A Parallel and Scalable Processor for JSON Data

Christina Pavlopoulou
University of California, Riverside
cpavl001@ucr.edu

E. Preston Carman, Jr
University of California, Riverside
ecarm002@ucr.edu

Till Westmann
Couchbase
tillw@apache.org

Michael J. Carey
University of California, Irvine
mjcarey@ics.uci.edu

Vassilis J. Tsotras
University of California, Riverside
tsotras@cs.ucr.edu

ABSTRACT

Increasing interest in JSON data has created a need for its efficient processing. Although JSON is a simple data exchange format, its querying is not always effective, especially in the case of large repositories of data. This work aims to integrate the JSONiq extension to the XQuery language specification into an existing query processor (Apache VXQuery) to enable it to query JSON data in parallel. VXQuery is built on top of Hyracks (a framework that generates parallel jobs) and Algebricks (a language-agnostic query algebra toolbox) and can process data on the fly, in contrast to other well-known systems which need to load data first. Thus, the extra cost of data loading is eliminated. In this paper, we implement three categories of rewrite rules which exploit the features of the above platforms to efficiently handle path expressions along with introducing intra-query parallelism. We evaluate our implementation using a large (803GB) dataset of sensor readings. Our results show that the proposed rewrite rules lead to efficient and scalable parallel processing of JSON data.

1 INTRODUCTION

The Internet of Things (IoT) has enabled physical devices, buildings, vehicles, smart phones and other items to communicate and exchange information in an unprecedented way. Sophisticated data interchange formats have made this possible by leveraging their simple designs to enable low overhead communication between different platforms. Initially developed to support efficient data exchange for web-based services, JSON has become one of the most widely used formats evolving beyond its original specification. It has emerged as an alternative to the XML format due to its simplicity and better performance [28]. It has been used frequently for data gathering [22], motion monitoring [20], and in data mining applications [24].

When it comes time to query a large repository of JSON data, it is imperative to have a scalable system to access and process the data in parallel. In the past there has been some work on building JSONiq add-on processors to enhance relational database systems, e.g. Zorba [2]. However, those systems are optimized for single-node processing.

More recently, parallel approaches to support JSON data have appeared in systems like MongoDB [10] and Spark [7]. Nevertheless, these systems prefer to first load the JSON data and transform them to their internal data model formats. On the other hand systems like Sinew [29] and Dremel [27] cannot query raw JSON data. They need a pre-processing phase to convert the input file into a readable binary for them (typically Parquet [3]). They can then load the data, transform it to their internal data model and proceed with its further processing. The above efforts are examples of systems that can process JSON data by converting it to their data format, either automatically, during the loading phase, or manually, following the pre-processing phase. In contrast, our JSONiq processor can immediately process its JSON input data without any loading or pre-processing phases. Loading large data files is a significant burden for the overall system's execution time as our results will show in the experimental section. Although, for some data, the loading phase takes place only in the beginning of the whole processing, in most real-time applications, it can be a repetitive action; data files to be queried may not always been known in advance or they may be updated continuously.

Instead of building a JSONiq parallel query processor from scratch, given the similarities between JSON and XQuery, we decided to take advantage of Apache VXQuery [4, 17], an existing processor that was built for parallel and scalable XQuery processing. We chose to support the JSONiq extension to XQuery language [8] to provide the ability to process JSON data. XQuery and JSONiq have certain syntax conflicts that need to be resolved for a processor to support both of them, so we enhanced VXQuery with the *JSONiq extension* to the XQuery language, an alteration of the initial JSONiq language designed to resolve the aforementioned conflicts [9].

In extending Apache VXQuery, we introduce three categories of JSONiq rewrite rules (*path expression*, *pipelining*, and *group-by* rules) to enable parallelism via pipelining and to minimize the required memory footprint. A useful by-product of this work is that the proposed group-by rules turn out to apply to both XML and JSON data querying.

Through experimentation, we show that the VXQuery processor augmented with our JSoniq rewrite rules can indeed query JSON data without adding the overhead of the loading phase used by most of the state-of-the art systems.

The rest of the paper is organized as follows: Section 2 presents the existing work on JSON query processing, while Section 3 outlines the architecture of Apache VXQuery. Section 4 introduces the specific optimizations applied to JSON queries and how they have been integrated into the current version of VXQuery. The experimental evaluation appears in Section 5. Section 6 concludes the paper and presents directions for future research.

2 RELATED WORK

Previous work on querying data interchange formats has primarily focused on XML data [26]. Nevertheless there has been considerable work for querying JSON data. One of the most popular JSONiq processors is Zorba [2]. This system is basically a virtual machine for query processing. It processes both XML and JSON data by using the XQuery and JSONiq languages respectively. However, it is not optimized to scale onto multiple nodes with multiple data files, which is the focus of our work. In

contrast, Apache VXQuery is a system that can be deployed on a multi-node cluster to exploit parallelism.

A few parallel approaches for JSON data querying have emerged as well. These systems can be divided into two categories. The first category includes SQL-like systems such as Jaql [14], Trill [18], Drill [6], Postgres-XL [11], MongoDB [10] and Spark [13], which can process raw JSON data. Specifically, they have been integrated with well-known JSON parsers like Jackson [1]. While the parser reads raw JSON data, it converts it to an internal (table-like) data model. Once the JSON file is in a tabular format, it can then been processed by queries. Our system can also read raw JSON data, but it has the advantage that it does not require data conversion to another format since it directly supports JSON's data model. Queries can thus be processed on the fly as the JSON file is read. It is also worthwhile mentioning that Postgres-XL (a scalable extension to PostgreSQL [12]) has a limitation on how it exploits its parallelism feature. Specifically, while it scales on multiple nodes it is not designed to scale on multiple cores. On the other hand, our system can be multinode and multicore at the same time. In the experimental section we show how our system compares with two representatives from this category (MongoDB and Spark).

We note that AsterixDB [5], can process JSON data in two ways. It can either first load the file internally (like the systems above) or, it can access the file as external data without the need of loading it. However, in both cases and in contrast to our system, AsterixDB needs to convert the data to its internal ADM data model. In our experiments we compare VXQuery with both variations of AsterixDB.

Systems in the second category (e.g. Sinew [29], Argo [19] and Oracle's system [25]) cannot process raw JSON data and thus need an additional pre-processing phase (hence an extra overhead than the systems above). During that phase, a JSON file is converted to a binary or Parquet ([3]) file that is then fed to the system for further transformation to its internal data model before query processing can start.

Systems like Spark and Argo process their data in-memory. Thus, their input data sizes are limited by a machine's memory size. Recently, [23] presents an approach that pushes the filters of a given query down into the JSON parser (Mison). Using data-parallel algorithms, like SIMD vectorization and Bitwise Parallelism, along with speculation, data not relevant to the actual query is filtered out early. This approach has been added into Spark and improves its JSON performance. Our work also prunes irrelevant data, but does so by applying rewrite rules. Since the Mison code is not available yet, we could not compare with them in detail; we also need to note that Mison is just a parallel JSON parser for JSON data. In contrast, VXQuery is an integrated processor that can handle the querying of both JSON and XML data (regardless of how complex the query is).

As opposed to the aforementioned systems, our work builds a new JSONiq processor that leverages the architecture of an existing query engine and achieves high parallelism and scalability via the employment of rewrite rules.

3 APACHE VXQUERY

Apache VXQuery was built as a query processing engine for XML data implemented in Java. It is built on top of two other frameworks, namely the Hyracks platform and the Algebricks layer. Figure 1, also, shows AsterixDB [5], which uses the same infrastructure.

