XML Query Optimization in Map-Reduce

Leonidas Fegaras Chengkai Li Upa Gupta Jijo J. Philip

University of Texas at Arlington, CSE
Arlington, TX 76019
{fegaras,cli,upa.gupta,jijo.philip}@uta.edu

ABSTRACT

We present a novel query language for large-scale analysis of XML data on a map-reduce environment, called MRQL, that is expressive enough to capture most common data analysis tasks and at the same time is amenable to optimization. Our evaluation plans are constructed using a small number of higher-order physical operators that are directly implementable on existing map-reduce systems, such as Hadoop. We report on a prototype system implementation and we show some preliminary results on evaluating MRQL queries on a small cluster of PCs running Hadoop.

1. INTRODUCTION

Many web service providers are facing the challenge of collecting and analyzing massive amounts of data, such as data collected by web crawlers, search logs, web logs, and streams of ad-click data. Often, these data come in the form of XML, such as the mediawiki dumps of Wikipedia articles. By analyzing these data, these companies gain a competitive edge by improving their web services, providing better ad selection, detecting fraudulent activities, and enabling data mining on large scale. The map-reduce programming model [8] is an emerging framework for cloud computing that enables this data analysis. It facilitates the parallel execution of ad-hoc, long-running large-scale data analysis tasks on a shared-nothing cluster of commodity computers connected through a high-speed network. In contrast to parallel databases, which require the programmer to first model and load the data before processing, the map-reduce model is better suited to one-time ad-hoc queries over write-once raw data. More importantly, compared to traditional DBMSs, map-reduce implementations offer better fault tolerance and the ability to operate in heterogeneous environments, which are critical for large scale data analysis on commodity hardware.

When defining a map-reduce job, one has to specify a map and a reduce task, which may be arbitrary computations written in a general-purpose language [8]. The map task specifies how to process a single key/value pair to generate a set of intermediate key/value pairs. The reduce task specifies how to merge all intermediate values associated with the same intermediate key. A map-reduce computation uses the map task to process all input key/value pairs in parallel by distributing the data among a number of nodes in the cluster (called the map workers), which execute the map task in parallel without communicating with each other. Then, the map results are repartitioned across a number of nodes (called the reduce workers) so that values associated with the same key are grouped and processed by the same node. Finally, each reduce worker applies the reduce task to its assigned partition.

Our goal was to design and implement an effective cost-based optimization framework for the map-reduce programming environment that improves large-scale data analysis programs over raw data, especially XML documents. We believe that it would be very hard to optimize general map-reduce programs expressed in a general-purpose programming language. Instead, as it is evident from the success of the relational database technology, program optimization would be more effective if the programs were written in a higher-level query language that hides the implementation details and is amenable to optimization. Therefore, one of our goals was to design a declarative query language that is powerful enough to capture most commonly used map-reduce computations, is easy to learn, has uniform syntax, is extensible, has simple semantics, and is easy to compile to efficient map-reduce programs. On one hand, we would like to have a declarative query language, powerful enough to avert the programmer from using ad-hoc map-reduce programs, which may result to suboptimal, error-prone, and hard to maintain code. On the other hand, we want to be able to optimize this query language, leveraging the relational query optimization technology. Unfortunately, relational query optimization techniques are not directly applicable to the map-reduce environment. Consider, for example, the following nested SQL query:

```
select * from X x
where x.D > (select sum(y.C) from Y y where x.A=y.B)
```

A typical method for evaluating this query in current DBMSs is to do a left-outer join between X and Y on x.A=y.B (it is a left-outer join because the query must also return the x tuples that are not joined with any y tuple), to group the result by the x key, and, for each group, to calculate the sum of all y.C and compare this sum with x.D. Unfortunately, this method is suboptimal in a map-reduce environment because it requires two map-reduce jobs, one for the join and one for the group-by. Instead, this query can be evaluated with one reduce-side join [24] (a partitioned join), which requires only one map-reduce job. Consequently, optimizing nested queries requires special techniques that take advantage of the special algorithms available in a map-reduce environment. Nested queries are very important because any arbitrary map-reduce computation can be expressed declaratively using nested queries, as we will show in Section 5. Capturing all map-reduce computations as simple queries was a very important design goal for our framework, since

Fourteenth International Workshop on the Web and Databases (WebDB 2011), June 12, 2011 - Athens, Greece.

it obviates the need for introducing a special map-reduce operation. Another important goal was to develop an optimizer that is able to recognize most syntactic forms in the query syntax that are equivalent to a map-reduce operation and derive a single map-reduce job for each such form. If neither of these two goals is achieved, a programmer may be forced to use explicit map-reduce computations, rather than declarative queries, which may result to suboptimal code.

