Comparing High Level MapReduce Query Languages

Robert J. Stewart, Phil W. Trinder, and Hans-Wolfgang Loidl

Mathematical and Computer Sciences
Heriot Watt University

Abstract. The MapReduce parallel computational model is of increasing importance. A number of High Level Query Languages (HLQLs) have been constructed on top of the Hadoop MapReduce realization, primarily Pig, Hive, and JAQL. This paper makes a systematic performance comparison of these three HLQLs, focusing on *scale up*, *scale out* and *runtime* metrics. We further make a language comparison of the HLQLs focusing on conciseness and computational power. The HLQL development communities are engaged in the study, which revealed technical bottlenecks and limitations described in this document, and it is impacting their development.

1 Introduction

The MapReduce model proposed by Google [8] has become a key data processing model, with a number of realizations including the open source Hadoop [3] implementation. A number of HLQLs have been constructed on top of Hadoop to provide more abstract query facilities than using the low-level Hadoop Java based API directly. Pig [18], Hive [24], and JAQL [2] are all important HLQLs.

This paper makes a systematic investigation of the HLQLs. We investigate specifically, whether the HLQLs are indeed more abstract: that is, how much shorter are the queries in each HLQL compared with direct use of the API? What performance penalty do the HLQLs pay to provide more abstract queries? How expressive are the HLQLs - are they relationally complete, SQL equivalent, or even Turing complete? More precisely, the paper makes the following research contributions with respect to Pig, Hive, and JAQL.

A systematic performance comparison of the three HLQLs based on three published benchmarks on a range of Hadoop configurations on a 32 node Beowulf cluster. The study extends previous, predominantly single language studies. The key metrics we report are the *scale up*, *scale out* and *runtime* of each language. The performance baseline for each benchmark is a direct implementation using the Hadoop API, enabling us to assess the overhead of each HLQL (Section 6).

A language comparison of the three HLQLs. For each benchmark we compare the conciseness of the three HLQLs with a direct implementation using the Hadoop API using the simple *source lines of code* metric (Section 4.2). We further compare the expressive power of the HLQLs (Section 3.2).

O. Temam, P.-C. Yew, and B. Zang (Eds.): APPT 2011, LNCS 6965, pp. 58–72, 2011.

Our study is already impacting HLQL development. The HLQL development communities were engaged in the study, replicating our results and provided recommendations for each benchmark. A preliminary set of the results from the study [20] has been widely downloaded (1,200 downloads at present).

2 Related Work

2.1 MapReduce

Google propose MapReduce (MR) as "a programming model and an associated implementation for processing and generating large data sets" [8], and argue that many real world tasks are expressible using it. MR programmers are only required at a minimum to specify two functions: *map* and *reduce*. The logical model of MR describes the data flow from the input of *key/value* pairs to the *list* output:

```
Map(k1,v1) -> list(k2,v2)
Reduce(k2, list (v2)) -> list(v3)
```

The **Map** function takes an input pair and produces a set of intermediate key/-value pairs. The **Reduce** function accepts the intermediate *key2* and a set of values for that key. It merges these values to form a possibly smaller set of values [8]. The model has built-in support for fault tolerance, data partitioning, and parallel execution, absolving considerations like reliability and distributed processing away from the programmer.

In relation to High Performance Computing (HPC) platforms, [25] states that the MapReduce model is comparatively strong when nodes in a cluster require gigabytes of data. In such cases, network bandwidth soon becomes the bottleneck (leaving nodes idle for extended time periods) for alternative HPC architectures that use such APIs as Message Passing Interface (MPI), over shared filesystems on the network. MPI gives close control to the programmer who coordinates the dataflow via low-level routines and constructs, such as sockets, as well as higher-level algorithms for the analysis. In contrast, MapReduce operates only at the higher level, leaving the programmer to think in terms of functions on key and value pairs, and the data flow is implicit [25].

