Run-Time Performance Optimization of a BigData Query Language

Yanbin Liu, Parijat Dube,
Scott C. Gray
IBM Watson Research Center
Yorktown Heights, USA.
ygliu,pdube,sgray@us.ibm.com

ABSTRACT

JAQL is a query language for large-scale data that connects BigData analytics and MapReduce framework together. Also an IBM product, JAQL's performance is critical for IBM InfoSphere BigInsights, a BigData analytics platform. In this paper, we report our work on improving JAQL performance from multiple perspectives. We explore the parallelism of JAQL, profile JAQL for performance analysis, identify I/O as the dominant performance bottleneck, and improve JAQL performance with an emphasis on reducing I/O data size and increasing (de)serialization efficiency. With TPCH benchmark on a simple Hadoop cluster, we report up to 2x performance improvements in JAQL with our optimization fixes.

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: Performance of Systems—*Performance attributes*

Keywords

BigInsights, JAQL, MapReduce, Multi-thread, I/O optimization, Java performance

1. INTRODUCTION

The proliferation of cheaper technologies for collecting and transmitting data of different modalities and formats from different sources such as sensors, cameras, Internet feeds, social networks, mobile phones etc. has rendered the business world with vast amount of data. This plethora of data, if properly processed, can provide significant new insights for tracking markets, customers and business performance. Information management of such vast amount of *Big Data* is critical and technologies for efficient and cost-effective filtering, storing and accessing of data are increasingly sought after. Big Data [2] is typically characterized by having 4 Vs: high Volume, high Velocity, high Variety and Veracity which is a measure of uncertainty in data. Processing of Big Data within reasonable time requires programming models supporting large-scale, parallel and efficient execution of queries on scale-out infrastructure. The MapReduce programming paradigm, first introduced by Google [10] provides a framework for large-scale parallel data processing. Apache Hadoop [1] is an open-source implementation of MapReduce.

While highly scalable, MapReduce is notoriously difficult to use. The Java API is tedious and requires programming expertise. Query languages like Apache Hive [17], Apache Pig [11] and JAQL [9] provide high-level abstraction to express queries which are then compiled into low-level MapReduce jobs.

IBM InfoSphere BigInsights [4] is a platform for scalable processing of Big Data analytics applications in an enterprise. BigInsights builds on top of open-source Hadoop by adding several features for cost effective management, processing and analysis of enterprise Big Data.

JAQL [9] is an integral part of BigInsights where it provides both the run time and an integration point for various analytics including text analytics, statistical analysis, machine learning, and ad-hoc analysis. Figure 1 shows the BigInsights platform stack. BigInsights applications are executed either entirely in JAQL or JAQL instantiates Hadoop MapReduce jobs for scalable query execution. In BigInsights, JAQL also provides modules to connect to various data sources like local and distributed file systems, relational databases, NoSQL databases. Performance of BigInsights applications is integrally tied with the performance of JAQL and Hadoop run-time.

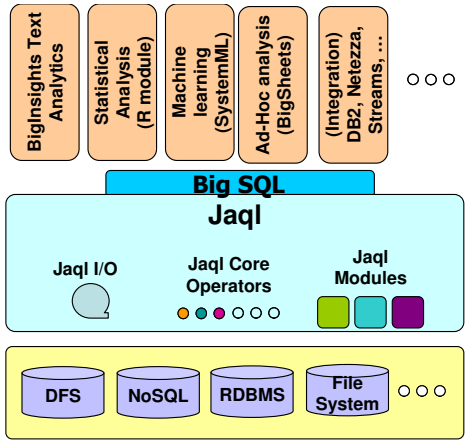

Figure 1: BigInsights Architecture

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICPE'14, March 22–26, 2014, Dublin, Ireland.
Copyright 2014 ACM 978-1-4503-2733-6/14/03 ...$15.00.
http://dx.doi.org/10.1145/2568088.2576800 .

BigInsights needs a fast run-time to improve performance and needs an ability to run SQL queries. Big SQL is a new and important component of BigInsights providing support for SQL queries execution. Big SQL server is multi-threaded supporting multiple sessions and thus can exploit performance scaling offered by multi-thread, multi-core systems. Since JAQL provides run-time environment for BigInsights applications, the focus on run-time efficiency and support for multi-thread execution in JAQL is important for scalable execution of Big SQL and other multi-threaded applications supported by BigInsights including IBM InfoSphere DataStage [5], Tableau [6], and IBM BigSheets [3]. Our work improves JAQL run-time performance in multi-thread, multi-core environments through: (i) Enabling run-time support for multi-threaded applications running on top of JAQL, and (ii) Speeding-up query execution time by identifying performance bottlenecks and fixing them.