We are presenting a novel framework for optimizing and evaluating map-reduce computations over XML data that are expressed in an SQL-like query language, called MRQL (the Map-Reduce Query Language). This language is powerful enough to express most common data analysis tasks over XML text documents, as well as over other forms of raw data, such as line-oriented text documents with comma-separated values. Although we have extensive experience in building XQuery optimizers, we decided to design our own query language because we are planning to extend MRQL to handle many other forms of raw data, such as JSON data, as well as structured data, such as relational databases and key-value maps, in the same query language. To evaluate MRQL queries, we provide a set of physical plan operators, such as the reduce-side join, that are directly implementable on existing map-reduce systems. Leveraging the research work in relational databases, our system compiles MRQL queries to an algebra, which is translated to physical plans using cost-based optimizations. Due to space limitations, this paper describes the XML fragmentation techniques used by our framework to break XML data into manageable fragments that are ready for map-reduce evaluation (Section 3), some of the MRQL syntax to query XML data (Section 4), and the physical operators used by our framework to evaluate queries (Section 5). The query algebra and optimizer are briefly sketched in Section 6. They will be described in detail in a forthcoming paper.

2. RELATED WORK

The map-reduce model was introduced by Google in 2004 [8]. Several large organizations have implemented the map-reduce paradigm, including Apache Hadoop [24] and Pig [20], Apache/-Facebook Hive [22], Google Sawzall [21], and Microsoft Dryad [14]. The most popular map-reduce implementation is Hadoop [13], an open-source project developed by Apache, which is used today by Yahoo! and many other companies to perform data analysis. There are also a number of higher-level languages that make map-reduce programming easier, such as HiveQL [22], PigLatin [20], Scope [6], and Dryad/Linq [15]. Hive [22, 23] is an open-source project by Facebook that provides a logical RDBMS environment on top of the map-reduce engine, well-suited for data warehousing. Using its high-level query language, HiveQL, users can write declarative queries, which are optimized and translated into map-reduce jobs that are executed using Hadoop. HiveQL does not handle nested collections uniformly: it uses SQL-like syntax for querying data sets but uses vector indexing for nested collections. Unlike MRQL, HiveQL has many limitations (it is a small subset of SQL) and neither does support nor optimize nested queries. Because of these limitations, HiveQL enables users to plug-in custom map-reduce scripts into queries. Although Hive uses simple rule-based optimizations to translate queries, it has yet to provide a comprehensive framework for cost-based optimizations. Yahoo!'s Pig [12] resembles Hive as it provides a user-friendly query-like language, called PigLatin [20], on top of map-reduce, which allows explicit filtering, map, join, and group-by operations. Like Hive, PigLatin performs very few optimizations based on simple rule transformations. HadoopDB [1] adapts a hybrid scheme between map-reduce and parallel databases to gain the benefit of both systems. Although, HadoopDB uses Hive as the user interface layer, instead of storing table tuples in DFS, it stores them in independent DBMSs in each physical node in the cluster. That way, it increases the speed of overall processing as it pushes many database operations into the DBMS directly, and, on the other hand, it inherits the benefits of high scalability and high fault-tolerance from the map-reduce framework. Hadoop++ [9] takes a different approach from HadoopDB: each map-reduce computation is decomposed into an execution plan, which is then transformed to take advantage of possible indexes attached to data splits. This work though does not provide a framework for recognizing joins and filtering in general map-reduce programs, in order to take advantage of the indexes. Manimal [4, 16] analyzes the actual map-reduce code to find opportunities for using B+-tree indexes, projections, and data compression. It assumes that an index is generated before query execution and is used frequently enough to justify the time overhead required to build the index. This assumption may not be valid for one-time queries against raw data. Finally, even though Hadoop provides a simple XML input format for XML fragmentation, to the best of our knowledge, there is no other system or language reported for XML query processing on a map-reduce environment. (Although there are plans for implementing XQuery on top of Hadoop, called Xadoop, by D. Kossmann's group at ETH [25].)

