}_1 \quad \mathcal{A}_2 \Rightarrow \mathcal{P}_2 \quad S_{\mathcal{A}_1}, S_{\mathcal{A}_2}; \rho \mapsto \rho' \quad f := \overline{ojoin}_l(\rho') \quad S_{\mathcal{A}_1} \rightsquigarrow K \quad f' := \overline{nopost}_l}{nojoin^l_\rho(\mathcal{A}_1, \mathcal{A}_2) \Rightarrow \mathtt{rd}^K_{f'}(\mathtt{cr}_f(\mathcal{P}_1, \mathcal{P}_2))} \quad \text{(NLO INEQUI-JOIN)}$$

Fig. 21. Inequi-join translation rules.

Definition 8 ($\overline{nopost}_{l\vee}$): Consider an input bag of records $\{\{r_1, \ldots, r_x\}\}$. Each record $r_{i,1\leq i\leq x}$ is separated in left and right side, i.e. $r_i = r_i^l{+}r_i^r$. Further, r_i^r contains a single field with a nested collection of records R_i. We denote by $\overline{nopost}_{l\vee}$ the post-processing UF that outputs a single record $r = r_i^l{+}r'$, where:

- If all nested collections $R_{i,1\leq i\leq x}$ contain only $\perp$-records, r' contains a field with a nested collection with a $\perp$-record conforming to the signature of the records in R_i.
- Otherwise, r' contains a field with a nested collection with the records contained in $R_{i,1\leq i\leq x}$, excluding $\perp$-records. ◇

APPENDIX D
JOINS ON INEQUALITIES

Our XQuery fragment also supports joins with inequality conditions. In this case, the translation uses *Cross* contracts. Further, just like for joins with disjunctive predicates, the non-nested and nested outer variants of the inequi-join require more than one PACT. We depict the corresponding translation rules in Figure 21. In the following, we explain the translation of this flavor of joins.

1) Inequi-join. Rule (INEQUI-JOIN) generates a PACT with a *Cross* input contract. The predicate ρ is transformed into ρ', which is equivalent but replaces the EXDM variables by positions in the PACTs records. Then the $\overline{pnjoin}$ UF introduced in the following is applied over each pair of records.

Definition 9 ($\overline{pnjoin}$): Given two records r_1, r_2 and a predicate ρ', the $\overline{pnjoin}(\rho')$ UF evaluates ρ' over r_1, r_2, and outputs $r_1{+}r_2$ if it evaluates to true. ◇

2) (Nested) left outer inequi-join. As it happens with the disjunctive equality predicates, the translation of the non-nested and nested variant of the outer inequi-join, described by the (LO INEQUI-JOIN) and (NLO INEQUI-JOIN) rules respectively, resemble each other.

The translation of the non-nested and nested left outer inequi-join results in two steps. The first step consists of a PACT with a *Cross* contract. The UF of the PACT is $\overline{ojoin}_l$, a traditional left outer join, that we introduced in the following.

Definition 10 ($\overline{ojoin}_l$): Given two records r_1, r_2 and a predicate ρ', the $\overline{ojoin}_l(\rho')$ UF evaluates ρ' over r_1, r_2, and:

- If it evaluates to *true*, outputs $r_1{+}r_2$.
- Otherwise, it outputs $r_1{+}\perp_2$, where $\perp_2$ is the $\perp$-record that conforms to the signature of r_2. ◇

The last PACT resulting from both translation rules uses a *Reduce* contract that groups together the records originated from the same left hand-side record. In the plain variant, the UF is $\overline{opost}_l$ that produces unnested records; otherwise, the PACT uses the $\overline{nopost}_l$ UF. We introduce both UFs in the following.

Definition 11 ($\overline{opost}_l$): Consider an input bag of records $\{\{r_1, \ldots, r_x\}\}$. Each record $r_{i,1\leq i\leq x}$ is separated in left and right side, i.e. $r_i = r_i^l{+}r_i^r$. We denote by $\overline{opost}_l$ the post-processing UF which:

- If $r^r_{i,1\leq i\leq x}$ are all $\perp$-records, it outputs one of them.
- Otherwise, it returns every $r_{i,1\leq i\leq x}$ where r_i^r is not a $\perp$-record. ◇

Definition 12 ($\overline{nopost}_l$): Consider an input bag of records $\{\{r_1, \ldots, r_x\}\}$. Each record $r_{i,1\leq i\leq x}$ is separated in left and right side, i.e. $r_i = r_i^l{+}r_i^r$. We denote by $\overline{nopost}_l$ the post-processing UF that outputs a single record $r = r_i^l{+}r'$, where:

- If $r^r_{i,1\leq i\leq x}$ are all $\perp$-records, r' contains a field with a nested collection with a $\perp$-record that conforms to the signature of r_i^r.
- Otherwise, r' contains a field with a nested collection with every $r^r_{i,1\leq i\leq x}$ that is not a $\perp$-record. ◇

