Having a ChuQL at XML on the Cloud

Shahan Khatchadourian[1], Mariano P. Consens[1], and Jérôme Siméon[2]

[1] University of Toronto
shahan@cs.toronto.edu, consens@cs.toronto.edu
[2] IBM Watson Research
simeon@us.ibm.com

Abstract. MapReduce/Hadoop has gained acceptance as a framework to process, transform, integrate, and analyze massive amounts of Web data on the Cloud. The MapReduce model (simple, fault tolerant, data parallelism on elastic clouds of commodity servers) is also attractive for processing enterprise and scientific data. Despite XML ubiquity, there is yet little support for XML processing on top of MapReduce.
In this paper, we describe ChuQL, a MapReduce extension to XQuery, with its corresponding Hadoop implementation. The ChuQL language incorporates records to support the key/value data model of MapReduce, leverages higher-order functions to provide clean semantics, and exploits side-effects to fully expose to XQuery developers the Hadoop framework. The ChuQL implementation distributes computation to multiple XQuery engines, providing developers with an expressive language to describe tasks over big data.

1 Introduction

The emergence of Cloud computing has led to growing interest in new programming paradigms for data-intensive applications. In particular, Google's MapReduce [7] has gained traction as an economically attractive, scalable approach to process, transform, integrate, and analyze massive amounts of Web data, as well as enterprise and scientific data. The MapReduce model provides simple, fault-tolerant, data parallelism on elastic clouds of commodity servers. Hadoop [12] is a popular open source framework implementing MapReduce on top of a reliable distributed file system. Because of Hadoop's simplicity and scalability, it is already being used beyond Web companies such as Yahoo! and Facebook[3].

Despite XML dominance as a standard format for many industries, such as publishing [9], government [13], finance [11], and life sciences [8], there is yet little support for XML processing on top of MapReduce frameworks, Hadoop in particular. Current approaches to process XML in Hadoop rely either on writing arbitrary Java code (e.g., using Hadoop's built-in `StreamXmlRecordReader` class to convert XML fragments into records), or on invoking (via a streaming utility) an executable or a script interpreted by a suitable language. While numerous

[3] E.g., see a list of known users at `http://wiki.apache.org/hadoop/PoweredBy` and `http://www.cloudera.com/customers/`.

languages and systems [1, 4–6, 15–18, 24] have been proposed to facilitate development in Cloud environments beyond the simple MapReduce/Hadoop model, there is little support for XML as a native format.

In this paper we propose ChuQL, a MapReduce extension to XQuery with a corresponding Hadoop implementation. ChuQL's goals are to provide developers with a familiar XQuery-based tool to express XML oriented data processing tasks on the cloud, while giving them transparent access to the Hadoop framework and its multiple customization points. XQuery developers using ChuQL can readily harness the benefits of Hadoop's scalability and fault-tolerance on commodity servers. ChuQL fully leverages XQuery's expressiveness (both its declarative aspects and its imperative extensions, including the ability to invoke external computations) to facilitate the development of complex processing tasks over massive amounts of Cloud data.

The ChuQL implementation takes care of distributing the computation to multiple XQuery engines running in Hadoop nodes, as described by one or more ChuQL MapReduce expressions. The ChuQL approach provides developers with the benefit of a powerful language implemented by mature XQuery processors to perform XML data processing tasks at each node in a Hadoop cluster. The ChuQL language incorporates records to support the key/value data model of MapReduce, leverages higher-order functions to provide clean semantics, and exploits side-effects to fully expose to XQuery developers the Hadoop framework.

Related Work. Numerous high-level languages for Hadoop have been proposed, with Yahoo!'s Pig/Latin [17], IBM's Jaql [15], and SQL-based Hive [22] and HadoopDB [1] included among the most prominent. Because these languages rely on more traditional data models, either relational or nested-relational (often in the form of JSON), such approaches offer limited native support for tree models such as XML. The Jaql query language can be used to execute complex analytical jobs for JSON-like data on top of Hadoop. The XML data model is captured only lossily based on the conversion of XML data to Jaql's JSON type (using a provided *XmlToJson* function). Pig Latin is a query language with a nested data model that also uses Hadoop. XML can only be represented as a string value of a field. Unlike Jaql, it does not include a function that converts XML content to a format that can be handled by its query processor and so requires third-party plug-ins to support XML. While Hive uses tables stored as files on HDFS, HadoopDB extends Hive to leverage the MapReduce processing model with single-node DBMS instances.