3. XML DATA FRAGMENTATION

A data parallel computation expects its input data to be fragmented into small manageable pieces, which determine the granularity of the computation. In a map-reduce environment, each map worker is assigned a data split that consists of data fragments. A map worker processes these data one fragment at a time. For relational data, a fragment is clearly a relational tuple. For text files, a fragment can be a single line in the file. But for hierarchical data and nested collections, such as XML data, the choice for a suitable fragment size and structure may depend on the actual application that processes these data. For example, the XML data may consist of a number of XML documents, each one containing a single XML element, whose size may exceed the memory capacity of a map worker. Consequently, when processing XML data, it would be desirable to allow custom fragmentations to suit a wide range of application needs. Hadoop provides a simple XML input format for XML fragmentation based on a single tagname. Given a data split of an XML document (which may start and end at arbitrary points in the document, even in the middle of tagnames), this input format allows us to read the document as a stream of string fragments, so that each string will contain a single complete element that has the requested tagname. Then, the programmer may use an XML parser to parse these strings and convert them to objects. The fragmentation process is complicated by the fact that the requested elements may cross data split boundaries and these data splits may reside in different data nodes in the DFS. Fortunately, this problem is implicitly solved by the Hadoop DFS by permitting to scan beyond a data split to the next, subject to some overhead for transferring data between nodes.

Our XML fragmentation technique, which was built on top of the existing Hadoop XML input format, provides a higher-level of abstraction and better customization. It is a higher-level because, instead of deriving a string for each XML element, it constructs XML data in the MRQL data model, ready to be processed by MRQL queries. In MRQL, the XML data type is actually a user-defined type based on data constructors (very similar to the data constructors in Haskell):

```
data XML = Node: ( String, list ((String,String)), list (XML) )
         | CData: String
```

That is, XML data can be constructed as nodes (which are tuples that contain a tagname, a list of attribute bindings, and a list of children) or text leaves (CData). For example, <a x="1">b</a> is constructed using Node("a",[("x","1")],[CData("b")]). The MRQL expression used for parsing an XML document is:

```
source( tags, xpath, file )
```

where tags is a bag of synchronization tags, xpath is the XPath expression used for fragmentation, and file is the document path. Given a data split from the document, this operation skips all text until it finds the opening of a synchronization tag and then stores the text upto the matching closing tag into a buffer. During the storing of an element, it may cross split boundaries, but during the skipping of text, it will stop at the end of the split. The buffer then becomes the current context for xpath, which is evaluated in stream-like fashion using SAX (based on our earlier work [11]), returning XML objects constructed in our MRQL data model.

For example, the following expression:

```
XMark = source( {"person"}, xpath (.), "xmark.xml" );
```

binds the variable XMark to the result of parsing the document *xmark.xml* (generated by the XMark benchmark [26]) and returns a list of person elements. The xpath expression here is the 'dot' that returns the current context. A more complex example is:

```
DBLP = source( {"article", " incollection ", "book","inproceedings"},
               xpath (.[year=2009]/ title ), "dblp.xml" )
```

which retrieves the titles of certain bibliography entries published in 2009 from DBLP [7]. Here, we are using multiple synchronization tags since we are interested in elements of multiple tagnames. Note that, although the document order is important for XML data, this order is ignored across fragments but is preserved within each fragment, as expected, since data splits are processed by worker nodes in parallel. MRQL also provides syntax to navigate through XML data. The projection operation $e.A$ has been overloaded to work on XML data. Given an expression e of type XML or list(XML), $e.A$ returns a list(XML) that contains the subelements of e with tagname A (much like e/A in XPath). Similarly, the syntax $e.*$, $e.@A$, and $e.@*$ corresponds to the XPaths $e/*$, $e/@A$, and $e/@*$, respectively.

4. THE MAP-REDUCE QUERY LANGUAGE

The MRQL query syntax is influenced by ODMG OQL [5], the OODB query language developed in the 90's, while its semantics has been inspired by the work in the functional programming community on list comprehensions with group-by and order-by [18]. The select-query syntax in MRQL takes the form:

select [**distinct**] e
from p_1 **in** $e_1, \ldots, p_n$ **in** e_n
[**where** e_c]
[**group by** p': e' [**having** e_h]]
[**order by** e_o]