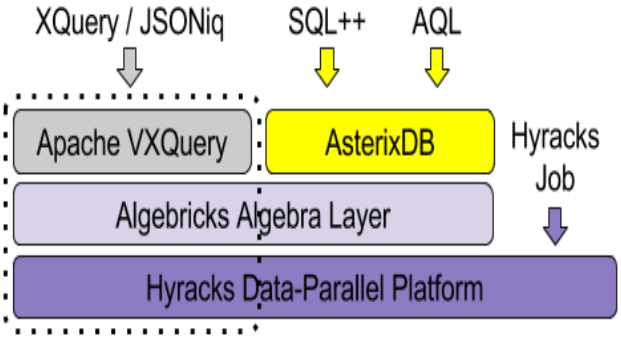

Figure 1: The VXQuery Architecture

3.1 Infrastructure

The first layer is *Hyracks* [16], which is an abstract framework responsible for executing dataflow jobs in parallel. The processor operating on top of Hyracks is responsible for providing the partitioning scheme while Hyracks decides how the resulting job will be distributed. Hyracks processes data in partitions of contiguous bytes, moving data in fixed-sized frames that contain physical records, and it defines interfaces that allow users of the platform to specify the data-type details for comparing, hashing, serializing and de-serializing data. Hyracks provides built-in base data types to support storing data on local partitions or when building higher level data types.

The next layer, *Algebricks* [15], takes as input a logical query plan and, via built-in optimization rules that it provides, converts it to a physical plan. Apart from the transformation, the rules are responsible for making the query plan more efficient. In order to achieve this efficiency, Algebricks allows the processor above (in this case Apache VXQuery) to provide its own language specific rewrite rules.

The final layer, *Apache VXQuery* [4, 17], supports a XQuery processor engine. To build a JSONiq processor, we used the JSONiq extension to XQuery specifications. Specifically, we focused mostly on implementing all the necessary modules to successfully parse and evaluate JSONiq queries. Additionally, several modules were implemented to enable JSON file parsing and support an internal in-memory representation of the corresponding JSON items.

The resulting JSONiq processor accepts as input the original query, in string form, and converts it to an abstract syntax tree (AST) through its query parser. Then, the AST is transformed with the help of VXQuery's translator to a logical plan, which becomes the input to Algebricks.

As mentioned above, VXQuery uses Hyracks to schedule and run data parallel jobs. However, Hyracks is a data-agnostic platform, while VXQuery is language-specific. This creates a need for additional rewrite rules to exploit Hyracks' parallel properties for JSONiq. If care is not taken, the memory footprint for processing large JSON files can be prohibitively high. This can make it impossible for systems with limited memory resources to efficiently support JSON data processing. In order to identify opportunities for parallelism as well as to reduce the runtime memory footprint, we need to examine in more depth the characteristics of the JSON format as well as the supported query types.

3.2 Hyracks Operators

We first proceed with a short description of the Hyracks logical operators that we will use in our query plans.

- **EMPTY-TUPLE-SOURCE:** outputs an empty tuple used by other operators to initiate result production.
- **DATASCAN:** takes as input a tuple and a data source and extends the input tuple to produce tuples for each item in the source.
- **ASSIGN:** executes a scalar expression on a tuple and adds the result as a new field in the tuple.
- **AGGREGATE:** executes an aggregate expression to create a result tuple from a stream of input tuples. The result is held until all tuples are processed and then returned in a single tuple.
- **UNNEST:** executes an unnesting expression for each tuple to create a stream of output tuples per input.
- **SUBPLAN:** executes a nested plan for each tuple input. This plan consists of an AGGREGATE and an UNNEST operator.
- **GROUP-BY:** executes an aggregate expression to produce a tuple for each set of items having the same grouping key.

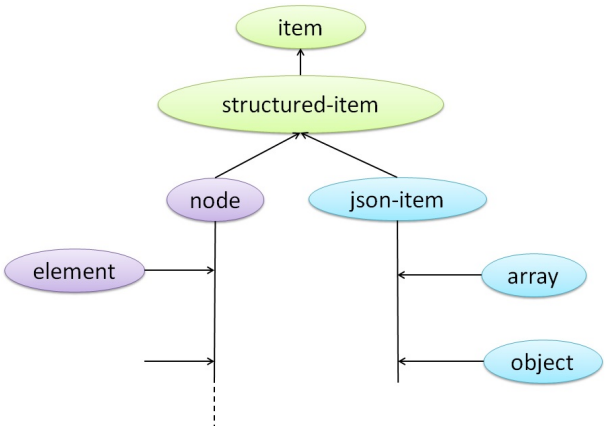

Figure 2: XML vs JSON structure

It is imperative for understanding this work to describe the representation along with the navigation expressions of JSON items according to the JSONiq extension to the XQuery specification. A json-item can be either an array or an object, in contrast to an XML structure, which consists of multiple nodes as described in Figure 2. An array consists of an ordered list of items (members), while an object consists of a set of pairs. Each pair is represented by a key and a value. The following is the terminology used for JSONiq navigation expressions:

- **Value:** for an array it yields the value of a specified (by an index) array element, while for an object it yields the value of a specified (by a field name) key.
- **Keys-or-members:** for an array it outputs all of its elements, and for an object it outputs all of its keys.

4 JSON QUERY OPTIMIZATION

The JSONiq rewrite rules are divided into three categories: the Path Expression, Pipelining, and Group-by Rules. The first category removes some unused expressions and operators, as well as streamlining the remaining path expressions. The second category reduces the memory needs of the pipeline. The last category focuses on the management of aggregation, which also contains the group-by feature (added to VXQuery in the XQuery 3.0 specification). For all our examples, we will consider the bookstore structure example depicted in Listing 1.

```
{
  "bookstore": {
    "book": [
      {
        "-category": "COOKING",
        "title": "Everyday Italian",
        "author": "Giada De Laurentiis",
        "year": "2005",
        "price": "30.00"
      },
    ...
    ]
 }
```

Listing 1: Bookstore JSON File

4.1 Path Expression Rules

The goal of the first category of rules is to enable the unnesting property. This means that instead of creating a sequence of all the targeted items and processing the whole sequence, we want to process each item separately as it is found. This rule opens up opportunities for pipelining since each item is passed to the next stage of processing as the previous step is completed.

```
json-doc("books.json")("bookstore")("book")()
```

Listing 2: Bookstore query

The example query in Listing 2 asks for all the books appearing in the given file. Specifically, it reads data from the JSON file ("book.json") and then, the *value* expression is applied twice, once for the bookstore object (("bookstore")) and once for the book object (("book")). In this way, it is ensured that only the matching objects of the file will be stored in memory. The value of the book object is an array, so the *keys-or-members* expression (()) applied to it returns all of its items. To process this expression, we first store in a tuple all of the objects from the array and then we iterate over each one of them. The result that is distributed at the end is each book object separately.

```
DISTRIBUTE-RESULT( $$9 )
UNNEST( $$9:iterate($$8) )
ASSIGN( $$8:(keys-or-members($$2)) )
ASSIGN( $$2:value(value(json-doc(promote(data("books.json"),
string)),"bookstore"),"book"))
EMPTY-TUPLE-SOURCE
```

Figure 3: Original Query Plan

In more detail, we can describe the aforementioned process in terms of a logical query plan that is returned from VXQuery (Figure 3). It follows a bottom-up flow, so the first operator in the query plan is the EMPTY-TUPLE-SOURCE leaf operator. The empty tuple is extended by the following ASSIGN operator, which consists of a *promote* and a *data* expression to ensure that the json-doc argument is a string. Also, the two *value* expressions inside it verify that only the book array will be stored in the tuple.

The next two operators depict the two steps of the processing of the *keys-or-members* expression. The first operator is an ASSIGN, which evaluates the expression to extend its input tuple. Since this expression is applied to an array, the returned tuple includes all of the objects inside the array. Then, the UNNEST operator applies an iterate expression to the tuple and returns a stream of tuples including each object from the array.

The final step according to the query plan is the distribution of each object returned from the UNNEST. From the analysis above, we can observe that there are opportunities to make the logical plan more efficient. Specifically, we observe that there is no need for two processing steps of *keys-or-members*.

Originally, the tuple with all the book objects produced by the keys-or-members expression flows into the UNNEST operator whose iterate expression will return each object in a separate tuple. Instead, we can merge the UNNEST with the keys-or-members expression. That way, each book object is returned immediately when it is found.

Finally, to further clean up our query plan, we can remove the *promote* and *data* expressions included in the first ASSIGN operator. The fully optimized logical plan is depicted in Figure 4.

```
DISTRIBUTE-RESULT( $$13 )
UNNEST( $$13:keys-or-members($$2))
ASSIGN( $$2:value(value(json-doc("books.xml") ,"bookstore"),"book")
EMPTY-TUPLE-SOURCE
```

Figure 4: Updated Query Plan

The new and more efficient plan opens up opportunities for pipelining since when a matching book object is found, it is immediately returned and, at the same time, passed to the next stage of the process.