Several high-level languages have been proposed for non-Hadoop environments. While Hyracks [4] and Nephele [3] extend the parallel processing model beyond MapReduce, BOOM [2] shows how a declarative language can be used to implement parallel processing models in orders of magnitude fewer lines of code. DryadLINQ [24] and Scope [5] are two high-level declarative languages that are compiled for processing tables on a Dryad [14] cluster. DryadLINQ [24] is of particular interest since it does provide support for XML through an elegant 'model agnostic' support for iterators at the programming language level, which allows various models to be integrated, including XML.

Organization and Contributions. The main contributions of the paper are:

- In Section 2, we illustrate how to use ChuQL to write fairly complex data processing jobs on top of XML data, including common Cloud data processing operations, such as co-grouping.
- The ChuQL language is defined precisely, in Section 3, as an extension of XQuery with MapReduce capabilities. This relies on a small extension of the XML data model with records, which is used to model the notion of key/value pairs that is central to processing with MapReduce.
- ChuQL has been implemented using Hadoop and the IBM WebSphere XML feature pack [10]. We give a brief overview of that implementation in Section 4, and mention early experimental results.
- Section 5 concludes the paper and suggests some future work.

2 ChuQLing

In this section, we give a brief introduction to MapReduce/Hadoop, and introduce ChuQL through examples.

2.1 MapReduce and Hadoop

MapReduce is a parallel processing framework for analyzing large datasets residing on distributed storage. In our work, we focus on Hadoop, a popular open source implementation of MapReduce written in Java. Hadoop's default storage system is the fault-tolerant Hadoop Distributed File System (HDFS), which maintains copies of the data on a cluster of commodity machines. One of the benefits of using a file system rather than a traditional database index is that it can serve any kind of data, such as website access logs or XML files.

Hadoop implements the MapReduce processing model, and can process large datasets stored on HDFS. The processing is performed through a fixed number of phases, including the *map* phase and the *reduce* phase, which are then automatically decomposed into parallel tasks. Each phase is described as a function over key/value pairs that may contain any data. Key/value pairs which are output of the *map* are serialized to HDFS for fault-tolerance and then grouped together according to their key before they are passed to the *reduce* phase. As a result, the reduce phase processes groups of key/value pairs, giving the user the ability to efficiently evaluate aggregate operations over the data. In addition, Hadoop provides two additional phases to read and write files from/to HDFS. The *record reader* (RR, for short) phase is used to read the input from disk and create the initial key/value pairs for the *map*. The *record writer* (RW, for short) phase is used to serialize on disk the key/value pairs produced from the *reduce*.

One of the key benefits of using Hadoop is that it supports deployment on a cluster of machines. Hadoop takes care of aspects related to parallelization of the work, including splitting the work into independent tasks, orchestration of those tasks, and provides mechanisms for fault tolerance in the case of machine failure

or if tasks do not complete. Each pair of associated RR and map tasks runs in a distinct Java Virtual Machine (JVM), and each pair of associated reduce and RW tasks runs in a distinct JVM. Tasks are monitored and restarted if they fail. A map task that fails triggers the re-execution of that task (possibly on a different machine) with the initial input of the failed task, and a reduce task that fails triggers a re-execution of that task.

2.2 WordCount in ChuQL

WordCount is a simple program that leverages MapReduce to compute the cardinality of words in a text collection stored on HDFS. Since it typically serves as the *"Hello World!"* of MapReduce, we also use it here as our first ChuQL program. As shown on Figure 1, it uses the **mapreduce** expression that is the main extension of ChuQL.

```
1   mapreduce {
2     input { fn:collection("hdfs://input/") }
3     rr { for $line at $i in $in//line return { key: $i, val: $line } }
4     map { for $word in fn:tokenize($in=>val, "␣")
5           return { key: $word, val: 1 } }
6     reduce { { key: $in=>key, val: fn:count($in=>val) } }
7     rw { <word text="{$in=>key}" count="{$in=>val}"/> }
8     output { fn:put($in,"hdfs://output/") }
9   }
```