Section 2 provides background information on query processing in BigInsights. Various run-time performance issues of JAQL are highlighted in Section 3. Our solution for enabling multi-thread execution in JAQL run-time is discussed in Section 4. Experiments showcasing improvements in Big SQL/JAQL run-time performance as a result of our optimizations are presented in Section 5. Related work is covered in Section 6 followed by conclusion and future work in Section 7.

2. BACKGROUND

JAQL [9] consists of three major components as shown in Figure 2: a scripting language, a compiler and a runtime component for MapReduce framework to transparently exploit parallelism. JAQL scripting language is designed for analyzing complex/nested semi-structured data. It provides a flexible data model based on JSON (Java Script Object Notation), an easy-to-extend modular design including first-class functions and a set of syntax for supporting control-flow statements and SQL queries. JAQL compiler is the central-piece that detects parallelism in a JAQL script and translates it into a set of MapReduce jobs. JAQL runtime component for MapReduce framework defines map, combine, reduce, (de)serialization and etc. functions that will be executed in the MapReduce framework.

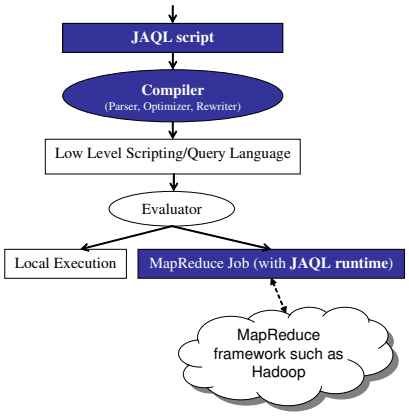

Figure 2: JAQL components

JAQL programming language has been utilized for a number of years for developing large scale analytics on Hadoop clusters. With the release of IBM BigInsights 2.1 (itself a Hadoop distribution) JAQL began to play a more important role by acting as the query parallelization engine and execution runtime for the Big SQL component and, as a result, it was important to dramatically increase the performance of JAQL, particularly when applied to the structured data world of SQL.

The Big SQL component of BigInsights contains a sophisticated SQL optimization and re-write engine that is responsible for taking modern, ANSI SQL, containing complex constructs, such as windowed aggregates, common table expressions, and subqueries, and optimizing the queries via a number of re-write steps. Examples of such optimization include:

- Decorrelation of subqueries
- Lifting of common, repeated queries or query fragments into common table expressions
- Identifying queries that cannot benefit from parallel execution
- Mapping of certain SQL constructs such as interval arithmetic into function calls

The output of the query optimization engine is a re-written SQL query that is then passed to JAQL for processing, at which point JAQL performs the following steps:

1. The SQL statement is immediately re-written into an equivalent JAQL expression. This expression is written in the same fashion that a user of JAQL would have written the query – that is, it contains no indication of how the query should be executed, but is simply an expression of the query that is to be performed.

2. The JAQL rewriter then performs a large number of query re-writes and optimizations, which includes basic activities such as simplifying expressions (e.g. 2 + 2 will be simplified to 4) to more complex re-writes, such as determining whether or not the query can be parallelized and, if so, how to approach the parallelization. Some of these optimizations are directed by the query rewriter via hints that were passed into JAQL via the original SQL statement.

3. The final result of all of these optimizations is a valid JAQL script. In many cases, this script will be decomposed into a series of explicit MapReduce jobs to achieve the end goal of executing the query.

While JAQL has been capable of the majority of the functionality described above for a number of years, in order to utilize it as the runtime for Big SQL, it was necessary to optimize a number of aspects of the language.

3. JAQL PERFORMANCE

3.1 Parallelism

JAQL performance, which includes the performance of both JAQL compiler and JAQL MapReduce runtime, can be influenced by multiple factors. As the multi-core, multi-processor become the main trend today, the first factor we look at is the parallelism. As we discussed above, by design,

JAQL already explores parallelism by transparently generating MapReduce jobs that can be executed in parallel. In this paper, we explore more parallelism by analyzing the multi-layer software stack of JAQL.

At the top-most level, a JAQL instance, which maintains the state information such as the declared variable and their values, function definitions and etc, can be started either by an interactive shell or an application that submits JAQL statements such as Big SQL. While an interactive shell is used by one user in general and corresponds to one JAQL session naturally, an application may submit JAQL statements on behalf of multiple users who have different objectives. For example, Big SQL supports multi-session and each session represents a separate expression stream from one user. In the past, each process could have at most one JAQL instance. With the changes we have made, we now allow one process to manage multiple JAQL instances (possibly each instance masquerading as a different user in MapReduce). While each JAQL session has at least one executing thread, we are able to improve the performance on a multi-core system with an increased number of threads.

