Versatile XQuery Processing in MapReduce

Caetano Sauer, Sebastian Bächle, and Theo Härder

University of Kaiserslautern
P.O. Box 3049, 67653 Kaiserslautern, Germany
{csauer,baechle,haerder}@cs.uni-kl.de

Abstract. The MapReduce (MR) framework has become a standard tool for performing large batch computations—usually of aggregative nature—in parallel over a cluster of commodity machines. A significant share of typical MR jobs involves standard database-style queries, where it becomes cumbersome to specify *map* and *reduce* functions from scratch. To overcome this burden, higher-level languages such as HiveQL, PigLatin, and JAQL have been proposed to allow the automatic generation of MR jobs from declarative queries. We identify two major problems of these existing solutions: (i) they introduce new query languages and implement systems from scratch for the sole purpose of expressing MR jobs; and (ii) despite solving some of the major limitations of SQL, they still lack the flexibility required by *big data* applications. We propose *BrackitMR*, an approach based on the XQuery language with extended JSON support. XQuery not only is an established query language, but also has a more expressive data model and more powerful language constructs, enabling a much greater degree of flexibility. From a system design perspective, we extend an existing single-node query processor, *Brackit*, adding MR as a distributed coordination layer. Such heavy reuse of the standard query processor not only provides performance, but also allows for a more elegant design which transparently integrates MR processing into a generic query engine.

1 Introduction

MapReduce [4] arose as a key technology in the context of *big data*, a term whose definition depends not only on the subjective interpretation of "big", but also on the problem context. Most big-data technologies consist of systems which are designed from scratch to fulfill technical requirements imposed by the large size of the data, which in a particular point in time are not fulfilled by prevalent technologies. In the case of MR, the related prevalent technology to which it is usually compared consists of parallel database systems for analytics and data warehousing. In contrast to such systems, MR hits an "abstraction sweet spot", because it only abstracts the distribution and fault tolerance aspects of the execution, leaving data formats and operations performed on individual items *under complete responsibility* of the developer. Furthermore, MR is concerned solely with the execution of tasks and not the management of stored data—a key difference to database systems, which exclusively control all read and

B. Catania, G. Guerrini, and J. Pokorný (Eds.): ADBIS 2013, LNCS 8133, pp. 204–217, 2013.

write operations on the data. Because it is only focused on task execution, a large portion of the query processing logic present in a DBMS must in fact be implemented from scratch by the programmer.

A significant share of typical MR jobs consists of a composition of traditional database query operations such as filtering, grouping, sorting, aggregating, and joining. Therefore, it makes sense to push the abstraction further and introduce query languages that allow expressing such tasks in a declarative manner. A great advantage of this approach is that it allows the reuse of *existing data processing operators* as well as *query rewrite* and *optimization logic*.

It is clear that database systems and SQL can be used for that end, but the requirements of *big data* applications make them prohibitive for two main reasons: First, large-scale data warehouse systems are complex, expensive, and hard to customize towards specific end-user needs, especially for applications outside the traditional business domain. Second, and perhaps most critical, they make use of SQL, which enforces a flat, normalized representation of data. This restriction is significant for application domains like scientific computing and the Web 2.0. What is even worse for such scenarios, which often involve *ad-hoc* analysis procedures, is that the exact schema must be specified upfront, which contributes further to the inflexibility and high setup costs of such systems.

1.1 Related Work

One of the earliest proposals for a query language for MapReduce was from Hive [11], a data warehouse system with an SQL-like interface. Its evaluation engine represents the basic mechanism for converting a computational model based on query operators into that of MR. However, being designed to resemble traditional relational systems, it suffers from all the major limitations of SQL and the relational model. Its main focus is to provide a relational data warehouse as similar as possible to existing systems, but relying on MR for the execution layer. Pig [7] introduces a dataflow-oriented language, which aims to be easier to use than SQL for users accustomed with imperative programming. It allows computations to be described using variables which hold references to collections of data items. Operations are then applied to these variables and stored on a new variable. By collecting data dependencies of such variables, a directed acyclic graph (DAG) of operators is built, which serves essentially the same purpose as a query plan. JAQL [3] came up as a more advanced approach, relying heavily on functional-programming features such as higher-order functions. JAQL offers a very elegant solution from the language perspective, it provides a flexible JSON-based data model with support for partial schema specifications. The performance of the three related systems was measured in [10].