Fig. 1. WordCount in ChuQL

In ChuQL, the **mapreduce** expression can be used to describe a MapReduce job, and is specified using the following clauses: (i) **input** and **output** clauses to respectively read and write onto HDFS, (ii) **rr** and **rw** clauses describing respectively the record reader and writer, (iii) **map** and **reduce** clauses describing respectively the *map* and *reduce* phases. Those clauses process XML values or key/value pairs of XML values to match the MapReduce model and are specified using XQuery expressions. In each of those expression, the special variable ***$in*** is bound to the input currently being processed. An expression can return key/-value pairs using the record constructor `{ key:` *Expr*`, val:` *Expr* `}`. The key or value can then be accessed using the record accessor expression *Expr*`=>key` or *Expr*`=>val`.

The ChuQL program on Figure 1 uses 9 lines of code as opposed to the original 150-line Java version. In the original Java implementation of WordCount over Hadoop, the RR task produces key/value pairs whose keys are the byte offsets of lines, and whose values are the corresponding lines of text. In our case, we assume the input is an XML document that includes a sequence of `<line>...</line>` elements containing text. The **rr** clause on line 3 accesses each line with its index (in variable ***$i***), and returns key/value pairs whose index is the key, and whose value is the line element.

Similar to the original Java implementation, the **map** clause on line 5 tokenizes the lines of text and produces key/value pairs whose keys are the tokenized

words, and whose values are all set to 1. Then, the keys with the same word are grouped together, and the **reduce** clause on line 6 counts each word by counting the number of 1's associated with each word. Finally, the **rw** clause on line 7 creates an XML element *word*, with an attribute *text* containing the word, and an attribute *count* containing the word count.

The *fn:collection* and *fn:put* functions are used in the **input** and **output** clauses on lines 2 and 8 respectively to specify the job's input and output. To that effect, we extend the XQuery URL scheme with support for *hdfs*. The function *fn:put* is part of the XQuery Update Facility [19]. It "stores" the values at the provided URL through a side-effect and returns the empty sequence. A consequence of using the *fn:put* function is that the value of the overall expression is also the empty sequence; however, another expression could then use the *fn:collection* function again to access that job's output. We show later in this section how to return a job's output directly into memory so that it can be further processed using XQuery.

In terms of implementation, the ChuQL engine, which itself runs in a JVM, processes a ChuQL expression by configuring and starting jobs, handling job output, and processing the remainder of the expression. A job configuration is built from the task expressions extracted from the abstract syntax tree of the ChuQL expression. A job is then started with that configuration, and finally, its output is either stored on HDFS or in memory. Within the job, each task JVM has an XQuery processor to compile and process task expressions.

2.3 Analytics in ChuQL

Another classic use of MapReduce is as a platform for analytics, notably leveraging the ability to aggregate information. In our second ChuQL example, we illustrate its use for aggregation. We consider a simple scenario where one data source exports a large amount ot sales data in XML, and one wants to correlate that information with a public Web collection, such as Wikipedia, to cross-reference city sales with e.g., local population or city statistics. If the size of the two collections is substantial, attempting to perfom the desired computation using a single XQuery processor may consume hundreds of hours; instead, using a Hadoop cluster to perform the computation can reduce the computation time significantly. Also, because one of the collections is not purely in XML (wiki markup), there is a need to support invocation of special-purpose code to process part of the data.