4.2 Pipelining Rules

This type of rule builds on top of the previous rule set and considers the use of the DATASCAN operator along with the way to access partitioned-parallel data. The sample query that we use is depicted in Listing 3.

```
collection("/books")("bookstore")("book")()
```

Listing 3: Bookstore Collection Query

According to the query plan in Figure 5, the ASSIGN operator takes as input data source a collection of JSON files, through the *collection* expression. Then, UNNEST *iterate* iterates over the collection and outputs each single file. The two value expressions integrated into the second ASSIGN output a tuple source filled with all the book objects produced by the whole collection. The last step of the query plan (created in the previous section) is implemented by the keys-or-members expression of the UNNEST operator, which outputs each single object separately.

The input tuple source generated by the *collection* expression corresponds to all the files inside the collection. This does not help the execution time of the query, since the result tuple can be huge. Fortunately, Algebricks offers its DATASCAN operator, which is able to iterate over the collection and forwards to the next operator each file separately. To accomplish this procedure, DATASCAN replaces both the ASSIGN *collection* and the UNNEST *iterate*.

```
DISTRIBUTE-RESULT( $$13 )
UNNEST( $$13:keys-or-members($$6))
ASSIGN( $$6:value(value($$4,"bookstore"),"book"))
UNNEST( $$4:iterate($$2))
ASSIGN( collection("/books"), $$2)
EMPTY-TUPLE-SOURCE
```

Figure 5: Query Plan for a Collection

```
DISTRIBUTE-RESULT( $$13 )
UNNEST( $$13:keys-or-members($$4))
ASSIGN( $$4:value(value ( $$2,"bookstore"),"book"))
DATASCAN( collection("/books"), $$2)
EMPTY-TUPLE-SOURCE
```

Figure 6: Introduction of DATASCAN

By enabling Algebrick's DATASCAN, apart from pipeline improvement, we also achieve partitioned parallelism. In Apache VXQuery, data is partitioned among the cluster nodes. Each node has a unique set of JSON files stored under the same directory specified in the *collection* expression. The Algebricks' physical plan optimizer uses these partitioned data property details to distribute the query execution. Adding these properties allows Apache VXQuery to achieve partitioned-parallel execution without any user-level parallel programming.

To further improve pipelining, we can produce even smaller tuples. Specifically, we extend the DATASCAN operator with a second argument (here it is the book array). This argument defines the tuple that will be forwarded to the next operator.

In the updated query plan (Figure 6), the newly inserted DATASCAN is followed by an ASSIGN operator. Inside it, the two *value* expressions populate the tuple source with all the book objects of the file fetched from DATASCAN. We can merge the value expressions with DATASCAN by adding a second argument to it. As a result, the output tuple, which was previously filled with each file, is now set to only have its book objects (Figure 7).

```
DISTRIBUTE-RESULT( $$13 )
UNNEST( $$13:keys-or-members($$4))
DATASCAN( collection("/books"), $$4, ("bookstore")("book") )
EMPTY-TUPLE-SOURCE
```

Figure 7: Merge *value* with DATASCAN Operator

At this point, we note that by building on the previous rule set, both the query's efficiency and the memory footprint can be further improved. In the query plan in Figure 7, DATASCAN *collection* is followed by an UNNEST whose *keys-or-members* expression outputs a single tuple for each book object of the input sequence.

```
DISTRIBUTE-RESULT( $$4 )
DATASCAN( collection("/books"), $$4, ("bookstore")("book")() )
EMPTY-TUPLE-SOURCE
```

Figure 8: Merge keys-or-members with Datascan Operator

This sequence of operators gives us the ability to merge DATASCAN with *keys-or-members* by extending its second argument.

Figure 8 shows this action, whose result is the fetching of even smaller tuples to the next stage of processing. Specifically, instead of storing in DATASCAN's output tuple a sequence of all the book objects of each file in the collection, we store only one object at a time. Thus, query's execution time is improved and Hyracks' dataflow frame size restriction is satisfied.

```
for $x in collection("/books")("bookstore")
("book")()
group by $author:=$x("author")
return count($x("title"))
```

Listing 4: Bookstore Count Query

4.3 Group-by Rules

The last category of rules can be applied to both XML and JSON queries since the group-by feature is part of both syntaxes. Group-by can activate rules enabling parallelism in aggregation queries.

```
for $x in collection("/books")("bookstore")
("book")()
group by $author:=$x("author")
return count(for $j in $x return $j("title"))
```

Listing 5: Bookstore Count Query (2nd form)

The example query that we will use to show how our rules affect aggregation queries is in Listings 4 and 5. Both forms of the query read data from a collection with book files, group them by author, and then return the number of books written by each author.

```
DISTRIBUTE-RESULT( $$20 )
UNNEST($$20:iterate($$19))
ASSIGN($$19:count(value($$18, "title")))
ASSIGN($$18:treat(item,$$16))
GROUP-BY($$17:0->$$14){
   AGGREGATE($$16:create_sequence($$13))
}
ASSIGN($$14:data(value($$13, "author")))
DATASCAN( collection("/books"), $$13, ("bookstore")("book")( ))
EMPTY-TUPLE-SOURCE
```

Figure 9: Query Plan with Count Function

In Figure 9, the DATASCAN *collection* passes a tuple, for one book object at a time, to ASSIGN. The latter applies the *value* expression to acquire the author's name for the specific object. GROUP-BY accepts the tuple with the author's name (group-by key) and then its inner focus is applied (AGGREGATE) so that all the objects whose author field have the same value will be put in the same sequence.

At this point, ASSIGN *treat* appears; this ensures that the input expression has the designated type. So, our rule searches for the type returned from the sequence created from the AGGREGATE operator. If it is of type item which is the treat type argument, the whole *treat* expression can be safely removed. As a result, the whole ASSIGN can now be removed since it is a redundant operator (Figure 10).

After the former removal, GROUP-BY is followed by an ASSIGN *count* which calculates the number of book titles (*value* expression) generated by AGGREGATE *sequence*. According to the JSONiq extension to XQuery, *value* can be applied only on a JSON object or array. However, in our case the query plan applies

```
DISTRIBUTE-RESULT( $$20 )
UNNEST($$20:iterate($$19))
ASSIGN($$19:count(value($$16, "title")))
GROUP-BY($$17:0->$$14){
   AGGREGATE($$16:create_sequence($$13))
}
ASSIGN($$14:data(value($$13, "author")))
DATASCAN( collection("/books"), $$13, ("bookstore")("book")( ))
EMPTY-TUPLE-SOURCE
```

Figure 10: Query Plan without *treat* Expression

value to a sequence, since GROUP-BY aggregates all the records having the same group-by key in a sequence. Thus, ("title") is applied on a sequence. To overcome this conflict, we convert the ASSIGN to a SUBPLAN operator (Figure 11). SUBPLAN's inner focus introduces an UNNEST *iterate* which iterates over AGGREGATE *sequence* and produces a single tuple for each item in the sequence. The inner focus of SUBPLAN finishes with an AGGREGATE along with a count function which incrementally calculates the number of tuples that UNNEST feeds it with.

```
DISTRIBUTE-RESULT( $$20 )
UNNEST($$20:iterate($$19))
SUBPLAN{
   AGGREGATE($$19:count(value($$18, "title")))
   UNNEST($$18:iterate($$16))
}
GROUP-BY($$17:0->$$14){
   AGGREGATE($$16:create_sequence($$13))
}
ASSIGN($$14:data(value($$13, "author")))
DATASCAN( collection("/books"), $$13, ("bookstore")("book")( ))
EMPTY-TUPLE-SOURCE
```

Figure 11: Convert scalar to aggregation expression

This conversion not only helps resolving the aforementioned conflict but it also converts the scalar count function to an aggregate one. This means that instead of calculating count on a whole sequence, we can incrementally calculate it as each item of the sequence is fetched.