The work in [9] provides more detailed qualitative analysis of the four query languages, presenting basic language requirements for flexible query processing in the big data domain. As the authors observe, HiveQL and PigLatin do not fully overcome the deficiencies of SQL, and although no query language leads in all criteria, JAQL and XQuery clearly provide a higher degree of flexibility.

1.2 Contribution

We present BrackitMR, an XQuery processor extended with MR capabilities. XQuery is a wide-spread, generic data-processing language which can be used in a vast range of applications, from traditional DBMS processing to document management, data integration, and message processing. It supports the processing of untyped data, allowing schema information to be added at later stages, as well as partial schema definitions, which enable gradual schema refinement. Furthermore, it provides a hierarchical data model with support for path-based queries, allowing efficient storage and processing of denormalized data. Our prototype extends the XQuery data model with JSON, thereby providing concise representation and efficient processing not only for XML, but also for relational datasets. The main advantage of our approach is that it provides a more flexible data model while still reaching the same performance as the state-of-the art approaches for strictly relational data.

BrackitMR is also a flexible tool for large-scale query processing, because it builds upon an existing single-core query processor, Brackit, adding MR as a distributed coordination layer. This means that all available query rewrite and optimization techniques are simply reused, which is a big advantage, because virtually all "sequential" optimizations also hold on the distributed setting (e.g., push selective operations to the beginning, exploit existing sort orders, etc.). Furthermore, the execution of query fragments within nodes of a cluster is carried out entirely by the Brackit engine. This approach transparently integrates MR into an existing data-processing environment, giving the flexibility to choose which parts of a computation are shipped to MR and which are executed locally, in the standard query engine.

A further advantage of Brackit is that it provides an extensible framework for specifying storage modules. The query processing logic is decoupled from specific storage implementations, and a well-defined interface is offered for communication between these two components. This interface allows storage-specific optimizations such as index scans and predicate push-down to be implemented and integrated transparently in the optimizer.

The flexibility provided by BrackitMR, both at the query language and system architecture levels, also contributes for filling the gap between MapReduce and database systems, moving towards a unified solution which embraces the best from both worlds.

The remainder of this paper is organized as follows. Section 2 introduces our extended version of the XQuery data model and describes our generic *collection framework*, which enables the integration of various kinds of data sources into a single query processing environment. Section 3 describes the process of mapping an XQuery plan into the MR programming model. Section 4 provides an experimental performance evaluation of BrackitMR. We demonstrate that BrackitMR is as fast as the competing approaches when it comes to relational processing. Finally, Section 5 concludes this paper.

2 Data Model

2.1 Motivation

One of the major complaints of users against XQuery is that its XML-based data model, albeit optimal for document-centric scenarios, is too cumbersome for data-centric applications. The flexibility of the XML format for representing data, and of XQuery for processing it, comes with a heavy impact on query conciseness and evaluation performance in scenarios where data is purely or partially relational. Such limitations contributed to the adoption of JSON as a format for representing simple collections of objects, trimming away the complexities of document-centric XML. JSON was quickly adopted in Web 2.0 applications, which often interact with relational databases or external service APIs using asynchronous HTTP requests (i.e., Ajax applications).

The major drawback of the JSON data model is that there is currently no standard query language that supports it. Driven by the popularity of *NoSQL* databases, established techniques of database query processing are mostly ignored in such systems, and complex queries are often implemented from scratch in the application layer. To overcome this drawback, several NoSQL products like MongoDB currently provide rather primitive query constructs. For complex analytical queries, developers must implement ad-hoc glue code to integrate such databases with MapReduce-based query engines like Pig and Hive.

Another obvious drawback of such JSON-based approaches is the lack of support for XML, which is fundamental for Web applications. Brackit overcomes these problems by simply extending the XQuery data model with support for JSON objects and arrays, an approach which is also proposed in the JSONiq specification [8]. The integration of JSON values in an XQuery engine is very simple, but it results in a tremendous practical value, as it integrates document- and data-centric query processing in a single environment.

A further advantage that contributes to the versatility of BrackitMR is its *collection framework*. It defines a common interface for the communication between query engine and different storage modules, thus enabling the transparent processing of several data sources using a unified language. The interface not only allows data to be read from and written into various data sources, but also supports basic storage-related optimizations such as filter and projection push-down in a generic manner.