The example on Figure 2 shows a possible implementation using ChuQL that combines the output of two distinct MapReduce jobs. The first job (lines 4 to 12) extracts city statistics from Wikipedia articles using the external function *extractcitystats* that interfaces to a Perl script that extracts the relevant city information from the wiki markup. First, the RR task on line 7 creates a value with the article wikitext which is then transformed in the map task on line 9 into key/value pairs containing city statistics using *extractcitystats*. The key/value pairs are then grouped by their key, the city in this case. The reduce

```
1  declare function my:extractcitystats($article) as element(my:stats) external;
2
3  let $stats :=
4    mapreduce {
5      input { fn:collection("hdfs://wikipedia/") }
6      rr { for $article in $in//article
7           return { key: <empty/>, val: $article } }
8      map { for $stat in my:extractcitystats($in=>val)
9            return { key: $stat/city, val: $stat } }
10     reduce { { key: $in=>key, val: <stats city="{$in=>key}">{$in=>val}</stats> } }
11     rw { $in=>val }
12     output { $in }
13   }
14 let $sales :=
15   mapreduce {
16     input { fn:collection("hdfs://sales/") }
17     rr { for $sale in $in//sale[@period="1Q2011"]
18          return { key: <empty/>, val: $sale } }
19     map { { key: $in=>val/city, val: $in=>val/sale } }
20     reduce { { key: $in=>key,
21                val: <sales city="{$in=>key}">{avg($in=>val/amount)}</sales> } }
22     rw { $in=>val }
23     output { $in }
24   }
25 for $city in $sales/@city
26 return
27    <salesreport city="{$city}">
28      <sales>{$sales[@city=$city]}</sales>
29      <stats>{$stats[@city=$city]}</stats>
30    </salesreport>
```

Fig. 2. Analytics across collections

task on line 10 aggregates each city's statistics from all the articles, which is then returned in memory and assigned to the *stats* variable on line 12. Similarly, the second job (lines 15 to 23) computes city sales. First, on line 18, a key/value pair is returned for each sale in the given period. Each is then transformed into a key/value pair by the map task on line 19 by placing the sale's city as the key and leaving the sale as the value. Like the first job, key/value pairs are grouped by city. The reduce task on line 21 performs the sales analysis for each city, which is the average of all the sales amounts, and then returns the result in memory and assigns it to the *sales* variable on line 22. Finally, lines 25 to 30 are used to combine the job outputs, including sales average, and statistics, for each city.

This example shows that the compositionality of XQuery makes it easy to express whether to store output on HDFS or to return it in memory. In WordCount, above, we showed that job output can be stored on HDFS using the side-effect function *fn:put* (and which then returns the emtpy sequence), but can alternatively be returned to the ChuQL expression by returning XML values from the output expression (in this case, the variable ***$in***).

2.4 Co-grouping

Finally, we show an example of co-grouping [17], a common operation in MapReduce scenarios. Co-grouping is a way to return *groups* of elements from several sets in a single result, differing from grouping in relational databases where sets

cannot typically be nested within a result row. Co-grouping can be performed efficiently using MapReduce due to the parallelism involved in examining and filtering collections. Most languages proposed for programming over MapReduce provide special purpose support for co-grouping [15,17]. In this example, we show how ChuQL can support co-grouping by leveraging imperative features proposed in the XQuery Scripting Extension [21].

```
1 mapreduce {
2   input {fn:collection("hdfs://wikipedia/")}
3   rr { for $article in $in//article
4        return { key: <empty/>, val: $article } }
5   map { for $revision in $in=>val/revision
6         return { key: fn:data($revision/date),
7                  val: ($in=>val/title | $in=>val/revision/author) } }
8   reduce { block {
9              declare variable $ts := <top/>;
10             declare variable $as := <top/>;
11             (for $t in $in=>val return
12               if ($t/title) then (insert node $t as last into $ts)
13               else if ($t/author) then (insert node $t as last into $as)
14               else ()),
15             { key: $in=>key, val: <mods date="{$in=>key}">
16                                     <titles>{$ts/title}</titles>
17                                     <auth>{$as/author}</auth></mods> } } }
18  rw { $in=>val }
19  output {fn:put($in,"hdfs://cogroupoutput/") }
20 }
```

Fig. 3. Co-grouping using XQuery Scripting

A ChuQL expression that co-groups authors and article titles from Wikipedia by revision date is shown in Figure 3. The Hadoop job is processed as follows. First, the RR expression on line 3 places each article element as the value of a key/value pair, each of which is then processed by the map task on line 5 to produce key/value pairs whose key is the article's revision date and whose value is an XML sequence containing the article title and revision author. After grouping the key/value pairs by revision date, the reduce task is specified as an XQuery Scripting expression that iterates over the *$in=¿val* sequence once (line 11). The variables *as* and *at* (initialized on line 9) are used to store the titles and authors on lines 12 and 13, respectively. Titles and authors grouped by each distinct revision date are returned as the value of key/value pairs on line 15. Finally, the record writer expression on line 18 selects the values of key/value pairs, and serializes them to HDFS on line 19. In contrast, if an XQuery Scripting expression were not used in the reduce task, the *$in=¿val* sequence would need to be iterated over twice - once to extract authors, and a second time to extract titles.