In Figure 11, GROUP-BY still creates a sequence in its inner focus, which is the input to SUBPLAN UNNEST. Instead, we can push the AGGREGATE operator of the SUBPLAN down to the GROUP-BY operator by replacing it (Figure 12). That way, we exploit the benefits of the aforementioned conversion and have the count function computed at the same time that each group is formed (without creating any sequences). Thus, the new plan is not only smaller (more efficient) but also keeps the pipeline granularity introduced in both of the previous rule sets.

At this point, it is interesting to look at the second format of the query in Listing 5. The for loop inside the count function conveniently forms a SUBPLAN operator right above the GROUP-BY in the original logical plan. This is exactly the query plan described in Figure 11, which means that in this case we can immediately push the AGGREGATE down to GROUP-BY, without any further transformations.

```
DISTRIBUTE-RESULT( $$20 )
UNNEST($$20:iterate($$16))
GROUP-BY($$17:0->$$14){
  AGGREGATE($$16:count(value($$13,"title")))
}
ASSIGN($$14:data(value($$13, "author")))
DATASCAN( collection("/books"), $$13, ("bookstore")("book")( ))
EMPTY-TUPLE-SOURCE
```

Figure 12: Updated Query Plan with Count Function

Now that the count function is converted into an aggregate function, there is another rule introduced in [17] that can be activated to further improve the overall query performance. This rule supports Algebricks' two-step aggregation scheme, which means that each partition can calculate locally the count function on its data. Then, a central node can compute the final result using the data gathered from each partition. Thus, partitioned computation is enabled, which improves parallelism effectiveness.

The final query plan, produced after the application of all the former rules, calculates the count function at the same time that each grouping sequence is built as opposed to first building it and then processing the aggregation function.

5 EXPERIMENTAL EVALUATION

We have tested our rewrite rules by integrating them into the latest version of Apache VXQuery [4]. Each node has two dual-core AMD Opteron(tm) processors, 8GB of memory, and two 1TB hard drives. For the multi-node experiments we built a cluster of up to nine nodes of identical configuration. We used a real dataset with sensor data and a variety of queries, described below in Sections 5.1 and 5.2 respectively. We repeated each query five times. The reported query time is an average of the five runs. We first consider single-node experiments and include measurements for execution time (before and after applying our rewrite rules) and for speed-up. For the multi-node experiments we measure response time and scale-up over different numbers of nodes. We, also, include comparisons with Spark SQL and MongoDB that show that the overhead of their loading phase is non-negligible. Finally, we compare with AsterixDB which has the same infrastructure as our system; in particular we compare with two approaches, one that sees the input as an external dataset (depicted in the figures as AsterixDB) and one that first loads the file internally (depicted as AsterixDB(load)).

5.1 Dataset

The data used in our experiments are publicly available from the National Oceanic and Atmospheric Administration (NOAA) [21]. The Daily Global Historical Climatology Network (GHCN-Daily) dataset was originally in dly format and was converted to its equivalent NOAA web service JSON representation.

Listing 6 shows an example JSON sensor file structure. All records are wrapped into a JSON item, specifically array, called `"root"`. Each member of the `"root"` array is an object item which contains the object `"metadata"` and the array `"results"`. The latter stores various measurements organized in individual objects. A specific measurement includes the date, the data type (a description of the measurement, with typical values being TMIN, TMAX, WIND etc.), the station id where the measurement was taken, and the actual measurement value. The `"count"` object included into the `"metadata"` denotes the number of measurements objects in the accompanying `"results"` array. Typically a `"results"` array contains measurements from a given station for one month (i.e. typically 30 measurements). A sensor file contains only one `"root"` array which may consist of several `"results"` (measurements from the same or different stations) accompanied by their `"metadata"`.

Sensor file sizes vary from 10MB to 2GB. Each node holds a collection of sensor files; the size of the collection varies from 100MB to 803GB. The collection size is varied throughout the experiments and is cited explicitly for each experiment. In our experiments, we assume that the data has already been partitioned among the existing nodes.

```
{
  "root": [
    {
      "metadata": {
        "count":31,
      },
      "results": [
        {
          "date":"20132512T00:00",
          "dataType":"TMIN",
          "station":"GSW123006",
          "value":4
        },
        ...
      ]
    },
    {
      "metadata": {
        "count":29,
      },
      "results": [
        {
          "date":"20142512T00:00",
          "dataType":"WIND",
          "station":"GSW957859",
          "value":30
        },
        ...
      ]
    },
    ...
  ]
}
```

Listing 6: Example JSON Sensor File Structure

5.2 Query Types

We evaluated our newly implemented rewrite rules by evaluating different types of queries including selection (Q0, Q0b), aggregation (Q1, Q1b) and join-aggregation queries (Q2). The query description follows.

```
for $r in collection("/sensors")("root")()("
    results")()
let $datetime := dateTime(data($r("date")))
where year-from-dateTime($datetime) ge 2003
  and month-from-dateTime($datetime) eq 12
  and day-from-dateTime($datetime) eq 25
return $r
```

Listing 7: Selection Query (Q0)

```
for $r in collection("/sensors")("root")()("
    results")()("date")
let $datetime := dateTime(data($r))
where year-from-dateTime($datetime) ge 2003
  and month-from-dateTime($datetime) eq 12
  and day-from-dateTime($datetime) eq 25
return $r
```

Listing 8: Selection Query (Q0b)

Q0: In this query (Listing 8), the user asks for historical data from all the weather stations by selecting all December 25 weather measurement readings from 2003 on.

Q0b is a variation of Q0 where the input path (1st line in Listing 8), is extended by a *value* expression ("date").

```
for $r in collection ("/sensors") ("root")()
("results")()
where $r("dataType")  eq "TMIN"
group by $date:= $r("date")
return count($r("station"))
```

Listing 9: Aggregation Query (Q1)

```
for $r in collection ("/sensors") ("root")()
("results")()
where $r("dataType")  eq "TMIN"
group by $date:= $r("date")
return count(for $i in $r return $i("station"))
```

Listing 10: Aggregation Query (Q1b)

Q1: This query (Listing 10), finds the number of stations that report the lowest temperature for each date. The grouping key is the date field of each object.

Q1b is a variation of Q1 that has a different returned result structure.

Q2: This self-join query (Listing 11) joins two large collections, one that maintains the daily minimum temperature per station and one that contains the daily maximum temperature per station. The join is on the station id and date and finds the daily temperature difference per station and returns the average difference over all stations.

```
avg(
  for $r_min in collection("/sensors")("root")()(
      "results")()
  for $r_max in collection("/sensors")("root")()(
      "results")()
  where $r_min("station") eq $r_max("station")
    and $r_min("date") eq $r_max("date")
    and $r_min("dataType") eq "TMIN"
    and $r_max("dataType") eq "TMAX"
  return $r_max("value") - $r_min("value")
) div 10
```

Listing 11: Join-Aggregation Query (Q2)

5.3 Single Node Experiments

To explore the effects of the various rewrite rules we first consider a single node, one core environment (i.e. one partition) and progressively enable the different sets of rules. We start by considering just the *path expression rules*. Figure 13 shows the execution time for all five queries with and without these rules. For this experiment, we used a 400MB collection of sensor files (each of size 10MB). Note that for these experiments we used a relatively small collection size since without the JSONiq rules Hyracks would need to process the whole (possibly large) file thus slowing its performance. The application of the Path Expression Rules results to a clear improvement of the execution time for all queries. These rules decrease the buffer size between operators since large sequences of objects are avoided. Instead, each object is passed on to the next operator separately, resulting in faster query execution.