2.2 JSON Support

The XQuery data model (XDM) is centered around *sequences*, which are ordered collections of *items*. Items can be either XML nodes or atomic values, but they cannot be sequences again, which means that nested sequences are not supported. A fundamental property of XQuery, which is the basis of its composability, is that an item is indistinguishable from a singleton sequence containing it. This means that all values in XQuery are sequences, and that every expression returns a sequence when evaluated.

To integrate JSON in our data model, we simply add two new kinds of items: *objects* and *arrays*. Objects are simple record structures which map attribute names into values. In XDM, such values are themselves sequences, thus enabling nested data structures. Like SQL, JSON defines a special NULL value, which has no equivalent concept in XQuery[1], and therefore we also define a special `null` value. Arrays are similar to sequences in which they are ordered collections of items, but because arrays are a kind of item, they can therefore be nested.

Note that our scheme simply "reuses" the atomic and XML values from the original data model. This means that our objects and arrays are more powerful than those of the original JSON specification, which support only numbers (with no distinction between integer, decimal, double, etc.), strings and Boolean values. XQuery, on the other hand, provides the complete type hierarchy of XML Schema, which also includes user-defined types. This also implies that XML nodes can be embedded inside objects and arrays, which gives great flexibility for mixed document- and data-centric workloads.

Figure 1 illustrates a query over two relational tables from the TPC-H benchmark, which are accessed as collections of JSON objects. For comparison, we show the equivalent SQL query on the right-hand side.

```
for $l in collection('lineitem')
for $o in collection('orders')
where $l=>orderkey = $o=>orderkey
  and $l=>shipdate >= '1995-01-01'
let $rf := $l=>returnflag
group by $rf
return { retflag: $rf,
         avg_price: avg($o=>totalprice) }
```

```
SELECT l.returnflag AS retflag,
  avg(o.totalprice) AS avg_price
FROM lineitem l, orders o
WHERE l.orderkey = o.orderkey
  AND l.shipdate >= '1995-01-01'
GROUP BY l.returnflag
```

Fig. 1. Example of XQuery expression with JSON support and equivalent SQL query

The query performs a join between the tables *lineitem* and *orders*, whereas the former is filtered by a predicate on its *shipdate* attribute. The tuples are then grouped by the *returnflag* attribute, which according to XQuery semantics must first be extracted into its own variable `$rf` before grouping. Finally, each return flag is returned together with the average of its associated order prices. In this query, the use of the JSON data model is demonstrated in expressions accessing object attributes with the `=>` operator. Furthermore, the query returns JSON objects using a constructor syntax that is equivalent to the string representation of JSON objects.

For brevity, we rely on this simple example to demonstrate the basic capabilities of JSON support in XQuery. For a more precise specification, we refer to the JSONiq language [8], which is implemented in the Zorba XQuery processor.

[1] In XQuery, the empty sequence is usually employed to denote the absence of a value, but this is not semantically equivalent to NULL. An empty sequence is an empty, but existing value, while NULL indicates the absence of a value.

2.3 Collection Framework

The main design goal of the Brackit[2] query engine is to provide a common framework for query compilation, optimization, and execution, abstracting away from specific storage modules. This abstraction is realized in the *collection framework*, which provides a Java interface that is implemented by specific storage modules. The query evaluation engine interacts with collection instances when the function `collection` is invoked, as in the example of Figure 1. In the XQuery specification, this function must return XML elements, but our implementation relaxes this constraint by allowing general XDM items to be returned. An item is represented by a Java interface as well, and hence storage modules are free to represent items in an efficient manner. A relational tuple, for instance, would be best represented internally as an array of atomic values, but it would still behave transparently, from the query engine perspective, as a JSON object.

In order to be found by the query engine, a collection must be registered in the metadata catalog, which in XQuery is modelled by the static context [12]. Because XQuery provides no standard mechanism for registering collections, our implementation introduces the `declare collection` primitive. It takes four parameters: the collection name as a string, the name of the class which implements the collection interface, and an optional URI for locating the collection within the implemented module, and the type of its items. Consider, for example, the following declaration of the collection *lineitem* of Figure 1:

```
declare collection lineitem of CSVCollection
at hdfs://lineitem.csv as object(type:lineitem)*;
```

It declares a collection called *lineitem*, which is instantiated using the class *CSVCollection*, an implementation for CSV files where each line is treated as a JSON object. The `at` keyword specifies a path, in this case in HDFS (i.e., Hadoop distributed file system), where the file can be located. The `as` keyword specifies the sequence type which is returned by the collection. The `object` keyword specifies that the items returned are JSON objects, each having the type `type:lineitem`, which is declared separately. The * symbol indicates that the collection contains zero or more items.[3] Because the original XQuery does not support JSON types, we must extend it with a type declaration primitive or schema import mechanisms. For brevity, we abstract the declaration of JSON types from our discussion. We refer to JAQL [3], which served as inspiration for JSON type declaration in our system.