where $e, e_1, \ldots, e_n, e_c, e', e_h$, and e_0 are arbitrary MRQL expressions, which may contain other nested select-queries. MRQL handles a number of collection types, such as lists (sequences), bags (multisets), and key-value maps. The difference between a list and a bag is that a list supports order-based operations, such as indexing. An MRQL query works on collections of values, which are treated as bags by the query, and returns a new collection of values. If it is an order-by query, the result is a list, otherwise, it is a bag. Treating collections as bags is crucial to our framework, since it allows the queries to be compiled to map-reduce programs, which need to shuffle and sort the data before reduction, and enables the use of joins for query evaluation. The **from** part of an MRQL syntax contains query bindings of the form 'p **in** e', where p is a pattern and e is an MRQL expression that returns a collection. The pattern p matches each element in the collection e, binding its pattern variables to the corresponding values in the element. In other words, this query binding specifies an iteration over the collection e, one element at a time, causing the pattern p to be matched with the current collection element. In general, a pattern can be a pattern variable that matches any data, or a tuple $(p_1, \ldots, p_n)$ or a record $<A_1 : p_1, \ldots, A_n : p_n>$ that contain patterns $p_1, \ldots, p_n$. Patterns are compiled away from queries before query optimization.

The group-by syntax of an MRQL query takes the form **group by** p': e'. It partitions the query results into groups so that the members of each group have the same e' value. The pattern p' is bound to the group-by value, which is unique for each group and is common across the group members. As a result, the group-by operation lifts all the other pattern variables defined in the from-part of the query from some type T to a bag of T, indicating that each such variable must contain multiple values, one for each group member. For example, the following query on XMark data:

Query 1:

```
select ( cat, os, count(p) )
from p in XMark,
     i in p. profile . interest
group by ( cat, os ): ( i .@category,
                        count(p.watches.@open_auctions) )
```

groups all persons according to their interests and the number of open auctions they watch. For each such group, it returns the number of persons in the group. The XMark data source returns the person elements, so that p is one person, and i is one of p's interests. The variables cat and os in the query header are directly accessible since they are group-by variables. The variable p, on the other hand, is lifted to a bag of XML elements. Thus, **count**(p) counts all persons whose interests include cat and watch os open auctions.

Finally, the '**order by**' syntax orders the result of a query (after the optional group-by) by the e_0 values. It is assumed that there is a default total order $\leq$ defined for all data types (including tuples and bags). The special parametric type Inv(T), which has a single data constructor inv(v) for a value v of type T, inverts the total order of T from $\leq$ to $\geq$. For example, as a part of a select-query

```
order by ( inv(count(p.watches.@open_auctions)), p.name )
```

orders people by major order **count**(p.watches.@open_auctions) (descending) and minor order p.name (ascending).

A more complex query, which is similar to the query Q10 of the XMark benchmark [26], is

Query 2:

```
select ( cat, count(p), select text(x.name) from x in p )
from p in XMark,
     i in p. profile . interest ,
     c in XMark
where c.@id = i.@category
group by cat: text(c.name);
```

which uses an XML source that retrieves both persons and categories:

```
XMark = source({"person","category"},xpath(.), "xmark.xml");
```

It groups persons by their interests, and for each group, it returns the category name, the number of people whose interests include this category, and the set of names of these people. The text function returns the textual content of element(s).

As yet another example over the DBLP bibliography:

```
    DBLP = source( {"article", " incollection ", "book","inproceedings"},
                   xpath (.),  "dblp.xml" )
```

the following query

Query 3:

```
    select ( select text (a. title ) from a in DBLP where a.@key = x,
             count(a) )
    from a in DBLP,
         c in a. cite
    where text(c) <> " ... "
    group by x: text (c)
    order by inv(count(a))
```

inverts the citation graph in DBLP by grouping the items by their citations and by ordering these groups by the number of citations they received. The condition text(c) <> "..." removes bogus citations. Note that, the DBLP source is also used in the query header to retrieve the citation title.

5. THE PHYSICAL OPERATORS

The MRQL physical operators form an algebra over the domain DataSet(T), which is equivalent to the type bag(T). This domain is associated with a source list, where each source consists of a file or directory name in DFS, along with an input format that allows to retrieve T elements from the data source in a stream-like fashion. The input format used for storing the intermediate results in DFS is a sequence file that contains the data in serialized form. The MRQL expression source, described in Section 3, returns a single source of type bag(XML) whose input format is an XML input format that uses synchronization tags and an XPath to extract XML fragments from XML documents. The rest of the physical operators have nothing to do with XML because they process fragments using map-reduce jobs, regardless of the fragment format. Each map-reduce operation though is parameterized by functions that are particular to the data format being processed. The code of these functional parameters is evaluated in memory (at each task worker), and therefore can be expressed in some XML algebra suitable for in-memory evaluation. Our focus here is in the map-reduce operations, which are novel, rather than in an XML algebra, which has been addressed by earlier work. In addition to the source expression, MRQL uses the following physical operators:

• Union(X, Y), returns the union of the DataSets X and Y. It simply concatenates the source lists of X and Y (the list of file names), forming a new DataSet.