3 The ChuQL Language

In this section, we describe the syntax and semantics of ChuQL, our MapReduce extension to XQuery 3.0 [20]. We first extend the XQuery data model (XDM) [23] and type system with records.

Data Model. Values in our data model include XML values and records. They are described by following grammar, in which a1...an are record fields.

```
1  Value ::= ... XQuery 3.0 values ...
2          | { a1: Value, ... , an: Value }
```

In this paper, records are simply used to provide support for key/value pairs, which play a central role in the MapReduce processing model. For instance, the following is a pair whose key is the integer 1, and whose value is a text node containing the string "Hello World".

```
1  { key : 1, value: text { "Hello World!" } }
```

Type System. Similarly, our type system include XML types and record types. They are described by the following grammar, in which a1...an are record fields.

```
1  Type ::= ... XQuery 3.0 types ...
2         | { a1: Type, ... , an: Type }
```

In this paper, types are used to ensure that expressions used from within **mapreduce** have the proper input and output, i.e., key/value pairs containing XML data. For instance, the following type describes pairs whose key is an integer, and whose value is a text node.

```
1  { key : xs:integer, value: text }
```

For readability, we introduce the following two built-in types to ChuQL. The first type stands for any XML value (item()* in the XQuery sequence type syntax). The second type stands for any key/value pair containing XML values.

```
1  define type chuql:xml { item()* };
2  define type chuql:keyval { { key: item()*, value: item()* } };
```

Relationship to XQuery. ChuQL is built on top of XQuery [20], the XML Query language designed by the W3C. We briefly review some of the main features of XQuery which are relevant to ChuQL. We make use of recent developments in the standard, notably XQuery 3.0 and XQuery Scripting Extensions.

- First, XQuery is compositional, i.e., it includes a set of expressions which can be composed together to build larger more complex expressions. This facilitates language extension, including ChuQL, as powerful features can be built by simply extending the original expression syntax, along with providing the proper typing rules and semantics.
- Second, we define the semantics of the **mapreduce** expression by leveraging XQuery 3.0 higher order functions, and the group by feature.

– Finally, we make use of XQuery Scripting in a number of examples, notably in order to describe MapReduce job whose output is being stored back onto HDFS.

Grammar. We extend XQuery's expression syntax with three new expressions, as described by the following grammar.

```
Expr  ::= ... XQuery 3.0 expressions ...
        | { a1: Expr, ... , an: Expr }     (: record creation :)
        | Expr=>ai                          (: record field access :)
        | mapreduce { input { Expr }       (: map/reduce expression :)
                    rr { Expr }
                    map { Expr }
                    reduce { Expr }
                    rw { Expr }
                    output { Expr } }
```

The first expression constructs a key/value pair (i.e., a record with two fields key and value). The second expression accesses either the key or the value field of a key/value pair. The last expression is the map/reduce expression, as was already illustrated in examples from Section 2.

Semantics. We are now ready to describe the semantics of ChuQL, focusing on the **mapreduce** expression. It is interesting to note that the semantics can be fully expressed in terms of XQuery 3.0 constructs.

We first define a built-in function that encodes the semantics of the mapreduce expression in terms of its sub-expressions. This function relies heavily on the recently proposed higher-order functions for XQuery 3.0. Higher-order functions are used to capture the fact that sub-expressions in the various mapreduce clauses are always expressed in terms of an input value, represented by the variable *$in*.

```
declare function chuql:mapreduce(
        $input as function() as chuql:xml,
        $rr as function($in as chuql:xml) as chuql:keyval*,
        $map as function($in as chuql:keyval) as chuql:keyval*,
        $reduce as function($in as chuql:keyval) as chuql:keyval*,
        $rw as function($in as chuql:keyval) as chuql:xml,
        $output as function($in as chuql:xml) as chuql:xml
    ) as chuql:xml
{
   let $input-result  := $input()
   let $rr-result     := for $in in $input-result return $rr($in)
   let $map-result    := for $in in $rr-result return $map($in)
   let $reduce-result :=
     for $in in $map-result
     let $key := $in=>key
     group by $key
     return $reduce({ key: $key, val: for $x in $in return $x=>val })
   let $rw-result := for $in in $reduce-result return $rw($in)
   let $output-result := for $in in $rw-result return $output($in)
   return $output-result
};
```