Note that the type argument can be any valid sequence type, and thus we can use abstract types like `item()*` to leave the schema unspecified. Partial schema specifications can be implemented in XML Schema using complex types, and in JSON using the JAQL notation with wild-card markers like * and `?`.

[2] `http://www.brackit.org`

[3] The use of such cardinality symbols may seem unnecessary, but it is part of the universal sequence-type syntax of XQuery, which is also used in function arguments and results.

3 Compilation and Execution

3.1 Query Plans

A query is compiled in Brackit into a tree of expressions which is evaluated in a bottom-up manner. A FLWOR expression is the standard construct used for processing bulk data, and hence we focus on the compilation of FLWOR expressions only. When compiled, a FLWOR expression is represented by a tree of operators, just like a relational query plan. For an overview of how such plans are evaluated, we refer to the plan on the left-hand side in Figure 2, which is generated for the query in Figure 1.

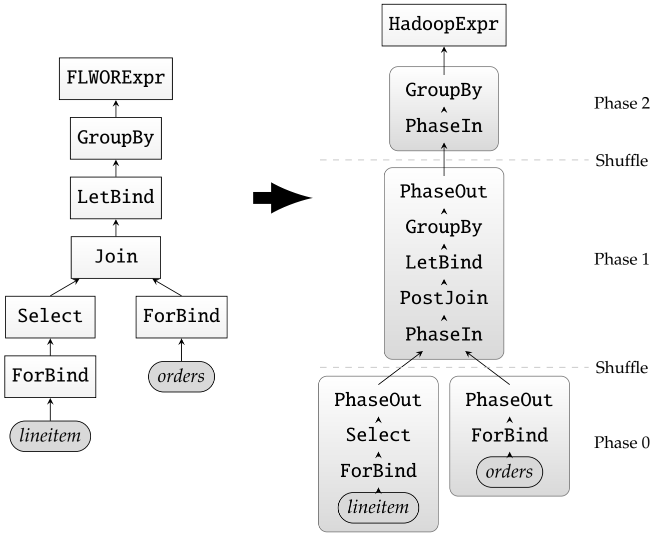


Fig. 2. Example of a query plan and its translation into task functions

At the top, we have a FLWOR expression node, which is responsible for evaluating the operators under it and delivering the final expression in the `return` clause as a result. Each operator in the plan consumes and produces a tuple of bound variables, in accordance to the semantics of FLWOR clauses [12]. These tuples, however, are not part of the XQuery data model, but internal structures which serve the exclusive purpose of evaluating operators. The `FLWORExpr` node, therefore, serves as a bridge between the iterator-based, set-oriented model of operators (i.e., the traditional model of relational query processing) and the sequence-based model of XQuery expressions.

The query starts with `ForBind` operators, which evaluate the collection function and bind each delivered item to a given variable, generating the initial tuples. Its functionality resembles that of a relational table scan. The input tuple of an operator contains the currently available variable bindings, whose values are used to evaluate expressions attached to each operator. Depending on the result of the evaluation, tuples are modified, discarded, grouped, or reordered. This corresponds to the classical mechanism used for query evaluation in relational systems, as established by the iterator model [5].

Each operator in the query plan is basically derived from a corresponding clause in the FLWOR expression. The `Join` operator is derived by our query rewrite engine from a sequence of two `ForBind` operators followed by a predicate on both their inputs. For a detailed description of each operator's semantics, we refer to the FLWOR expression specification in the XQuery standard [12].

3.2 Generalized MapReduce Model

The generation of MR jobs from query plans follows the same basic mechanism used in Hive, Pig, and JAQL. In previous work, we have discussed general mechanisms for compiling query languages into the MR programming model [9]. Our discussion of the mapping from XQuery to MR thus relies on the Generalized MapReduce model (GMR) introduced there. Nevertheless, we provide a brief explanation of the GMR model for completeness. It is derived from the original MR model by generalizing a few aspects of a job specification. Despite being a generalization, we emphasize that the model can be fully implemented in Hadoop, and therefore it is the same basic model used in Hive, Pig, and JAQL. The goal of our generalizations is simply to provide a more concise model for discussing the mapping of query plans into MR jobs.