• MapReduce$(m, r)\, S$, transforms a DataSet S of type bag(α) into a DataSet of type bag(β) using a map function m of type $\alpha \rightarrow$ bag((κ,γ)) and a reduce function r of type (κ,bag(γ)) $\rightarrow$ bag(β), for the arbitrary types α, β, γ, and κ. The map function m transforms values of type α from the input dataset into a bag of intermediate key-value pairs of type bag((κ,γ)). The reduce function r merges all intermediate pairs associated with the same key of type κ and produces a bag of values of type β, which are incorporated into the MapReduce result. More specifically, MapReduce$(m, r)\, S$ is equivalent to the following MRQL query:

```
    select w
    from z in (select $r$(key,y)
                  from x in $S$,
                       (k,y) in $m$(x)
                  group by key: k),
         w in z
```

that is, we apply m to each value x in S to retrieve a bag of (k,y) pairs. This bag is grouped by the key k, which lifts the variable y to a bag of values. Since each call to r generates a bag of values, the inner select creates a bag of bags, which is flattened out by the outer select query. A straightforward implementation of MapReduce$(m, r)\, S$ in a map-reduce platform, such as Hadoop, is the following Java pseudo-code:

```
    class Mapper
        method map ( key, value )
            for each $(k, v) \in m$(value) do emit$(k, v)$;
    class Reducer
        method reduce ( key, values )
            $B \leftarrow \emptyset$;
            for each $w \in$ values do $B \leftarrow B \cup \{w\}$;
            for each $v \in r$(key,$B$) do emit(key,$v$);
```

where the emit method appends pairs of key-values to the output stream. The actual implementation of MapReduce in MRQL is often stream-based, which does not materialize the intermediate bag B in the reduce code (the cases where streaming is enabled are detected statically by analyzing the reduce function). A variation of the MapReduce operation is Map$(m)\, S$, which is equivalent to MapReduce without the reduce phase. That is, given a map function m of type $\alpha \rightarrow$ bag(β), the operation Map$(m)\, S$ transforms a bag(α) into a bag(β). (It is equivalent to the concat-map or flatten-map in functional programming languages.)

• MapReduce2$(m_x, m_y, r)(X, Y)$, joins the DataSet X of type bag(α) with the DataSet Y of type bag(β) to form a DataSet of type bag(γ). The map function m_x is of type $\alpha \rightarrow$ bag((κ, α')), where κ is the join key type, the map function m_y is of type $\beta \rightarrow$ bag((κ, β')), and the reduce function r is of type (bag(α'), bag(β')) $\rightarrow$ bag(γ). This join can be expressed as follows in MRQL:

```
    select w
    from z in (select $r$(x',y')
                  from x in $X$, y in $Y$,
                       (kx,x') in $m_x$(x),
                       (ky,y') in $m_y$(y)
                  where kx = ky
                  group by k: kx),
         w in z
```

It applies the map functions m_x and m_y to the elements x $\in Y$ and y $\in Y$, respectively, which perform two tasks: transform the elements into x' and y' and extract the join keys, kx and ky. Then, the transformed X and Y elements are joined together based on their join keys. Finally, the group-by lifts the transformed elements x' and y' to bags of values with the same join key kx=ky and passes these bags to r. MapReduce2 captures the well-known equi-join technique used in map-reduce environments, called a reduce-side join [24] or COGROUP in Pig [12]. An implementation of MapReduce2$(m_x, m_y, r)(X, Y)$ in a map-reduce platform, such as Hadoop, is shown by the following Java pseudo-code:

```
    class Mapper1
        method map(key,value)
            for each $(k, v) \in m_x$(value) do emit$(k, (0, v))$;
    class Mapper2
        method map(key,value)
            for each $(k, v) \in m_y$(value) do emit$(k, (1, v))$;
    class Reducer
        method reduce(key,values)
            $xs \leftarrow \{\ v \mid (n, v) \in$ values, $n = 0\ \}$ ;
            $ys \leftarrow \{\ v \mid (n, v) \in$ values, $n = 1\ \}$ ;
            for each $v \in r(xs, ys)$ do emit(key, $v$);
```