Using the same dataset and having enabled the Path Expression rules, we now examine the effect of adding the *Pipelining* rules (Figure 14). We observe that in all cases the pipelining rules provide a drastic improvement (note the logarithmic scale!),

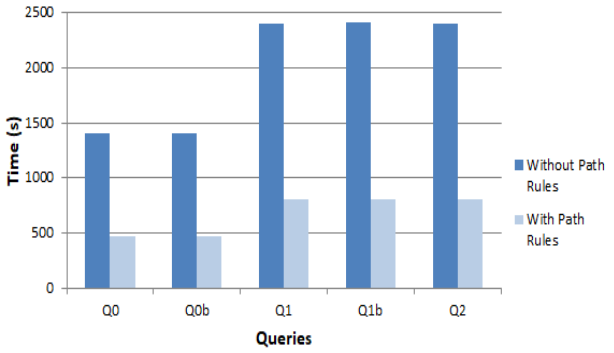

Figure 13: Execution Time before and after Path Expression Rules

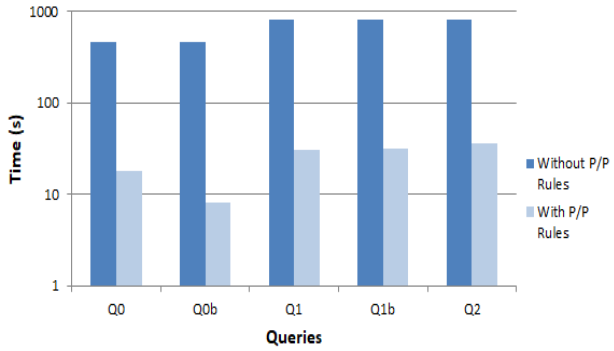

Figure 14: Execution Time (logscale) before and after the Pipelining Rules

speeding queries up by about two orders of magnitude. Applying these rules decreases the initial buffering requirement since we don't store the whole JSON document anymore, but just the matching objects. It is worth noting that the best performance is achieved by Q0b. Q0b stores in memory only the parts of the objects whose key field is "date". By focusing only on the "date", this gives the DATASCAN operator the opportunity to iterate over much smaller tuples than Q0. Clearly, the smaller the argument given to DATASCAN, the better for exploiting pipelining.

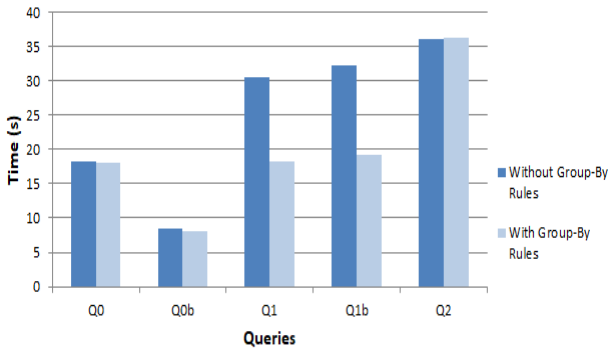

Figure 15: Execution Time before and after Group-by Rules

Having enabled both the path expression and the pipelining rules, we proceed considering the effect of adding the *Group-by* rules. The results are depicted in Figure 15. Clearly Q0, Q0b and Q2 are not affected since the Group-by rules do not apply. The Group-by rules improve the performance of both queries Q1 and Q1b. The improvement for both queries comes from the same rule, the rule that pushes the COUNT function inside the group-by. We note that the second Group-by rule, the one performing conversion, applies only to Q1; we do not enjoy an improvement from the conversion rule here because Q1b is already written in an optimized way. It is worth mentioning that the efficiency of the group-by rules depends on the cardinality of the groups created by the query. Clearly, the larger the groups, the better the observed improvement.

To study the effectiveness of all of the rewrite rules as the partition size increases, we ran an experiment where we varied the collection size from 100MB to 400MB. Figure 16 (again with a log scale) depicts the execution time for query Q1 both before and after applying all three sets of rewrite rules. We chose Q1 here because it is indeed affected by all of the rules. We can see that the system scales proportionally with the dataset size and that the application of the rewrite rules results in a huge query execution time improvement.

Figure 16: Execution Time (logscale) for Q1 before and after Rewrite Rules for different Data Sizes

From the above experiments, we can conclude that the Pipelining rules provide the most significant impact. It is also worth noting that the execution of the rewrite rules during query compilation adds a minimal overhead (just a few msec) to the overall query execution cost.

Single node Speed-up: To test the system's single node speed up, we used a dataset larger than our node's available memory space (8GB). Specifically, we used 88GB of JSON data, which we progressively divided from one up to eight partitions (our CPU has 4 cores and supports up to 8 hyperthreads). Each partition corresponds to a thread. The results are shown in Figure 17.

For up to 4 partitions and for almost all query categories, we achieve good speed-up since our observed execution time is reduced by a factor close to the number of threads that are being used. On the other hand, when using 8 hyperthreaded partitions we observe no performance improvements and in some cases a slightly worst execution time. This is because our processing is CPU bound (due to the JSON parsing), hence the two hyperthreads are effectively run in sequence (on a single core). In summary, we see the best results when we match the number of partitions to the number of cores.

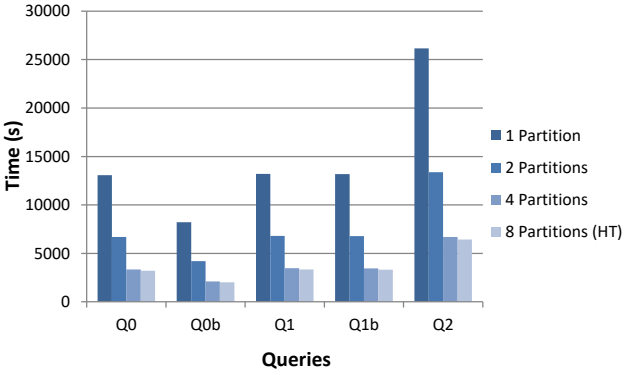

Figure 17: Single Node Speed-up

Comparison with MongoDB and AsterixDB: When comparing our performance against MongoDB and AsterixDB we observed that the performance of these systems is affected by the structure of the input JSON file. For example, a file with the structure of Listing 6 will be perceived by MongoDB and AsterixDB as a single, large document. Since MongoDB and AsterixDB are optimized to work with smaller documents (MongoDB has in addition a document size limitation of 16MB), to make a fair comparison we examined their performance while varying the number of documents per file.

We first unwrapped all the JSON items inside `"root"` (Listing 6). This results to many individual documents per file, each document containing a `"metadata"` JSON object and its corresponding `"results"` JSON array (typically with 30 measurements). We further manipulated the number of documents per file by varying the number of member objects (measurements) inside the `"results"` array from 30 (one month of measurements per document) to 1 (one day/measurement per document).

Figure 18.a depicts the query time performance for query Q0b for VXQuery, MongoDB, AsterixDB and AsterixDB(load); the space used by each approach appears in Figure 18.b. The total size of the dataset is 88GB and we vary the measurements per JSON array.

In contrast to MongoDB and the AsterixDB approaches, we note that the performance of VXQuery is independent of the number of documents per file. MongoDB has better query time for larger documents (30 measurements per array). Since MongoDB performs compression per document, larger documents allow for better compression and thus query time performance. This can also be seen in figure 18.b, where the space requirements increase as the document becomes smaller (and thus less compression is possible). The space for both AsterixDB approaches and VXQuery is independent from the document size (which is to be expected as currently these systems do not use compression).

AsterixDB shows a different query performance behavior than MongoDB. Its best performance is achieved for smaller document sizes (one measurement per document). Since it shares the same infrastructure as VXQuery, the difference in its performance relative to VXQuery is due to the lack of the JSONiq Pipeline Rules. Without them, the system waits to first gather all the measurements in the array before it moves them to the next stage of processing. This holds for both AsterixDB and AsterixDB(load). Among the two approaches, the AsterixDB(load) approach has better query performance since it is optimized to work better for data that is already in its own data model. Interestingly, for the

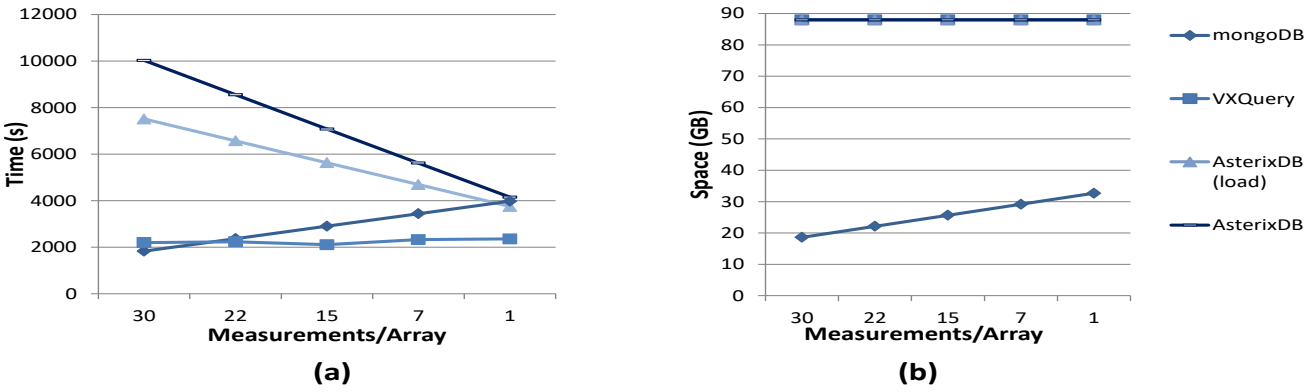

Figure 18: (a) Execution Time and (b) Space Consumption for Different Measurement Sizes per Array