Note that the function merely evaluates each phase in order, passing the result of each phase to the next. The only phase that requires special attention is, not surprisingly, the reduce phase, as it must group its input based on each key. It simply uses the XQuery 3.0 group by to capture the grouping semantic

of the map/reduce processing model. Note that, in accordance to the XQuery 3.0 semantics, after the group by is applied, the *$in* variable does not contain each key/value pair, but all the key/value pairs with the same key. Also note that in XQuery 3.0, the grouping criteria is always applied using deep-equality, which gives a precise, but not always flexible, semantics for the ChuQL **mapreduce** expression. We are currently considering extensions to the grouping semantics which would allow a user to explicitly specify the kind of equality they desire, and requires an extension to the XQuery 3.0 group by expression.

The **mapreduce** expression can be defined by simply mapping it into the `chuql:mapreduce` function we just defined.

```
1  ⟦ mapreduce {
2      input { InputExpr }
3      rr { RRExpr }
4      map { MapExpr }
5      reduce { ReduceExpr }
6      rw { RWExpr }
7      output { OutputExpr }
8    } ⟧
9  ≡
10 chuql:mapreduce(function() as chuql:xml { InputExpr },
11                 function($in as chuql:xml) as chuql:keyval* { RRExpr },
12                 function($in as chuql:keyval) as chuql:keyval* { MapExpr },
13                 function($in as chuql:keyval) as chuql:keyval* { ReduceExpr },
14                 function($in as chuql:keyval) as chuql:xml { RWExpr },
15                 function($in as chuql:xml) as chuql:xml { OutputExpr })
```

Finally, it may be worth mentioning that this approach provides the complete semantics for ChuQL, but does not make any aspects of parallel execution explicit.

4 ChuQL on the IBM Cloud

Due to space limitations, we give here only a brief overview of the implementation and of preliminary experiments run using a ChuQL deployment on the IBM Cloud.

Implementation Our ChuQL processor is built on top of Hadoop 0.21.0 and the so-called "thin client" of the WebSphere XML Feature Pack [10], which provides a complete XQuery implementation in Java. We modified the XQuery processor to accept the new `mapreduce` construct by modifying the abstract syntax tree (to configure jobs), updated the internal intermediate language (to invoke jobs), as well as adding the new record type described in the previous section. A ChuQL job is based on a regular Hadoop job with custom RR, map, reduce, and RW task classes written in Java. Within each customized task class, the class' JVM invokes the XQuery processor to compile and process a task expression (taken from the corresponding ChuQL expression).

Experiments Up to 20 virtual machines on the IBM Cloud were used for the experiments. Each instance, configured to use 0.9 of a physical machine, had a 32-bit quad-core CPU, 3.64 GB of RAM, and a 350 GB hard disk. The IBM 1.6 JDK was used as the JVM.

Some of Wikimedia's XML datasets were used as for the experiments. Each dataset is provided as a single large XML file containing the latest revision of each article, including the article's metadata, and its content as wikitext. We partitioned and stored each dataset on HDFS as XML files containing 1,000 articles each; this value, obtained by hand tuning, produced XML files that could be processed by the XQuery Engine within memory limits.

We report initial performance and scalability results for a co-grouping job that uses XQuery Scripting. The job on the largest dataset completed in over 15 hours with 6 machines and approximately 7 hours with 20 machines. Our initial performance results show that our approach is realistic and that ChuQL can be used for processing large XML collections in parallel using Hadoop. Additional experiments will examine larger datasets (such as the multi-terabyte Wikipedia collection containing all article revisions), and they will also compare different ChuQL expressions (e.g., to quantify the performance benefits of using XQuery Scripting).