That is, we need to specify two mappers, each one operating on a different data set, X or Y. The two mappers apply the join map functions m_x and m_y to the X and Y values, respectively, and tag each resulting value with a unique source id (0 and 1, respectively). Then, the reducer, which receives the values from both X and Y grouped by their join keys, separates the X from the Y values based on their source id, and applies the function r to the two resulting value bags. The actual implementation of MapReduce2 in MRQL is asymmetric, requiring only ys to be cached in memory, while xs is often processed in a stream-like fashion. (Such cases are detected statically, as is done for MapReduce.) Hence the Y should always be the smallest data source or the 1-side of the 1:N relationship.

• Finally, Collect(S), allows us to exit the DataSet algebra by returning a bag(T) value from the DataSet(T), S. That is, it extracts the data from the data set files in S and collects them into a bag. This operation is the only way to incorporate a map-reduce computation inside the functional parameters of another map-reduce computation. It is used for specifying a *fragment-replicate join* (also known as memory-backed join [19]), where the entire data set Y is cached in memory and each map worker performs the join between each value of X and the entire cached bag, Y. This join is effective if Y is small enough to fit in the mapper's memory.

For example, for *Query 1*, MRQL generates a plan that consists of a single MapReduce job:

```
MapReduce(λp.select ( ( i.@category,
                         count(p.watches.@open_auctions) ),
                        p )
                 from i in p. profile . interest ,
            λ((cat,os),ps). { ( cat, os, count(ps) ) } )
  ( source({"person"},xpath (.), "xmark.xml") )
```

where an anonymous function $\lambda x.e$ specifies a unary function (a lambda abstraction) f such that $f(x) = e$, while an anonymous function $\lambda(x, y).e$ specifies a binary function f such that $f(x, y) = e$. Here, for convenience, we have used the MRQL syntax to describe in-memory bag manipulations. In the actual implementation, we use a bag algebra based on the concat-map operator (also known as flatten-map in functional programming languages) to manipulate bags in memory. We have used two kinds of implementations for in-memory bags: stream-based (as Java iterators) and vector-based. The MRQL compiler uses the former only when it statically asserts that the bag is traversed once.

MRQL translates *Query 2* into a workflow of two cascaded MapReduce jobs: The inner MapReduce performs a self-join over the XMark DataSet, while the outer one groups the result by category name. Self-joins do not require a MapReduce2 (an equi-join) operation. Instead, they can be evaluated using a regular MapReduce where the map function repeats each input element twice: once under a key equal to the left join key, and a second time under a key equal to the right join key. MRQL translates *Query 3* into a workflow of three cascaded jobs: a MapReduce, a MapReduce2, and a MapReduce. The first MapReduce groups DBLP items by citation. The MapReduce2 joins the result with the DBLP DataSet to retrieve the title of each citation. The last MapReduce orders the result by the number of items that reference each citation. Both *Query 2* and *Query 3* are evaluated in Section 7.

6. THE MRQL OPTIMIZER

MRQL uses a novel cost-based optimization framework to map the algebraic forms derived from the MRQL queries to efficient workflows of physical plan operators. The most important MRQL algebraic operator is the join operator $\text{join}(k_x, k_y, r)(X, Y)$, which is a restricted version of MapReduce2, because it uses the key functions k_x and k_y to extract the join keys, instead of the general map functions m_x and m_y that transform the values:

$$\begin{aligned}&\text{join}(k_x, k_y, r)(X, Y)\\&= \text{MapReduce2}(\ \lambda x.\{(k_x(x), x)\},\ \lambda y.\{(k_y(y), y)\},\ r\ )\ (\ X,\ Y\ )\end{aligned}$$

MapReduce2 is more general than join because it allows cascaded Map operations to be fused with a MapReduce or a MapReduce2 operation at the physical level, thus requiring one map-reduce job only. Recall that the reduce function r can be any function that gets two bags as input and returns a bag as output, so that, for each join key value, the input bags hold the elements from X and Y that have this key value. Function r can combine these two input bags in such a way that the nesting of elements in the resulting bag would reflect the desirable nesting of the elements in the MRQL query. This is a powerful idea that, in most cases, eliminates the need for grouping the results of the join before they are combined to form the query result.