After we have multiple JAQL sessions in one process, we continue with inter-session parallelism. A JAQL session consists of a sequence of JAQL statements, some of which will be rewritten as MapReduce jobs and submitted to a MapReduce framework such as Apache Hadoop [1]. We look at the input and output of the MapReduce jobs and check if there is data dependency among them. When there is no data dependency, we do not wait for the previous jobs to return, but continue with submitting jobs to the MapReduce framework. In this paper, we restrict ourselves to modify JAQL only and will not discuss the scheduling of MapReduce jobs. With ample resources and/or a capable MapReduce scheduler, we are able to improve the performance with submitting MapReduce jobs as early as possible.

3.2 I/O

JAQL is designed for large-scale data analysis and not surprisingly, I/O performance is critical for JAQL performance. This is verified by our experiment that for the benchmark we are running, I/O can be accounted for more than 35% of the total execution time.

3.2.1 I/O Data Size

I/O performance is defined by both I/O speed and I/O data size. When we are discussing the problem in the environment of large-scale data, I/O data size is the first thing that draws our attention. How to reduce the size of the data that needs to be read/write to either the file system or the network is critical for I/O performance. For a MapReduce job executed in Hadoop, its map tasks read input from a distributed file system such as HDFS and reduce tasks write output to HDFS. Inside map and reduce tasks, intermediate results are read from and written to the local file system. In this paper, we are more focused on how to reduce the size of the intermediate data.

3.2.2 I/O (De)Serialization Speed

I/O speed consists of I/O hardware speed, such as the speed of hard disks, and I/O software speed. In this paper, we are concerned about only I/O software speed and especially I/O software speed decided by the JAQL code.

JAQL runtime defines the serialization and de-serialization functions. Inside Hadoop, when data is written to and read from HDFS and local disk or when the data is merged and combined, data is serialized and de-serialized multiple times. We show that we can improve (de)serialization speed by 20%.

3.3 Methodology

Our methodology to analyze and improve JAQL performance is an iterative process as shown in Figure 3. At first, we profile different JAQL components. Then, depending on the profiling results, we identify and locate the performance bottlenecks. After that, we develop solutions to address the performance problems and implement them in the code. To verify the correctness of the solution and to locate new bottlenecks, we profile the new version of the implementation and repeat the process.

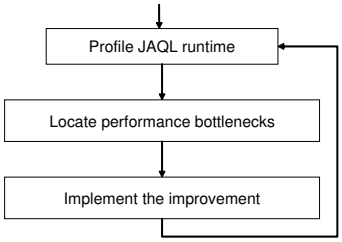

Figure 3: Methodology

4. THREAD-SAFE JAQL RUN-TIME

As we discussed above, JAQL should be able to support multi-session and thus multi-thread to improve its performance on multi-core, multi-processor systems. In general, this problem is a typical textbook problem. However, the problem gets tricky when JAQL sessions share the underlying Hadoop environment including file systems. We will describe it in more details later.

Each JAQL instance supports multiple sessions and each JAQL session has one main thread to parse, compile and evaluate JAQL statement one by one. We save the session resources in the thread local storage of the main thread and call it the session context.

Originally, we make all the shared data as session resources, which are shared among the threads of the same session. That is, we keep a copy of the shared data in the session context. For example, for system properties that describe the environment configuration, we maintain a copy in each session such that each session can configure its environment independently.

Then, we convert some session resources to globally shared resources in the following cases.

- The resource is expensive to initialize. For example, for Hadoop 1.1 we used in this paper, creating a default JobConf [1] involves reading multiple configuration files and is an expensive operation. We make the default configuration a global resource, create it once at the beginning. Each session then makes its own copy of the default JobConf and be able to modify its copy later and thus configure its Hadoop jobs independently.

[1]JobConf describes a map-reduce job and needs to be defined when a user submits a job to Hadoop.

- The resource is read-only after initialization. For example, some JAQL specific data structure such as exception handlers falls into this category.

- The resource is not accessed frequently. We make these resources globally shared and synchronize the access to them. The decision is a trade-off between speed and space. For example, we have a variable *tz* to represent the current time zone. Another variable *df* represents a date format; a user can call *df.format* to return a string with the pre-defined date format. The implementation of both variables are from third parties and can not be changed easily. One the one hand, we make *tz* a global resource to save space since it is not changed frequently. One the other hand, we make *df* a session resource to avoid synchronizing every call to *df.format*.

The tricky part of making JAQL thread-safe is because of its feature of using Hadoop. The map and reduce functions defined by JAQL read from and write to files. The Hadoop API's, while largely thread safe, do maintain certain global state information, such as the currently active file system, as well as the working and temporary directory within this file system. Without any modification, each JAQL session will use Hadoop APIs to set the active file system as well as working and temporary directories and assuming these global values stay unchanged, use relative paths to locate files later. Unfortunately, it is not the case when there is another JAQL session that may change these values. Thus, as part of the effort to make JAQL thread safe, it was necessary to save each session's settings of the file system and call the Hadoop API carefully to locate the files correctly.