5 Conclusion

In this work we have described ChuQL, a language that allows XML to be processed in a distributed manner using MapReduce. ChuQL extends the syntax, grammar, and semantics of XQuery, and also leverages the compositionality, side-effects, and higher-order functions of XQuery 3.0. Our initial experimental results show that ChuQL facilitates XML processing of large datasets on Hadoop Clouds.

We conclude mentioning two promising directions for future work. The first is expanding ChuQL capabilities to take advantage of richer Cloud platforms (such as advanced distributed stores and parallel processing models beyond MapReduce). The second is to provide the ability to compile XQuery expressions into ChuQL (i.e., generating a data parallel plan for non-ChuQL expressions, with the added convenience that the plan created can be reviewed and then executed as a ChuQL expression).

Acknowledgements The first author was supported by an IBM CAS Fellowship. We also thank the reviewers for their detailed comments.

References

1. A. Abouzeid, K. Bajda-Pawlikowski, D. J. Abadi, A. Rasin, and A. Silberschatz. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. *PVLDB*, 2(1):922–933, 2009.
2. P. Alvaro, T. Condie, N. Conway, K. Elmeleegy, J. M. Hellerstein, and R. Sears. Boom analytics: exploring data-centric, declarative programming for the cloud. In *EuroSys*, pages 223–236, 2010.
3. D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke. Nephele/-PACTs: a programming model and execution framework for web-scale analytical processing. In *SoCC*, pages 119–130, 2010.

4. V. Borkar, M. J. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *ICDE*, 2011.
5. R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: easy and efficient parallel processing of massive data sets. *PVLDB*, 1(2):1265–1276, 2008.
6. S. Das, Y. Sismanis, K. Beyer, R. Gemulla, P. Haas, and J. McPherson. Ricardo: Integrating R and Hadoop. In *SIGMOD*, 2010.
7. J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.
8. R. Dolin, L. Alschuler, S. Boyer, C. Beebe, F. Behlen, P. Biron, and A. Shabo. HL7 Clinical Document Architecture, Release 2. *Journal of the American Medical Informatics Association*, 13(1):30–39, 2006.
9. ePub. Open Publication Structure (OPS) 2.0, July 2007. Recommended Specification.
10. WebSphere application server feature pack for XML. `http://www-01.ibm.com/software/webservers/appserv/was/featurepacks/xml/`.
11. Introducing FpML: A New Standard for ecommerce. http://www.fpml.org, 1999.
12. Apache Hadoop. `http://hadoop.apache.org`, Jan. 2011.
13. Modernized e-File (MeF) Guide for Software Developers And Transmitters. Internal Revenue Service, Electronic Tax Administration. Publication 4164.
14. M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev.*, 41:59–72, March 2007.
15. Jaql: A JSON Query Language. `http://code.google.com/p/jaql`, Jan. 2011.
16. C. Olston, S. Chopra, and U. Srivastava. Generating example data for dataflow programs. In *SIGMOD*, pages 245–256, 2009.
17. C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: a not-so-foreign language for data processing. In *SIGMOD Conference*, pages 1099–1110, 2008.
18. R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the Data: Parallel Analysis with Sawzall. *Scientific Programming*, 13(4):277–298, 2005.
19. J. Robie, D. Chamberlin, M. Dyck, D. Florescu, J. Melton, and J. Siméon. XQuery update facility. `http://www.w3.org/TR/xquery-update-10/`, Jan. 2011.
20. J. Robie, D. Chamberlin, M. Dyck, and J. Snelson. XQuery 3.0: An XML Query Language. `http://www.w3.org/TR/xquery-30/`, Dec. 2010.
21. J. Snelson, D. Chamberlin, D. Engovatov, D. Florescu, G. Ghelli, J. Melton, and J. Siméon. XQuery Scripting Extension 1.0. `http://www.w3.org/TR/2010/WD-xquery-sx-10-20100408`.
22. A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy. Hive – a petabyte scale data warehouse using hadoop. In *ICDE*, pages 996–1005, 2010.
23. N. Walsh and J. Snelson. XQuery and XPath Data Model (XDM) 3.0. `http://www.w3.org/TR/xpath-datamodel-30/`, Dec. 2010.
24. Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, pages 1–14, 2008.