The first, and most important, generalization of GMR is the use of *task functions* instead of the original map and reduce functions. As noted in [6], the Mapper and Reducer tasks—which iterate over the input key-value pairs and invoke the map or reduce function on each of them—are both implementations of the traditional `map` higher-order function of functional programming languages. The only difference between the map and reduce functions is that a reduce is applied on a list of values rather than on a single value, but if we consider a list as a proper value, the functions are essentially the same. Thus, we can specify a job as a sequence of Mapper-Shuffle-Mapper stages. Furthermore, we allow the user to specify the function which is executed by a task for each partition of the input, which we refer to as *task function.* This is in contrast to the original map and reduce functions, which are called for each key-value pair in an input partition. With this generalization, MR jobs can be specified as a pair of generalized task functions—one executed before and the other after the Shuffle stage.

The further generalizations enable the specification of jobs as arbitrary trees of task functions. First, we allow jobs to contain multiple map functions and a single reduce, or, equivalently in our generalization, multiple task functions before a Shuffle stage. The intermediate Shuffle keys are then grouped regardless of which task function produced them, and a tag is appended to allow separating the key-value pairs again on the reduce side. This is a standard technique for performing joins in Hadoop, discussed in mored detail in [13]. Furthermore, we can allow arbitrarily long sequences of task functions interleaved with Shuffle stages. This is necessary to implement complex operations that require multiple MR jobs. In the original MR model, this can be simulated by executing jobs with empty map functions, which simply propagate the key-value pairs generated by a preceding reduce to another Shuffle stage. In our GMR model, this finally allows the specification of trees of task functions, where between each level of the tree there is a Shuffle stage.

The GMR model provides an ideal level of abstraction for discussing query processing in MR, because it has greater resemblance to the model of query plans and the iterator model. On the other hand, the model is not over-generalized, such that the MR nature would be lost. It can be fully implemented in Hadoop, and it also applies to query compilation in Pig, Hive, and JAQL. For further details on GMR, we refer to [9].

3.3 Mapping Query Plans to GMR

The use of the GMR model makes it very natural to map query plans into MR jobs. The process is based on the classification of operators into *blocking* and *non-blocking*, and it is illustrated in Figure 2 for our example query. The idea is to convert all blocking operators (in our case `Join`, `GroupBy`, and `OrderBy`) into an MR Shuffle, which is the only blocking operation provided by the MR framework. The Shuffle performs a grouping operation on multiple inputs, followed by a repartition of the key-value pairs across the worker nodes in the cluster. It is generic enough to implement all standard blocking operations, but it requires, in some cases, special non-grouping post-processing operators which restore the semantics of the original operator after a Shuffle.

A sequence of non-blocking operators, on the other hand, is "packed" into a task function, as illustrated in Figure 2. Therefore, the execution of a task function across worker nodes, which we refer to as a *phase*, consists of compiling and executing the operators using the standard Brackit query processor, based on the iterator model. In order to generate proper key-value pairs from tuples and to build tuples back after shuffling, the special operators `PhaseOut` and `PhaseIn` are used. A key in this case is simply a subset of the tuple's fields (i.e., a *sub-tuple*), whereas a value is composed of the remaining fields.

To clarify the compilation and execution process, we make use of the example in Figure 2, starting at the root node `FLWORExpr`. The FLWOR expression itself is compiled into a `HadoopExpr`, which executes the contained query plan in the MR framework and then brings the generated results back to the client machine, serving as a "bridge" between local and distributed evaluation. Note that this approach allows a hybrid query evaluation mechanism in which only the heavy parts of an XQuery program—namely those that access collections on distributed storage—are shipped for execution in MR. This means that a `HadoopExpr` may, for example, compute aggregations on a large dataset and return the results as a sequence to the parent expression, which may generate an XML report or store the results in a local database.