We have implemented the thread-safe JAQL and tested it with a suite of unit tests, which consists of 48 scripts that test different functions provided by JAQL. We have verified the correctness of our implementation by running the suite of tests in three JAQL optimization settings and be able to finish the tests more than 10 times faster.

5. PERFORMANCE ANALYSIS AND OPTIMIZATION

5.1 Profiling JAQL

To improve JAQL performance, we first identify the latency oriented bottlenecks. We accomplish this work by profiling JAQL using JProfiler [16], Hadoop monitoring tools and self-developed monitoring tools.

In this paper, we shall focus on the performance of SQL queries that are issued by Big SQL, which is an important component of BigInsights. JAQL compiler, which is executed inside of the Big SQL process, is responsible of parsing and interpreting SQL query, optimizing the query plan, and forming and submitting the MapReduce jobs. After the job is submitted to Hadoop, JAQL runtime for MapReduce will be executed in the JVMs started by map and reduce tasks. Thus, we shall profile both the Big SQL process and the JVMs of Hadoop map and reduce tasks to analyze JAQL performance.

The first observation we make is that the latency is mostly due to MapReduce job. For a query that takes minutes to finish, the time spent inside Big SQL before the job submission is in milliseconds level. Besides, this time will not change much with a different input and output while the time of MapReduce job depends greatly on the size of input and output data. Next, we should focus on the JAQL runtime for MapReduce since this is where most time is spent and where we can achieve most improvement of latency.

One SQL query issued by Big SQL can generate multiple MapReduce jobs; one MapReduce job can include multiple map and reduce tasks where each task starts a new JVM by default Hadoop configuration. In order to avoid resource competition of multiple JVMs on the same host, especially the competition over I/O and do a clean profiling, we configure Hadoop such that at any time, there is only one JVM running. Specifically, we set

mapred.tasktracker.map.tasks.maximum=1 and
mapred.tasktracker.reduce.tasks.maximum=1

such that there is only one slot for map task and one slot for reduce task on the host. Then, we set

mapred.reduce.slowstart.completed.maps=1.0

such that Hadoop should start reduce task after all the map tasks are completed. Thus, at any time, we will have only one JVM, at first for map tasks, then for reduce tasks. However, this still give us many JVMs when the number of map and reduce tasks is big. We go one step further to set

mapred.job.reuse.jvm.num.tasks=-1

such that all map tasks will share one JVM instead of starting a new one for each map task and all reduce tasks will share one JVM.

Since we restrict ourselves to JAQL runtime for MapReduce, the configuration discussed above is not only simplifying our profiling work, but also removing the overhead details that are outside of our interest range.

5.2 TPCH Benchmark

We use TPCH benchmark [18] to measure our performance. TPCH is a benchmark consisted of a set of SQL queries, which involves operations such as join, union, filter, sort, aggregate functions and etc, to a database that can be populated to a specified size. For the experiment results in this paper, we populate the database with 1G data and we choose to profile 6 queries which involve different operations as shown in Table 1 and they all involve querying the biggest table, which has 6G rows, in the database.

5.3 Performance Analysis and Optimization

5.3.1 Big SQL

We profile Big SQL server and verify that JAQL performance is critical for Big SQL performance. As shown in Figure 4, more than 70% of the execution time is spent on JAQL runtime, among which 31% is spent on parsing the query, 19% on rewriting and optimizing the query plan, and 22% on evaluating the query plan. One interesting thing to notice is that 15% of the execution time comes from the international component for unicode (com.ibm.icu) that is used to initiate JsonDate.

5.3.2 JAQL Runtime for MapReduce

As we mentioned earlier in the paper, the execution time of MapReduce jobs dominants the total execution time and deserves more attention for performance consideration. We profile the JVMs for map and reduce tasks with the Hadoop

query	operations	# M/R jobs	results
q1	aggregate functions (sum, avg)	2	4 rec with 11 col
q3	join	4	10 rec with 4 col
q5	nested join	4	5 rec with 2 col
q7	join and union all	5	4 rec with 4 col
q21	join and right outer join	7	100 rec with 2 col

Table 1: TPCH queries

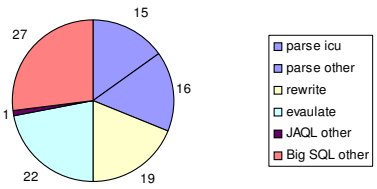

Figure 4: Big SQL Execution Time

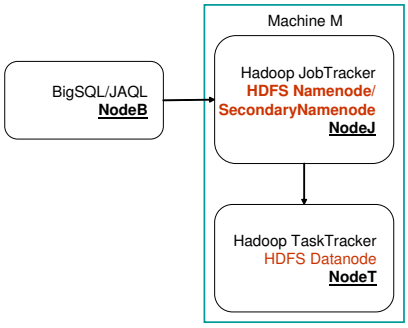

Figure 5: Hadoop Cluster

configuration discussed earlier and in the following, we report our observations and the improvements we made.