Example 5. Consider the following XQuery that extracts the name of users and (if any) the items they bought that were valued more than their monthly incoming:

```
let $pc := collection('people'),
 $cc := collection('closed_auctions')
for $p in $pc/site/people/person
let $n := $p/name
let $r :=
  for $c in $cc//closed_auction, $i in $p/@id,
   $b in $c/buyer/@person, $x in $p/profile/@income,
   $y in $c/price
  let $a := $c/itemref
  where $i = $b and $x < $y
  return $a
return <res>{$n,$r}</res>
```

The XML algebra expression generated from this query is shown in Figure 22a. Using the rule (LO $\vee$ INEQUI-JOIN) in Figure 21, the algebraic expression corresponding to the query is translated into the PACT plan depicted in Figure 22b. ◇

APPENDIX E
EXPERIMENTAL QUERIES DETAIL

This section lists the XQuery queries used in the experimental section. They are based on the queries provided by the XMark benchmark [42].

Query 1: Return the name of the person with ID 'person0'.

```
let $pc := collection('XMarkPeople')
for $p in $pc/site/people/person[@id="person0"]
let $n := $p/name/text()
return $n
```

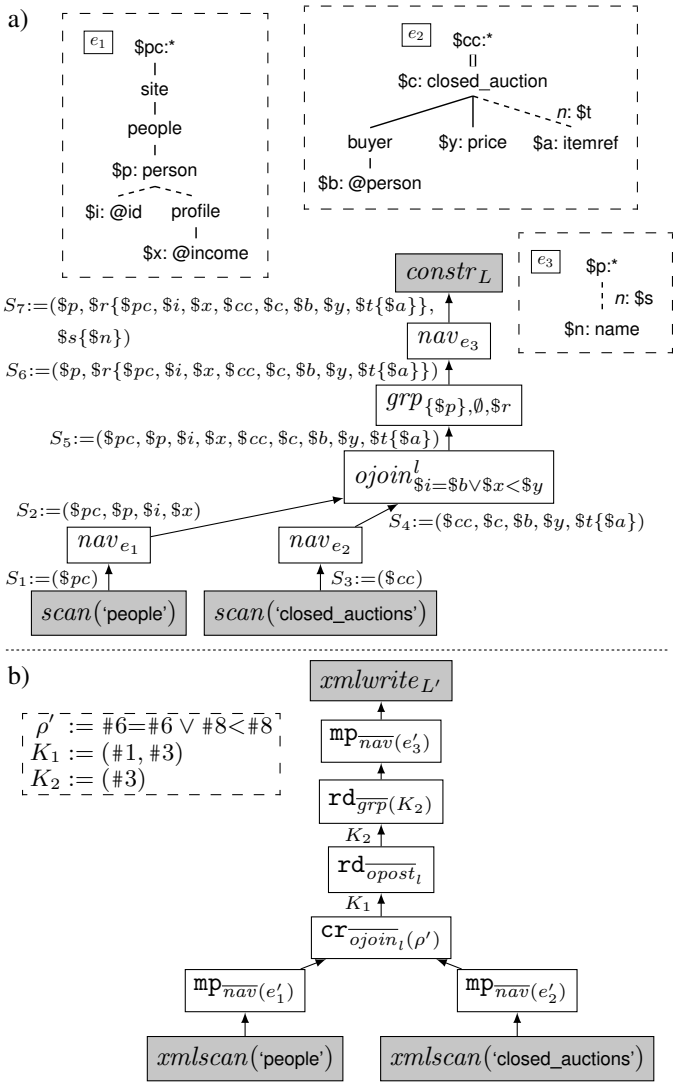


Fig. 22. Logical expression (a) and corresponding PACT plan (b) for the query in Example 5.

Query 2: List the names of items registered in Australia along with their descriptions.

```
let $ic := collection('XMarkItems')
for $i in $ic/site/regions/australia/item
let $n := $i/name/text(), $d := $i/description
return <item name="{$n}">{$d}</item>
```

Query 3: Return the names of all items in Europe whose description contains the word 'gold'.

```
let $ic := collection('XMarkItems')
for $i in $ic/site//europe/item,
    $d in $i/description/text/text()
let $n := $i/name/text()
where contains($d, "gold")
return $n
```

Query 4: Print the keywords in emphasis in annotations of closed auctions.

```
let $cc := collection('XMarkClosedAuctions')
for $a in
  $cc/site/closed_auctions/closed_auction/
    annotation/description/parlist/listitem/
    parlist/listitem/text/emph/keyword/text()
return <text>{$a}</text>
```

Query 5: Return the IDs of those auctions that have one or more keywords in emphasis.

```
let $cc := collection('XMarkClosedAuctions')
for $a in
  $cc/site/closed_auctions/closed_auction
for $k in $a/annotation/description/parlist/
  listitem/parlist/listitem/text/emph/keyword/text()
let $s := $a/seller/@person
where not(empty($k))
return <person id="{$s}"/>
```