The evaluation of the plan starts at the leaf task functions, which must start with a `ForBind` operator. MR jobs usually exploit the data parallelism of the input datasets, by processing each key-value pair independently. This is exactly the functionality of the `for` clause in a FLWOR expression, namely to iterate over a sequence of items, bind it to a variable, and execute the following clauses once for each operator. The non-blocking operators are then evaluated until a `PhaseOut` is reached. It extracts specific fields of the input tuple into a sub-tuple which will be used as key in the Shuffle stage. The fields to be extracted depend of course on the blocking operator being executed. In the example, the two leaf task functions serve as input to a `Join`, and hence the keys extracted are the join keys, namely the *orderkey* attribute. Note that the variables `$l` and `$o` contain the whole JSON objects, so their *orderkey* attributes have to be extracted into their own variables using a `LetBind`, which we omitted here for space reasons.

The task function in the second phase of our example starts with a `PhaseIn` operator, which will rebuild the original tuples from the key and value sub-tuples. The `PostJoin` separates the input tuples based on the tag value and joins them locally. Our implementation is based on a hash-join algorithm, and so the tuples are repartitioned by the Shuffle based on a hash value of the keys. Furthermore, tuples within each partition are sorted by the tag value, so that all tuples from the right input come first and are used to build the hash table.

After the `Join`, the `LetBind` operator is executed and its outputs are fed into a `GroupBy` operator. Note, despite being a blocking operator, the `GroupBy` is executed inside the task function. It pre-aggregates values locally before the global grouping that occurs in the Shuffle. This functionality corresponds to the MR *combine* function, but instead it reuses the `GroupBy` operator provided by the Brackit engine. The local `GroupBy` is a *partition-wide* blocking operator [9], meaning that it blocks the evaluation of a single partition only. In general, non-blocking as well as partition-wide blocking operators can be executed inside a task function. Note that the `PostJoin` operator also belongs to this class.

The last task function then groups the pre-aggregated tuples to compute the global aggregations, and another `GroupBy` is used for that end. Because it is the last operator in the evaluation, its results are written to distributed storage and retrieved by the `HadoopExpr` instance. It is also possible to directly transmit the output tuples back to the client, but this would require some synchronization mechanism in order to keep the sort order of the output.

Note, because Hadoop does not directly implement the GMR model, the task function tree of Figure 2 requires one MR job for the execution of phases 0 and 1, inside Mapper and Reducer tasks, respectively, and a second job with an empty Mapper is then required to process phase 2.

3.4 Processing XML Data

So far, we have discussed examples based only on relational datasets modelled as JSON collections. However, one major advantage of BrackitMR towards the related approaches is its native support for XML data. This feature is primordial in scenarios like Web page crawling, generation of HTML reports or SVG graphs, and many other typical scenarios in the Web 2.0. Our generic collection framework combined with the transparent integration of JSON in XDM enables queries which mix JSON and XML data sources, and because arbitrary items can be nested inside JSON objects, queries can also generate JSON documents or relational tables which contain embedded XML data.

BrackitMR, however, only supports XML data stored in collections, because they exhibit the degree of data parallelism required in MR processing. If large distributed XML documents are to be processed in MR, we need to provide the query engine with knowledge about how XML fragments are partitioned. Such partitioning strategies heavily depend on the particular structure of each document, and the absence of schema information complicates the problem even further. Even if the partitioning scheme is provided, the query still needs to be checked at compile time to detect paths that cross partition boundaries. Given the management complexity, large XML data sources rarely occur as a single document, but this does not represent a major limitation for BrackitMR. If we consider the domain of MR processing, this becomes an even smaller concern, because typical MR tasks like log processing or document crawling usually deal with large collections of small and independent items.

3.5 Limitations

XQuery is in fact a Turing-complete functional programming language, and thus it provides general recursive functions, arbitrarily nested FLWOR expressions, as well as several constructs that rely on strict sequential processing. Because MR provides a simple programming model, suited only for data-parallel computations, it is not possible to cover the complete XQuery standard in BrackitMR. Obvious limitations are, for instance, expressions that depend on sequential evaluation, like the `count` clause in FLWOR expressions or the position variable binding using the `at` keyword inside `for` clauses.

A further limitation of our current prototype is related to dependent subqueries. When evaluated naively, a nested-loops computation is required, which is catastrophic in the MR scenario, given the extremely high latencies intrinsic to the batch processing model. The Brackit query engine extracts arbitrarily nested sub-queries into a single unnested pipeline, using a technique referred to as *pipeline unnesting*. Using this feature, described in detail in [2], nested query semantics is simulated by left outer joins and grouping operations on additional *count variables*, which keep track of the position of a tuple within an iteration. Furthermore, it requires operators to keep track of “empty” iterations that emerge when sub-queries do not deliver any tuples. The unnesting feature is currently not supported in BrackitMR, but because the rewrite rules of the standard Brackit engine can be simply reused, it is not a conceptual limitation.