To focus on the profiling of JAQL runtime for MapReduce, we set up a simple Hadoop cluster with 2 nodes: one node J as Hadoop job tracker, HDFS Name node and secondary Name node; the other node T as Hadoop task tracker and HDFS Data node. The Big SQL server is on a separate node B. Node J and T are both VMWare virtual machines instantiated on one physical machine, which is a Dual-Core AMD processor with 2.2G CPU. Node B is a Intel Xeco CPU with 24 processors each of which has 1.6G CPU.

We issue the TPCH queries from Big SQL server and profile the map and reduce tasks executed. For the map tasks of the 1st job from query q1 of TPCH benchmark, we find that 50% is spent on JAQL defined map function and 46% is spent on combining and sorting the MapperŠs output among which 40% is spent on JAQL defined combine function. In total, JAQL runtime accounts to 90% of the execution time of the map task and is obviously crtical for the performance of MapReduce jobs.

We look at the profiling results deeper and observe that I/O serialization is dominant. For the same scenery as above, we observe that 16% is spent on

...binary.temp.TempBinaryFullSerializer.write

, which is responsible of writing the output of mappers, and 19% on

...binary.temp.TempBinaryFullSerializer.read

, which is responsible of the reading the input of mappers.

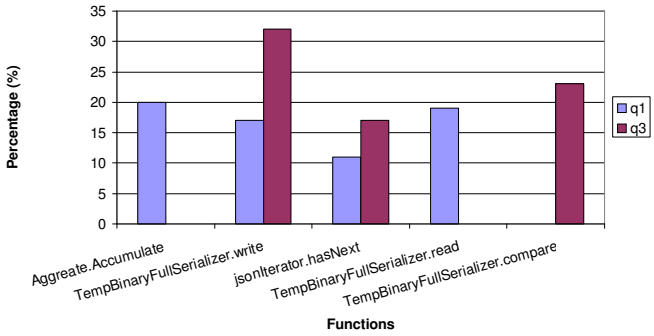

Figure 6: Comparison between Q1 and Q3

The dominance of I/O is not unique for q1, though the exact pattern may be different. As the comparison of q1 and the 1st job of q3 shown in Figure 6, we can see that for q1, the I/O accounts for 35% combining both write and read operations while for q3, the I/O accounts for 32% for write operation only. Besides that, q1 spends 20% on Aggregate.Accumulate which deals with the aggregate functions in the query such as avg, sum, count and etc.; q3 spends 23% on TempBinaryFullSerializer.compare which compares the values. The remaining function jsonIterator.hasNext is used to iterate the expressions and evaluate their values.

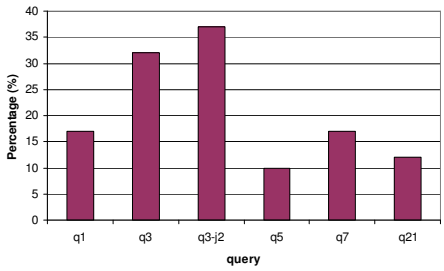

Figure 7: Percentage of Query's write operation

For other queries, we show the comparison result of the I/O write operation in Figure 7. From the figure, we can see that the percentage of

...binary.temp.TempBinaryFullSerializer.write

can be as high as 37% for the map task of the 2nd job of q3 and can be as low as 10% for q5.

5.4 I/O data size

Since I/O performance is dominant in many queries as shown in the above figures, we focus on I/O performance to improve JAQL overall performance. I/O performance is decided by both I/O data size and I/O speed.

We first summarize our effort to reduce I/O data size from different perspectives as follows. Since we cannot reduce the data size of initial input and final output, we check the query plan generated by JAQL and remove unnecessary intermediate data.

5.4.1 Number of Jobs

The goal is to reduce the number of jobs. As fewer jobs are generated, there is no need to pass the data to next job and thus can reduce I/O data size. Also, all the overhead of starting a new job will be eliminated. Both Big SQL and JAQL have optimized the query plan to reduce the number of jobs.

In this paper, we have JAQL dynamically adjust whether or not a particular job should be done locally or submitted to Mapreduce by examining the output of the previous job, which is also the input of the current job. That is, if the input of the current job is smaller than a pre-defined threshold, the job will be done locally to save the overhead of MapReduce framework and remote execution.