Measurements/Array	30	22	15	7	1
MongoDB	9000	11703	14443	17146	19876
AsterixDB (load)	24659	23987	24205	24547	24612

Table 1: Loading Time in sec for AsterixDB (load) and MongoDB for Different Record Sizes

smaller document sizes (where compression is limited), AsterixDB and MongoDB have similar query performance. For larger document sizes their query performance difference seems to be directly related to their data sizes. For example, with 30 measurements per document, MongoDB uses about 4.5 times less space due to compression and has about 4 times less query time than AsterixDB(load).

Table 1 depicts the loading times for MongoDB and AsterixDB(load) for different measurements/array (in contrast there is no loading time for AsterixDB and VXQuery). The different loading times can also be explained by the space consumed by MongoDB and AsterixDB(load) (Figure 18.b). Specifically, MongoDB needs less loading time due to its use of compression; as expected, its loading time increases as the number of measurements per array is decreased due to less compression.

Comparison with SparkSQL: We next compare with a well-known NoSQL system, SparkSQL. In this experiment we ran Q1 both on VXQuery with the JSONiq rewrite rules and on Spark SQL and we compared their execution times. We used a single node and one core as the setup for both systems. We varied the dataset sizes starting from 400MB up to 1GB. We could not run experiments with larger input files because Spark required more than the available memory space to load such larger datasets.

Table 2 shows the SparkSQL loading times for the datasets used in this experiment. Figure 19 shows the query times for query Q1 for the different data sizes. Note that the VXQuery bar shows the total execution time for each file (which includes the loading and query processing) while the SparkSQL bar corresponds to the query processing time only. VXQuery's total execution time is slower than Spark's query time for small files. The two systems show similar performance for 800MB files. However, as the collection size increases, Spark's behavior starts to deteriorate. For the 1GB dataset our system's overall performance is clearly faster. If one counts also for the file loading time of SparkSQL (the overhead added by loading and converting JSON data to a SQL-like format), the VXQuery performance is faster. While the overhead of the loading phase is not a significant burden for

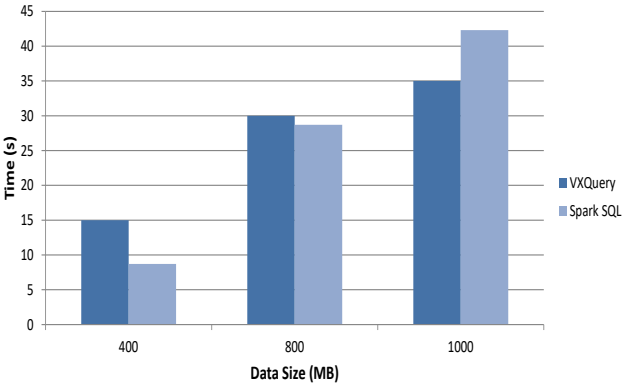

Figure 19: Spark SQL vs VXQuery Execution Time for Query Q1 Using Different Data Sizes (MB)

Data Size (MB)	Loading Time (s)
400	6.3
800	15
1000	40

Table 2: Loading Time for Spark SQL

Data Size (MB)	Spark Memory (MB)	VXQuery Memory (MB)
400	5650	1690
800	6230	1750
1000	7953	1760

Table 3: Data size to system memory in MBs

SparkSQL when considering small inputs (400MB) it becomes an important limiting factor even for medium size files (800MB).

We also examined the memory allocated by both systems (Table 3). The results show that VXQuery stores only data relevant to the query in memory, as opposed to SparkSQL, which stores everything. For file sizes above 2GB, the memory needs of SparkSQL exceeded the node's available 16GB, so it was unable to load the input data so as to query it.

5.4 Cluster Experiments

The goal of this section is to examine the cluster speed-up and scale-up achieved by VXQuery due to our JSONiq rewrite rules

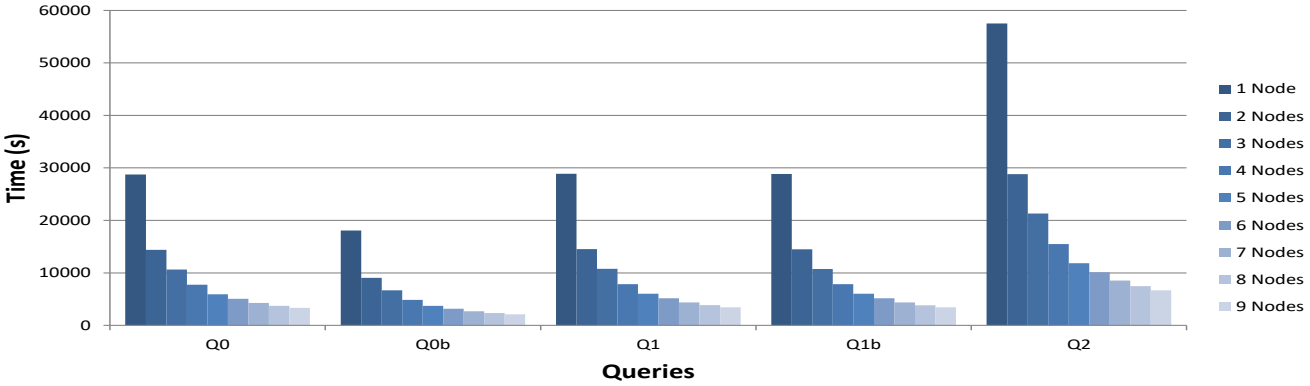

Figure 20: VXQuery Cluster Speed-up for all Queries (803GB Dataset)

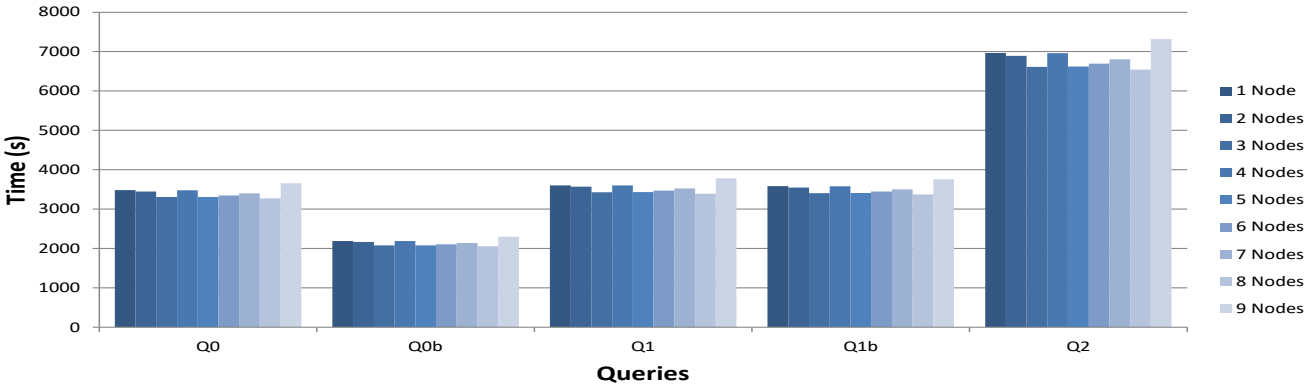

Figure 21: VXQuery Cluster Scale Up for all Queries (88GB per Node)