A further use case in XQuery is the use of recursive functions. Such functions which access collections require an evaluation model that supports fixed point computations. The technique required to implement recursive queries is essentially the same used in recursive SQL, and one approach for XQuery was proposed in [1]. Our current prototype does not implement the technique. However, none of the related approaches support recursive queries, and so the use case remains a corner stone for MR processing.

Aggregate task:

```
for $l in collection('lineitem')
let $month :=
      substring($l=>shipdate, 1, 7)
group by $month
return { month: $month,
    price: sum($l=>extendedprice) }
```

Join task:

```
for $l in collection('lineitem'),
    $o in collection('orders')
where $l=>orderkey eq $o=>orderkey
  and $l=>shipdate gt '1994-12-31'
  and $o=>totalprice gt 70000.00
return { o: $o=>orderkey,
         l: $l=>linenumber }
```

TPC-H Q3 Task:

```
for $c in collection("customer"),
    $o in collection("orders"),
    $l in collection("lineitem")
where $c=>custkey = $o=>custkey
    and $l=>orderkey = $o=>orderkey
    and $o=>orderdate < "1995-09-15"
    and $l=>shipdate > "1995-09-15"
let $orderkey := $l=>orderkey,
    $orderdate := $o=>orderdate,
    $shippriority := $o=>shippriority,
    $discounted := $l=>extendedprice
      * (1 - $l=>discount)
group by $orderkey,
    $orderdate,
    $shippriority
let $revenue := sum($discounted)
order by $revenue
return { order_key: $orderkey,
      revenue: $revenue,
      order_date: $orderdate,
      ship_priority: $shippriority }
```

Fig. 3. Queries used in the experiments

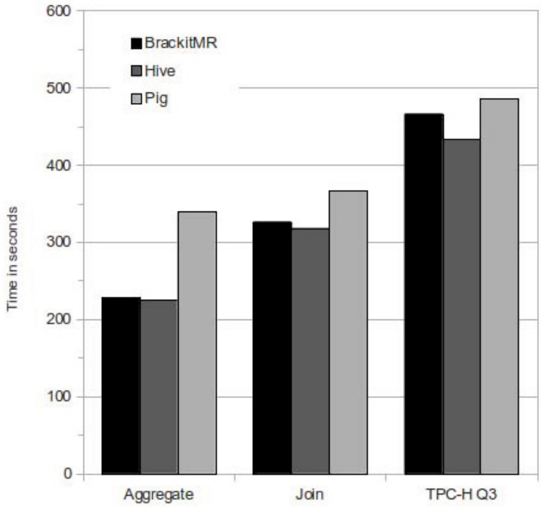
4 Experiments

To measure the performance of BrackitMR, we ran experiments based on the TPC-H dataset with a size of 10 GB, stored in plain CSV files. The experiments were run on a small cluster with 5 worker nodes and a separate master. In order to ensure similar conditions, we have tuned the Hadoop jobs to use the same number of Mapper and Reducer tasks. We also disabled compression of data shipped between Mapper and Reducer tasks.

We compared the execution times with Pig and Hive. JAQL was not included because its development was moved to a proprietary data warehouse system, and thus its open-source release is not being maintained anymore. It also depends on a discontinued version of Hadoop. However, published measurements comparing JAQL with Pig and Hive have shown that it is outperformed in all tests [10].

We used three queries in our experiments, shown in Figure 3. The first one is an aggregation task on the *lineitem* table, which computes the sum of item prices for each month. The second is a Join task, which filters and joins the *lineitem* and *orders* tables. Last, we ran a slightly modified version of the official TPC-H query number 3. Figure 4 shows the queries expressed in XQuery and the measured execution times in seconds.

The optimization techniques implemented in BrackitMR make use of the collection framework to push down filters and projections. After close inspection of the data produced at each phase of the computation, we conclude that the same



	Brackit	Hive	Pig
Aggregate	228	225	339
Join	326	318	366
TPC-H Q3	466	433	486

Measured times in seconds

Fig. 4. Query execution times in seconds

techniques are employed by Hive. Pig, however, does not perform automatic projection, and so the queries were manually modified so that unused columns were discarded.