5.4.2 Number of Objects

Another goal of optimizing the query plan is to reduce the number of objects that needs to be passed between map and reduce tasks of the same job and the ones passed between jobs of the same query.

The improvement of JAQL on this perspective includes 1) push down the filtering conditions as deep as possible and thus filter out unnecessary objects as early as possible; 2) optimize the order of multiple filtering conditions with hints to do the most effective filtering first; 3) optimize the join order of multiple joins and join methods if they are not explicitly defined with hints to reduce the number of intermediate join results. The hints can be the size of the input tables, the number of rows that may match the join and filter condition, and other operations of the same query.

Another improvement has been reported in [19] which proposes adapting MapReduce to achieve a higher performance. In specific, "situation-aware" map tasks are implemented such that they can communicate through a distributed meta-data store and thus be able to have a global picture of the job. Besides the benefit of load-balancing, another benefit is to generate a smaller number of objects and thus reduce I/O size.

5.4.3 Number of Columns

It is obvious that we should eliminate the columns if they are not needed in the future. The trick is to examine the query plan that was chosen and decide, at any given step, if a particular column is no longer needed after that step, that it is eliminated and the values for the column are not transmitted to the next job in the chain. For example, if a column is only referenced during a single join, but no where else in the query, then after that join has been processed, the column is no longer needed for subsequent steps, such as additional joins, or sorting and grouping.

5.4.4 Number of Encoded Bytes

From both previous experience of other researchers and our experiments, we do not see observable improvement of enabling Hadoop data compression. Thus, we implement our own mechanism to encode the data and reduce the number of encoded bytes.

JAQL, being a language based upon JSON, tends to carry values around in JSON structures that are records [2]. So a given row may look like a record of:

{ c1: 10, c2: "bob", c3: "nelson" }

However, in a language like SQL, where you always have a strict schema, it is not necessary to carry around such structures, because the metadata exists externally to the data, the same value as above may be carried around as a simple array [3] of [10, "bob", "nelson"] saving significant space, where the column names don't need to be represented in every row, and processing time, where values are retrieved by position, not name.

The optimization that was performed is to make JAQL more aware of when the data is well formed with a schema, and that all operations performed on the data will result in data that is well formed. In these circumstances, JAQL can treat records as arrays. It can compile out the references to fields of records, such as $t1.c1 > 10$ and replace it with $t1[0] > 10$ and thus can avoid carrying around the associated metadata for the columns in each record.

5.5 I/O (de)serialization overhead

In this paper, for I/O speed, we only consider the I/O overhead imposed by JAQL code, especially the serialization and de-serialization. Besides the de-serialization and serialization of input and output, the intermediate results will need to go through multiple de-serialization and serialization when they are mapped, combined, shuffled, spilled and reduced.

By further analysis, we discovered that we can reduce the number of times when the objects were being de-serialized, especially in the mapper. The problem is that during grouping, Hadoop is de-serializing previously serialized objects in order to compare them to determine which group to place the new object into. To avoid this, JAQL is changed to serialize all of its data in such a fashion that values are binary-collatable. That is, given two buffers full of serialized data, the buffers can have a byte comparison performed in order to collate, rather than having to de-serialize either of the buffers.

By profiling, we are also able to catch the inefficiency points of JAQL implemented serialization and de-serialization functions. We identify RecordSerializer.partition, which deals with record, and ArraySerializer.write, which deals with array, as the hot methods since they are called many times and the total time spent on them is big.

JAQL, being initially designed for loosely structured data, allows for arrays of heterogeneous data, and structures that can be completely different from row to row. As a result, the process of serializing data consists of a significant amount of introspection of a value's type before serialization. To

[2] A JAQL record is an un-ordered collection of name-value pairs where name is a literal string and value can be an atomic value, record or list.

[3] A JAQL array is an ordered collection of values.

improve on this situation, we enhance JAQL to recognize situations in which the schema is fully computable, such as in SQL operations, and the general purpose serializers that JAQL would normally use are swapped out with special-purpose serializers that understand, for example, how to serialize only an array of non-nullable long values. These optimizations even include recognizing when a particular value is invariant (for example, could never be anything but the value "3") and avoiding serialization for such values altogether.

5.6 Other improvement

In the following, we describe other improvements we have made into JAQL.

By profiling, we observe that an important portion of execution time is spent on jsonIterator.hasNext as shown in Figure 6. Inside this function, Expr.eval is called to evaluate an expression's value. Once a value is recognized as math expression, MathExpr.evalRaw is called to get the value. However, this function will again call Expr.eval and MathExpr.evalRaw repeatedly, nested to many levels. We also observe that ArrayExpr.evalRaw is expensive due to the fact that it involves many layers of type check.