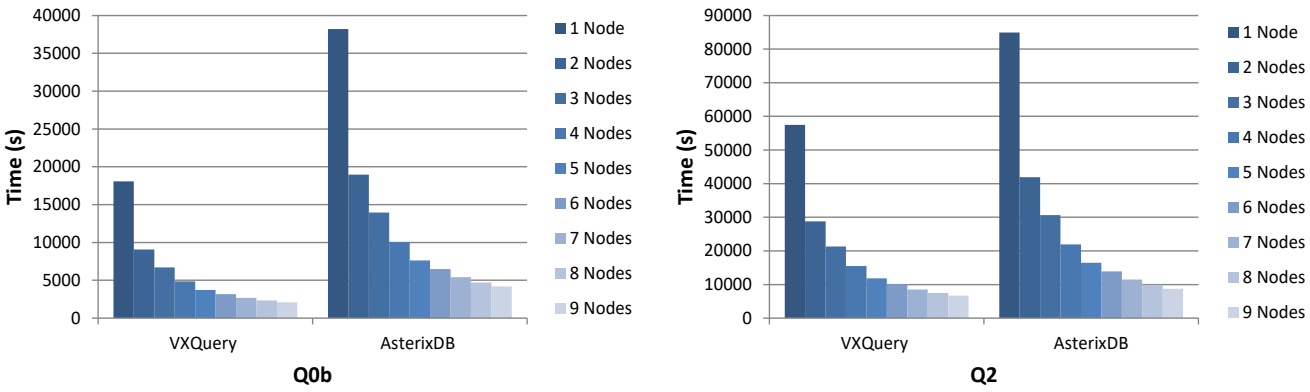

Figure 22: VXQuery vs AsterixDB: Cluster Speed-up for Q0b and Q2 (803GB Dataset)

and compare it with AsterixDB and MongoDB. For all the following experiments we used 4 partitions per node which achieves the best execution time as shown in the previous section.

To measure the *cluster speed-up* we started with a single node and extended our cluster by one node until it reached to 9 nodes. We used 803GB of JSON weather data which were evenly partitioned among the nodes used in each experiment. This dataset exceeds the available cluster memory. The results for this evaluation are shown in Figure 20. We note that in all the cases cluster speed-up is proportional to the number of nodes being used, without depending on the type of the query. We observe that the last query (Q2) takes the most time to execute. This is expected because Q2 is a self join query, which means that it has to process twice the amount of data. On the other hand, for VXQuery, Q0b has the best response time due to its small input search path as described in previous sections.

To show the *scalability* achieved by VXQuery, we started with a dataset of size 88GB which exceeds one node's available memory (8GB). With each additional node added we add four partitions totaling 88GB (hence when using 9 nodes the whole collection is 803GB). The results appear in Figure 21. As it can be seen our system achieves very good scale-up performance; the query execution time remains roughly the same, which means that the additional data is processed in roughly the same amount of time.

Comparison with AsterixDB: In the cluster experiments, we compare against AsterixDB (i.e. without loading; each dataset

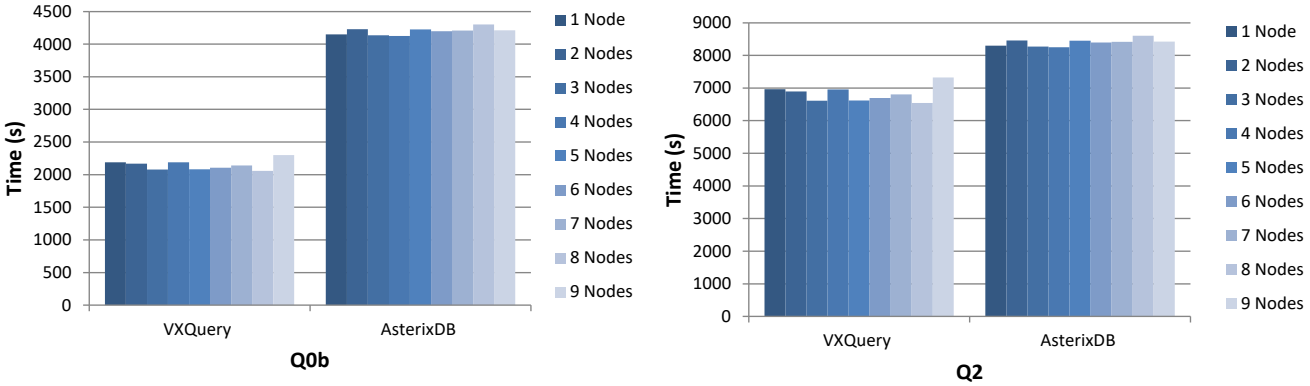

Figure 23: VXQuery vs AsterixDB: Cluster Scale-up for Q0b and Q2 (88GB per Node)

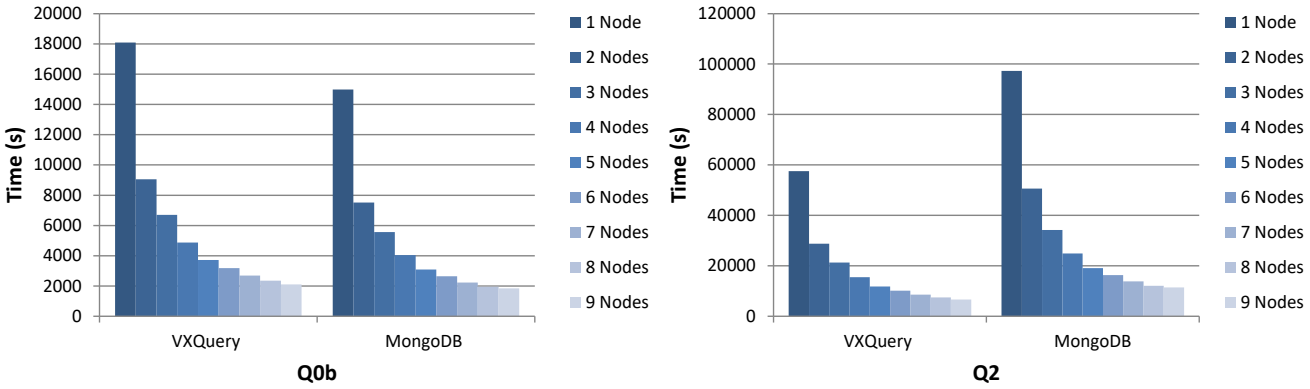

Figure 24: VXQuery vs MongoDB: Cluster Speed-up for Q0b and Q2 (803GB Dataset)

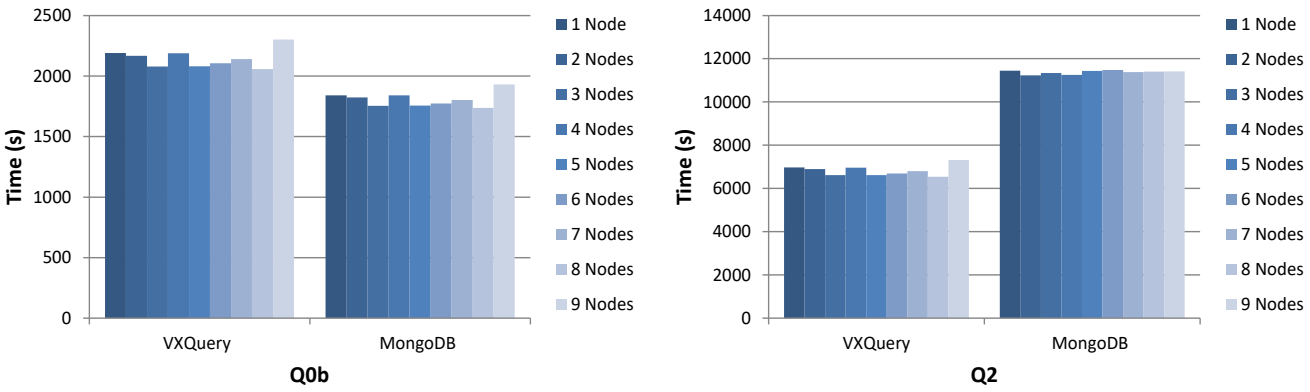

Figure 25: VXQuery vs MongoDB: Cluster Scale-up for Q0b and Q2 (88GB per Node)

is provided as an external data source). As seen in the single-node experiments, the best performance for AsterixDB is achieved when `"results"` consists of only one measurement; thus we use this structure for the following evaluation.