The experiments show that Hive is the fastest system for the three tasks, followed closely by BrackitMR and then by Pig. However, the difference between BrackitMR and Hive for each of the queries was 1.3%, 2,5%, and 7.6%, respectively. Thus, it empirically confirms our claim that despite having a more generic data model and more expressiveness in the query language, BrackitMR does exhibit the same performance as the state-of-the art approaches.

5 Conclusion

We have developed BrackitMR, an extension of the Brackit query engine which executes XQuery FLWOR expressions in the MR framework. In comparison to existing approaches for query processing in MR, our system relies on an already established, flexible query language. Despite having similar characteristics, such as a semi-structured data model and a more flexible means to compose operations, languages like Hive or PigLatin are currently only used in the context of MR, whereas XQuery is widespread on varying application scenarios from database systems to the Web. Our approach of query engine reuse represents an elegant solution, which simplifies the MR computational model, making greater use of long-established query processing logic.

As a model for distributed query processing, MR actually has significant drawbacks, especially if we consider complex queries with multiple non-blocking operators. The main reason is that it simulates the GMR model using identity Mapper tasks. These tasks represent a major performance bottleneck, because the output of a Reducer phase is always written to the distributed file system, which in most scenarios has a replication factor of three. This output is then fed into the identity Mapper tasks, which simply write the whole data unmodified to the local file systems of the worker nodes. Only then a Shuffle phase can start to group data and perform the operation required by the non-blocking operator. Note that a much more efficient variant would allow the Reducer output to stay within local file system boundaries and be fetched directly by the shuffle tasks.

BrackitMR is at an early stage, and the goal of this paper was simply to show the flexibility potential of XQuery combined with our modular architecture. A crucial requirement, however, is that the higher flexibility does not incur a higher cost in performance. Our goal is that whenever the conditions for efficient data processing are met—in this case, relational structures with full schema information—the query engine must perform as fast as an approach designed specifically for those conditions. We believe this has been achieved so far, as reported in our experiments.

References

1. Afanasiev, L., Grust, T., Marx, M., Rittinger, J., Teubner, J.: An Inflationary Fixed Point Operator in XQuery. In: ICDE Conference, pp. 1504–1506. IEEE (2008)
2. Bächle, S.: Separating Key Concerns in Query Processing – Set Orientation, Physical Data Independence, and Parallelism. Ph.D. thesis, University of Kaiserslautern, Germany (2012)
3. Beyer, K.S., Ercegovac, V., Gemulla, R., Balmin, A., Eltabakh, M.Y., Kanne, C.C., Özcan, F., Shekita, E.J.: Jaql: A Scripting Language for Large-Scale Semistructured Data Analysis. PVLDB 4(12), 1272–1283 (2011)
4. Dean, J., Ghemawat, S.: MapReduce: A Flexible Data Processing Tool. Commun. ACM 53(1), 72–77 (2010)
5. Graefe, G.: Query Evaluation Techniques for Large Databases. ACM Comput. Surv. 25(2), 73–170 (1993)
6. Lämmel, R.: Google's MapReduce Programming Model – Revisited. Sci. Comput. Program. 70(1), 1–30 (2008)
7. Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A.: Pig Latin: A Not-So-Foreign Language for Data Processing. In: SIGMOD Conference, pp. 1099–1110 (2008)
8. Robie, J., Brantner, M., Florescu, D., Fourny, G., Westmann, T.: JSONiq: XQuery for JSON, JSON for XQuery, pp. 63–72 (2012)
9. Sauer, C., Härder, T.: Compilation of Query Languages into MapReduce. Datenbank-Spektrum 13(1), 5–15 (2013)
10. Stewart, R.J., Trinder, P.W., Loidl, H.-W.: Comparing High Level MapReduce Query Languages. In: Temam, O., Yew, P.-C., Zang, B. (eds.) APPT 2011. LNCS, vol. 6965, pp. 58–72. Springer, Heidelberg (2011)
11. Thusoo, A., Sarma, J.S., Jain, N., Shao, Z., Chakka, P., Zhang, N., Anthony, S., Liu, H., Murthy, R.: Hive – A Petabyte Scale Data Warehouse using Hadoop. In: ICDE Conference, pp. 996–1005 (2010)
12. W3C: XQuery 3.0: An XML Query Language (2011), http://www.w3.org/TR/xquery-30/
13. White, T.: Hadoop - The Definitive Guide: Storage and Analysis at Internet Scale, 2nd edn. O'Reilly (2011)