As with the scenario of serialization, described in the previous section, many operations in JAQL allow for heterogeneous data types. For example, in the expression

$$<\text{prev expr}> .. \rightarrow \text{filter } \$.a + \$.b > 10$$

the + and > operators will inspect their values each and every time they are called to determine the input types, how to compute the result, and what type the result should be, thus a significant amount of time is spent during this inspection and making these decisions. This was optimized by, again, recognizing the cases in which the types of the operands are invariant (based upon the schema of the expression) and replacing the general purpose operator with a special purpose operator that only knows how to perform the operation on the specific data type(s) involved in the operation.

5.7 Experiment Results

Next, we demonstrate the performance improvement of the proposed methods discussed above.

In Figure 8, we show the improvement ratio of each job generated by the queries. For example, q7, q7-j2, q7-j3 are the three jobs generated by query q7. The performance improvement of reducing the number of jobs as discussed above is small and is not shown in the figure. The reason is that only the jobs with small input are executed locally and the improvement is in the seconds level while the queries tested here will take minutes to finish. Thus, the improvement ratio is small.

The first set of bars in the figure shows the improvement by reducing the number of objects. The 2nd set of bars shows the improvement by reducing I/O (de)serialization overhead and other improvement. The 3rd set of bars shows the improvement of reducing the number of columns and the number of decoded bytes. We can see that reducing the I/O data size has a bigger effect than reducing (de)serialization overhead in our experiment. And with all the improvements, we are able to improve the performance of queries up to more than 2X.

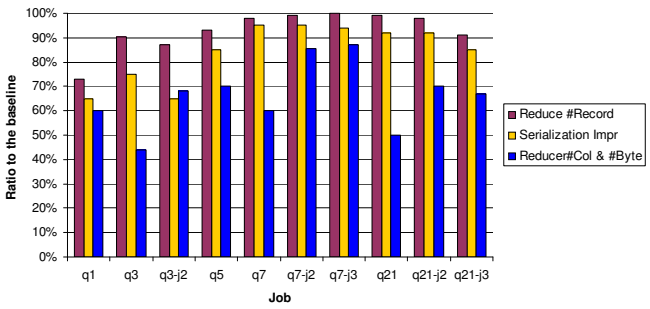

Figure 8: Performance Improvement

6. RELATED WORK

The authors of [15] provide a comparison of three high level query languages for Map reduce, namely, Pig, Hive, and JAQL, on the bases of their functional features and run-time performance. The scalability tests in this study are done using simple non-commercial benchmarks. A qualitative evaluation of these query languages and compilation of their queries into MapReduce jobs is presented in [14]. It was concluded that JAQL, with its expressive power and flexibility, is best suited for large-scale data processing in Big Data analytics. In [13] XQuery language is extended to support JSON data model and the XQuery processor is extended to support MapReduce execution.

MapReduce paradigm is based on isolated execution of individual map tasks (belonging to the same job) which sometimes restrict the choice of algorithms that can be executed in the map-phase. To overcome this limitation, an Adaptive Map-Reduce approach based on Situation-Aware Mappers (SAMs) was introduced in [8, 19]. SAMs which are basically mappers with an asynchronous communication channel between them for exchanging state information. This improves performance of a class of aggregate functions by limiting the output data from map jobs that needs to be shuffled and copied to reduce nodes. They also proposed a new API for developing User Defined Aggregates (UDAs) in JAQL to exploit SAMs.

A technique for run-time performance prediction of JAQL queries (with fixed data flows) over varying input data sets is developed in [12]. Such techniques can be used for optimal resource provisioning and scheduling by MapReduce schedulers like FLEX [20] to meet query performance related Service Level Agreements (SLAs).

The research of [7] describes eXtreme Analytics Platform (XAP), a powerful platform for large-scale data-intensive analytics developed at IBM Research. The main building blocks of XAP are JAQL, FLEX scheduler for optimized allocation of resources to Hadoop jobs, data-warehouses connectors and libraries and tools for advanced analytics. Many of the XAP technologies are incorporated in IBM InfoSphere BigInsights [4] product.

7. CONCLUSION

JAQL is a query language designed for large-scale data analysis. It provides an easy-to-use, flexible and extensible interface to the BigData analytics and explores massive parallelism using MapReduce framework. It is an important

component of BigInsights, an IBM flagship product on BigData, and its performance is critical for SQL queries submitted by Big SQL and other data analytics applications built on it.