Following similar reasoning with the single-node comparison, we observe that VXQuery performs better both for speed up (Figure 22) and scale up (Figure 23), using queries Q0b and Q2 as representative examples.

Comparison with MongoDB: Similarly, we compare against the MongoDB configuration that achieved the best performance in the single-node experiments (i.e., `"results"` contains all monthly measurements). Overall, MongoDB has faster query time for selection queries than VXQuery (Figure 24 shows speedup for query Q0b; the Q0 query performed similarly). Since MongoDB performs a compression during the loading phase of the dataset, the dataset stored in the database is much smaller giving an advantage to selection queries. However, VXQuery's execution time for query Q0b is still comparable since its small input search path gives the opportunity for the Pipeline rules to be exploited.

In contrast, VXQuery has a faster execution time than MongoDB on join queries (like Q2). For this self-join, MongoDB tries to put all the documents that share the same station and date in the same document; thus creating huge documents which exceed the 16MB document size limit causing it to fail. To overcome this problem, we perform an additional step before the actual join. We unwind the `"results"` array and we project only the

Data Size (GB)	Loading Time (sec)
88	9000
803	81000

Table 4: Loading Time for MongoDB

necessary fields. After that, we perform the actual join on the corresponding attributes (i.e. station, date of measurement).

On the other hand, in VXQuery there is no document size limitation, making VXQuery more efficient in handling large JSON items. Table 4 shows the MongoDB loading times per node. This adds a huge overhead to the performance of the overall system and it can be prohibitively large for real-time applications where the dataset may not been known in advance.

Comparison with SparkSQL: As mentioned in the single node experiments SparkSQL could not run experiments with larger input files because of the required memory space to load such larger datasets. Hence we omit multi-node experiments with SparkSQL, since the dataset size will be very small to indicate and difference in the execution time.

6 CONCLUSIONS AND FUTURE WORK

In this work we introduced two categories of rewrite rules (path expression and pipelining rules) based on the JSONiq extension to the XQuery specification. We also introduced a third rule category, group-by rules, that apply to both XML and JSON data. The rules enable new opportunities for parallelism by leveraging pipelining; they also reduce the amount of memory required as data is parsed from disk and passed up the pipeline. We integrated these rules into an existing system, the Apache VXQuery query engine. The resulting query engine is the first that can process queries in an efficient and scalable manner on both XML and JSON data. Through experimentation, we showed that these rules improve performance for various selection, join and aggregation queries. In particular, the pipelining rules improved performance by several orders of magnitude. The system was also shown both to speed-up and scale-up effectively. Moreover, when compared with other systems that can handle JSON data, VXQuery shows significant advantages. In particular, our system can directly process JSON data efficiently without the need to first load it and transform it to an internal data model.

We are currently working on supporting indexing over both types of data (XML and JSON). Indexing presents a significant challenge, as there is no simple way to decide the level at which an object could be indexed; indexing will further improve the system's performance since the searched data volume will be significantly reduced. All of the code for our JSONiq extension is available through the Apache VXQuery site [4] and it will be included in the next Apache VXQuery release. Furthermore, we plan to add the proposed path and pipelining rules directly to AsterixDB given that it shares the same infrastructure (Algebricks and Hyracks) with VXQuery.

7 ACKNOWLEDGMENTS

This research was partially supported by NSF grants CNS-1305253, CNS-1305430, III-1447826 and III-1447720.

REFERENCES

[1] 2012. Jackson project. https://github.com/FasterXML/jackson. (2012).
[2] 2013. Zorba. http://www.zorba.io/home. (2013).
[3] 2014. Apache Parquet. https://parquet.apache.org/. (2014).
[4] 2016. Apache VXQuery. http://vxquery.apache.org. (2016).
[5] 2017. Apache AsterixDB. https://asterixdb.apache.org/. (2017).
[6] 2017. Apache Drill. https://drill.apache.org/. (2017).
[7] 2017. Apache Spark. https://spark.apache.org/. (2017).
[8] 2017. JSONiq Extension to XQuery. http://www.jsoniq.org/docs/JSONiqExtensionToXQuery/html-single/index.html. (2017).
[9] 2017. JSONiq Language. http://www.jsoniq.org/docs/JSONiq/html-single/index.html. (2017).
[10] 2017. MongoDB. https://www.mongodb.com/. (2017).
[11] 2017. Postgres-XL. https://www.postgres-xl.org/. (2017).
[12] 2017. PostgreSQL. https://www.postgresql.org/. (2017).
[13] Michael Armbrust and others. 2015. Spark SQL: Relational data processing in Spark. In *Proceedings of ACM SIGMOD Conference*. 1383–1394.
[14] Kevin S Beyer, Vuk Ercegovac, Rainer Gemulla, Andrey Balmin, Mohamed Eltabakh, Carl-Christian Kanne, Fatma Ozcan, and Eugene J Shekita. 2011. Jaql: A scripting language for large scale semistructured data analysis. In *Proceedings of VLDB Conference*.
[15] Vinayak Borkar, Yingyi Bu, E Preston Carman Jr, Nicola Onose, Till Westmann, Pouria Pirzadeh, Michael J Carey, and Vassilis J Tsotras. 2015. Algebricks: a data model-agnostic compiler backend for Big Data languages. In *Proceedings of 6th ACM Symposium on Cloud Computing*. 422–433.
[16] Vinayak Borkar, Michael Carey, Raman Grover, Nicola Onose, and Rares Vernica. 2011. Hyracks: A flexible and extensible foundation for data-intensive computing. In *27th International Conference on Data Engineering*. 1151–1162.
[17] E Preston Carman, Till Westmann, Vinayak R Borkar, Michael J Carey, and Vassilis J Tsotras. 2015. A scalable parallel XQuery processor. In *IEEE International Conference on Big Data*. 164–173.
[18] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, Danyel Fisher, John C Platt, James F Terwilliger, and John Wernsing. 2014. Trill: A high-performance incremental query processor for diverse analytics. *Proceedings of the VLDB Endowment* 8, 4 (2014), 401–412.
[19] Craig Chasseur, Yinan Li, and Jignesh M Patel. 2013. Enabling JSON Document Stores in Relational Systems.. In *WebDB*, Vol. 13. 14–15.
[20] Gabriel Filios, Sotiris Nikoletseas, Christina Pavlopoulou, Maria Rapti, and Sébastien Ziegler. 2015. Hierarchical algorithm for daily activity recognition via smartphone sensors. In *2nd IEEE World Forum on Internet of Things (WF-IoT)*. 381–386.
[21] Felix N Kogan. 1995. Droughts of the late 1980s in the United States as derived from NOAA polar-orbiting satellite data. *Bulletin of the American Meteorological Society* 76, 5 (1995), 655–668.
[22] Shamanth Kumar, Fred Morstatter, and Huan Liu. 2013. *Twitter Data Analytics*. Springer Publishing Company.
[23] Yinan Li, Nikos R Katsipoulakis, Badrish Chandramouli, Jonathan Goldstein, and Donald Kossmann. 2017. Mison: a fast JSON parser for data analytics. In *Proceedings of the VLDB Endowment*, Vol. 10. 1118–1129.
[24] Jimmy Lin and Dmitriy Ryaboy. 2013. Scaling big data mining infrastructure: the twitter experience. *ACM SIGKDD Explorations Newsletter* 14, 2 (2013), 6–19.
[25] Zhen Hua Liu, Beda Hammerschmidt, and Doug McMahon. 2014. JSON data management: supporting schema-less development in RDBMS. In *Proceedings of ACM SIGMOD Conference*. 1247–1258.
[26] Jason McHugh and Jennifer Widom. 1999. Query Optimization for XML. *Proceedings of the 25th VLDB Conference* (1999).
[27] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2010. Dremel: interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 330–339.
[28] Nurzhan Nurseitov, Michael Paulson, Randall Reynolds, and Clemente Izurieta. 2009. Comparison of JSON and XML data interchange formats: a case study. *22nd Intl. Conference on Computer Applications in Industry and Engineering (CAINE)* (2009), 157–162.
[29] Daniel Tahara, Thaddeus Diamond, and Daniel J Abadi. 2014. Sinew: a SQL system for multi-structured data. In *Proceedings of ACM SIGMOD Conference*. ACM, 815–826.