The MRQL optimizer uses a polynomial heuristic algorithm, which we first introduced in the context of relational queries [10], but adapted to work with nested queries and dependent joins. It is a greedy bottom-up algorithm, similar to Kruskal's minimum spanning tree algorithm. Our approach to optimizing general MRQL queries is capable of handling deeply nested queries, of any form and at any nesting level, and of converting them to near-optimal join plans. It can also optimize dependent joins, which are used when traversing nested collections and XML data. The most important component missing from our framework is a comprehensive cost model based on statistics. In the future, we are planning to use dynamic cost analysis, where the statistics are collected and the optimization is done at run-time. More specifically, we are planning to develop a method to incrementally reduce the query graph at run-time, and enhance the reduce stage of a map-reduce operation to generate enough statistics to decide about the next graph reduction step.

7. CURRENT STATUS OF IMPLEMENTATION AND PERFORMANCE RESULTS

MRQL is implemented in Java on top of Hadoop. The source code is available at `http://lambda.uta.edu/mrql/`. Currently, our system can evaluate any kind of MRQL query, over two kinds of text documents: XML documents and record-oriented text documents that contain basic values separated by user-defined delimiters. We currently provide two different implementations for our physical algorithms, thus supporting two different levels of program testing. The first one is memory-based, using Java vectors to store data sets. This implementation allows programmers to quickly test their programs on small amounts of data. The second implementation uses Hadoop, on either single node or cluster deployment, and is based on the interpretation of physical plans. That is, each physical operator is implemented with a single Hadoop map-reduce program, parameterized by the functional parameters of the physical operator, which are represented as plans (trees) that are evaluated using a plan interpreter. These plans are distributed to the map/reduce tasks through the configuration parameters (which Hadoop sends to each task before each map-reduce job). This implementation method imposes an execution time overhead due to the plan interpretation. We are planning to provide a third implementation option, which will translate the query plans to Java code, will package the code into a Java archive, and will cache this archive to the HDFS, thus making it available to the task trackers.

The platform used for our evaluations was a small cluster of 6 low-end Linux servers, connected through a Gigabit Ethernet switch. Each server has 8 2GHz Xeon cores and 8GB memory. The first set of experiments was over the DBLP dataset [7], which was 809MBs. The MRQL query we evaluated was *Query 3*, which our system compiles into a workflow of two MapReduce and one MapReduce2 jobs. We evaluated the query on 3 different cluster configurations: 2 servers, 4 servers, and 6 servers. Since the number of map tasks is proportional by the number of splits, which we do not have any control, we performed our evaluations by vary-