We improve JAQL performance from multiple perspectives. We make JAQL thread-safe and further explore its parallelism; we profile JAQL intensively for performance analysis; we identify that JAQL compiler is dominant in execution time inside Big SQL and JAQL runtime for MapReduce is dominant in execution time inside MapReduce framework; we further identify I/O is critical for performance and be able to improve I/O performance by reducing the I/O data size and increase (de)serialization efficiency.

We profile and measure our improvements in a simple Hadoop cluster with special configuration to reduce the number of JVMs. We show that the performance of TPCH queries can be improved up to 2 times and the biggest improvement comes from the reduction of data size.

In the future, we would like to explore parallelism further inside JAQL. For example, we can check the data dependency beyond the range of one statement (one SQL query) and continue with the statements without data dependency as long as possible. As Hadoop YARN is available, we would like to investigate how JAQL will benefit from the new Hadoop infrastructure. We are also interested in the generation of query plans and investigate if we can reduce I/O data size by a better query plan.

8. REFERENCES

[1] Apache Hadoop. Available at http://hadoop.apache.org.

[2] IBM Big Data & Analytics Hub. Available at http://www.ibmbigdatahub.com/.

[3] IBM BigSheets. Available at http://www-01.ibm.com/software/ebusiness/jstart/bigsheets/.

[4] IBM InfoSphere BigInsights. Available at http://www.ibm.com/software/data/infosphere/biginsights/.

[5] IBM InfoSphere DataStage. Available at http://www-03.ibm.com/software/products/en/ibminfodata/.

[6] Tableau Software. Available at http://www.tableausoftware.com/.

[7] A. Balmin, K. Beyer, V. Ercegovac, J. McPherson, F. Ozcan, H. Pirahesh, E. Shekita, Y. Sismanis, S. Tata, and Y. Tian. A platform for extreme analytics. *IBM Journal of Research and Development*, 57(3/4):4:1–4:11, 2013.

[8] Andrey Balmin, Vuk Ercegovac, Rares Vernica, and Kevin S. Beyer. Adaptive processing of user-defined aggregates in jaql. *IEEE Data Eng. Bull.*, 34(4):36–43, 2011.

[9] Kevin S. Beyer, Vuk Ercegovac, Rainer Gemulla, Andrey Balmin, Mohamed Y. Eltabakh, Carl C. Kanne, Fatma Özcan, and Eugene J. Shekita. Jaql: A Scripting Language for Large Scale Semistructured Data Analysis. *PVLDB*, 4(12):1272–1283, 2011.

[10] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: a flexible data processing tool. *Communications of the ACM*, 53(1):72–77, 2010.

[11] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM.

[12] Adrian D. Popescu, Vuk Ercegovac, Andrey Balmin, Miguel Branco, and Anastasia Ailamaki. Same queries, different data: Can we predict query performance? In *Proceedings of the 7th International Workshop on Self Managing Database Systems*, Washington DC, USA, April 2012.

[13] Caetano Sauer, Sebastian BÃd'chle, and Theo HÃd'rder. Versatile xquery processing in mapreduce. In Barbara Catania, Giovanna Guerrini, and Jaroslav PokornÃ¡, editors, *Advances in Databases and Information Systems*, volume 8133 of *Lecture Notes in Computer Science*, pages 204–217. Springer Berlin Heidelberg, 2013.

[14] Caetano Sauer and Theo HÃd'rder. Compilation of query languages into mapreduce. *Datenbank-Spektrum*, 13(1):5–15, 2013.

[15] RobertJ. Stewart, PhilW. Trinder, and Hans-Wolfgang Loidl. Comparing high level mapreduce query languages. In Olivier Temam, Pen-Chung Yew, and Binyu Zang, editors, *Advanced Parallel Processing Technologies*, volume 6965 of *Lecture Notes in Computer Science*, pages 58–72. Springer Berlin Heidelberg, 2011.

[16] EJ technologies. Jprofiler manual. 2013.

[17] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2(2):1626–1629, August 2009.

[18] Transaction Processing Performance Council (TPC). Tpc benchmark^{TM} h standard specification revision 2.16.0. 2013.

[19] Rares Vernica, Andrey Balmin, Kevin S. Beyer, and Vuk Ercegovac. Adaptive mapreduce using situation-aware mappers. In *Proceedings of the 15th International Conference on Extending Database Technology*, EDBT '12, pages 420–431, New York, NY, USA, 2012. ACM.

[20] Joel Wolf, Deepak Rajan, Kirsten Hildrum, Rohit Khandekar, Vibhore Kumar, Sujay Parekh, Kun-Lung Wu, and Andrey Balmin. Flex: A slot allocation scheduling optimizer for mapreduce workloads. In Indranil Gupta and Cecilia Mascolo, editors, *Middleware 2010*, volume 6452 of *Lecture Notes in Computer Science*, pages 1–20. Springer Berlin Heidelberg, 2010.