Query 6: Which persons have a homepage?

```
let $pc := collection('XMarkPeople')
for $p in $pc/site/people/person
let $h := $p/homepage,
    $n := $p/name/text()
where not(empty($h))
return <person name="{$n}"/>
```

Query 7: How many sold items cost more than 40?

```
let $cc := collection('XMarkClosedAuctions')
let $p :=
  for $i in $cc/site/closed_auctions/closed_auction
  where $i/price/text() >= 40
  return $i/price
let $c := count($p)
return $c
```

Query 8: How many items are listed on all continents?

```
let $ic := collection('XMarkItems')
let $i := $ic/site/regions//item
let $c := count($i)
return $c
```

Query 9: List the number of buyers per city of France.

```
let $pc := collection('XMarkPeople'),
    $cc := collection('XMarkClosedAuctions')
for $p in $pc/site/people/person
                        [address/country/text()='France']
let $a := $p/address/city/text()
for $c in $cc/site/closed_auctions/closed_auction,
    $i in $p/@id, $b in $c/buyer/@person
where $i = $b
group by $a
return <res><city>{$a}</city>
  <num>{count($p)}</num></res>
```

Query 10: List the names of persons and the names of the items they bought in Europe.

```
let $pc := collection('XMarkPeople'),
    $cc := collection('XMarkClosedAuctions'),
    $ic := collection('XMarkItems')
let $ca := $cc/site/closed_auctions/closed_auction,
    $ei := $ic/site/regions/europe/item
for $p in $pc/site/people/person
let $pn := $p/name/text()
let $a :=
  for $t in $ca, $i in $p/@id,
      $b in $t/buyer/@person
  where $i = $b
  return
    let $n :=
      for $t2 in $ei, $ti2 in $t2/@id,
          $ti in $t/itemref/@item
      where $ti = $ti2
      return $t2
    let $in := $n/name/text()
    return <item>{$in}</item>
return <person name="{$pn}">{$a}</person>
```

Query 11: List all persons according to their interest; use French markup in the result.

```
let $pc := collection('XMarkPeople')
for $i in distinct-values($pc/site/people/person/
                          profile/interest/@category)
let $p :=
  for $t in $pc/site/people/person,
      $c in $t/profile/interest/@category
  let $r1 := $t/profile/gender/text(),
      $r2 := $t/profile/age/text(),
      $r3 := $t/profile/education/text(),
      $r4 := $t/profile/@income,
      $r5 := $t/name/text(),
      $r6 := $t/address/street/text(),
      $r7 := $t/address/city/text(),
      $r8 := $t/address/country/text(),
      $r9 := $t/emailaddress/text(),
      $r10 := $t/homepage/text(),
      $r11 := $t/creditcard/text()
  where $c = $i
  return
  <personne>
    <statistiques>
      <sexe>{$r1}</sexe>
      <age>{$r2}</age>
      <education>{$r3}</education>
      <revenu>{$r4}</revenu>
    </statistiques>
    <coordonnees>
      <nom>{$r5}</nom>
      <rue>{$r6}</rue>
      <ville>{$r7}</ville>
      <pays>{$r8}</pays>
      <reseau>
        <courrier>{$r9}</courrier>
        <pagePerso>{$r10}</pagePerso>
      </reseau>
    </coordonnees>
    <cartePaiement>{$r11}</cartePaiement>
  </personne>
return <categorie><id>{$i}</id>{$p}</categorie>
```

Query 12: List the names of persons and the number of items they bought.

```
let $pc := collection('XMarkPeople'),
    $cc := collection('XMarkClosedAuctions')
for $p in $pc/site/people/person
let $n := $p/name/text()
let $a :=
  for $t in $cc/site/closed_auctions/closed_auction,
      $b in $t/buyer/@person, $i in $p/@id
  where $b = $i
  return $t
let $c := count($a)
return <item person="{$n}">{$c}</item>
```

Query 13: List the name of users in France and the items that they bought or sold in an auction.

```
let $pc := collection('XMarkPeople'),
    $cc := collection('XMarkClosedAuctions')
for $p in $pc/site/people/person,
    $i in $p/@id,
    $ad in $p/address/country/text()
let $a :=
  for $c in $cc//closed_auction,
      $b in $c/buyer/@person,
      $s in $c/seller/@person
  let $ir := $c/itemref
  where $i = $b or $i = $s
  return $ir
let $n := $p/name
where $ad = 'France'
return <res>{$n,$a}</res>
```

Query 14: For each rich person, list the number of cars-related items currently on sale whose price does not exceed 0.02% of the person's income.

```
let $pc := collection('XMarkPeople'),
    $oc := collection('XMarkOpenAuctions')
for $p in $pc/site/people/person
let $l :=
  for $o in $oc/site/open_auctions/open_auction,
      $i in $o/initial/text(),
      $si in $p/profile/@income,
      $a in $o/annotation//text/text()
  let $x := 5000*$i
  where $si > $x and contains($a,"car")
  return $i
for $li in $p/profile/@income
let $n := count($l)
where $li > 200000
return <items person="{$li}">{$n}</items>
```