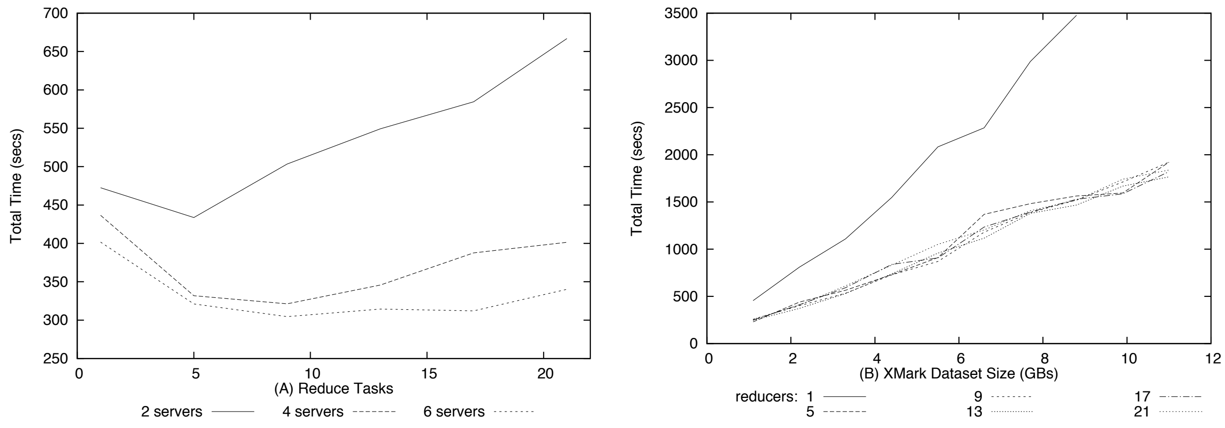

Figure 1: Evaluation time for MRQL queries over (A) DBLP data and (B) XMark data

ing only the number of reduce tasks per job: 1, 5, 9, 13, 17, and 21. Furthermore, we did not force the results of the query (which were produced by multiple reducers) to be merged into one HDFS file, but, instead, we left them on multiple files. Figure 1.A shows the results. As expected, we got the best performance when we used all 6 servers, but the difference from the 4-server configuration was not substantial. In addition, we got the fastest response when the number of reducers was set to 9. Apparently, an increase in the number of reductions causes an increase in the number of splits, which need to be processed by more mappers in the subsequent job. The second dataset used for our evaluations was the synthetic dataset XMark [26]. We generated 10 data sets in single files, ranging from 1.1GBs up to 11GBs. We evaluated *Query 2*, which MRQL translates into a workflow of two MapReduce jobs. Here we used all 6 servers and tried six values for the numbers of map and reduce tasks per job: 1, 5, 9, 13, 17, and 21. Figure 1.B shows the results. As expected, the evaluation time is proportional to the data size. What was unexpected was that the performance worsen only when number of reducers was set to 1, while all other settings for the number of reducers produced similar results.

8. CONCLUSION

We have presented a powerful query language, MRQL, for map-reduce computations over XML data that has the familiar SQL syntax and is expressive enough to capture many common data analysis tasks. We have also presented a small set of physical plan operators that are directly implementable on existing map-reduce systems and form a suitable basis for MRQL query evaluation. As a future work, we are planning to extend our framework with more optimizations, more evaluation algorithms, and more data formats, including relational databases and key-value indexes.

9. REFERENCES

[1] A. Abouzeid, *et al*. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. In *VLDB'09*.

[2] D. Battre, *et al*. Nephele/PACTs: A Programming Model and Execution Framework for Web-Scale Analytical Processing. In *SOCC'10*.

[3] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: Efficient Iterative Data Processing on Large Clusters. In *VLDB'10*.

[4] M. J. Cafarella and C. Ré. Manimal: Relational Optimization for Data-Intensive Programs. In *WebDB'10*.

[5] R. Cattell. The Object Data Standard: ODMG 3.0. Morgan Kaufmann, 2000.

[6] R. Chaiken, *et al*. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. In *PVLDB'08*.

[7] DBLP XML records, the DBLP Computer Science Bibliography. Available at `http://dblp.uni-trier.de/xml/`.

[8] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04*.

[9] J. Dittrich, *et al*. Hadoop++: Making a Yellow Elephant Run Like a Cheetah (Without It Even Noticing). In *VLDB'10*.

[10] L. Fegaras. A New Heuristic for Optimizing Large Queries. In *DEXA'98*.

[11] L. Fegaras. The Joy of SAX. In *XIME-P'04*.

[12] A. F. Gates, *et al*. Building a High-Level Dataflow System on top of Map-Reduce: the Pig Experience. In *PVLDB* 2(2), 2009.

[13] Hadoop. `http://hadoop.apache.org/`.

[14] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys'07*.

[15] M. Isard and Y. Yu. Distributed Data-Parallel Computing Using a High-Level Programming Language. In *SIGMOD'09*.

[16] E. Jahani, M. J. Cafarella, and C. Ré. Automatic Optimization for MapReduce Programs. In *PVLDB'11*, 4(6).

[17] Jaql: Query Language for JavaScript Object Notation (JSON). At `http://code.google.com/p/jaql/`.

[18] S. P. Jones and P. Wadler. Comprehensive Comprehensions (Comprehensions with 'Order by' and 'Group by'). In *Haskell'07*.

[19] J. Lin and C. Dyer. Data-Intensive Text Processing with MapReduce. Book pre-production manuscript, April 2010.

[20] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: a not-so-Foreign Language for Data Processing. In *SIGMOD'08*.

[21] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the Data: Parallel Analysis with Sawzall. *Scientific Programming* 13(4), 2005.

[22] A. Thusoo, *et al*. Hive: a Warehousing Solution over a Map-Reduce Framework. In *PVLDB* 2(2), 2009.

[23] A. Thusoo, *et al*. Hive: A Petabyte Scale Data Warehouse Using Hadoop. In *ICDE'10*.

[24] T. White. Hadoop: The Definitive Guide. O'Reilly, 2009.

[25] Xadoop. At `http://www.xadoop.org/`.

[26] XMark – An XML Benchmark Project. At `http://www.xml-benchmark.org/`.