2.2 Hadoop

Hadoop [3] is an Apache open source MR implementation, which is well suited for use in large data warehouses, and indeed has gained traction in industrial datacentres at Yahoo, Facebook and IBM. The software stack of Hadoop is packaged with a set of complimentary services, and higher level abstractions from MR. The core elements of Hadoop however, are *MapReduce* - the distributed data processing model and execution environment; and the *Hadoop Distributed Filesystem* (HDFS) - a distributed filesystem that runs on large clusters. The HDFS provides high throughput access to application data, is suitable for applications that have large data sets, and the detection and recovery from faults is

a primary design goal of the HDFS. In addition, Hadoop provides interfaces for applications to move tasks closer to the data, as moving computation is cheaper than moving very large quantities of data [4].

Example MapReduce Application. Word count is a simple MapReduce application (Listing 1.1), commonly used for demonstration purposes [8]. The *map* function tokenizes a list of strings (one per line) as maps, and assigns an arbitrary value to each key. The reduce function iterates over this set, incrementing a counter for each key occurrence.

Listing 1.1. Pseudo Code: MapReduce Word Count

```
map( 'input.dat' ){
 Tokenizer tok <- file.tokenize();
   while (tok.hasMoreTokens){
     output(tok.next(),"1"); // list(k2,v2)
   }
}

reduce( word, values ){
 Integer sum = 0;
   while(values.hasNext()){
     sum += values.next();
   }
 output(word,sum); // list(v3)
}
```

3 High Level Query Languages

Justifications for *higher* level query languages over the MR paradigm are presented in [15]. It outlines the lack of support that MR provides for complex *N-step* dataflows, that often arise in real-world data analysis scenarios. In addition, explicit support for multiple data sources is not provided by MR. A number of HLQLs have been developed on top of Hadoop, and we review Pig [18], Hive [24], and JAQL [2] in comparison with raw MapReduce. Their relationship to Hadoop is depicted in Figure 1. Programs written in these languages are compiled into a sequence of MapReduce jobs, to be executed in the Hadoop MapReduce environment.

3.1 Language Query Examples

Pig - A High Level Data Flow Interface for Hadoop. *Pig Latin* is a high level dataflow system that aims at a sweet spot between SQL and MapReduce, by providing high-level data manipulation constructs, which can be assembled in an explicit dataflow, and interleaved with custom MR functions [15]. Programs written in *Pig Latin* are firstly parsed for syntactic and instance checking. The output from this parser is a logical plan, arranged in a directed acyclic graph,

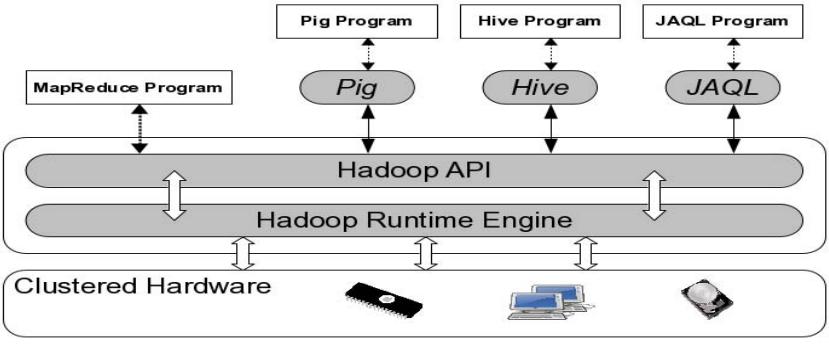

Fig. 1. HLQL Implementation Stack

allowing logical optimizations, such as *projection pushdown* to be carried out. The plan is compiled by a *MR compiler*, which is then optimized once more by a *MR optimizer* performing tasks such as early partial aggregation, using the MR *combiner* function. The MR program is then submitted to the Hadoop job manager for execution.

Pig Latin provides both simple scalar types, such as *int* and *double*, and also complex non-normalized data models. A *bytearray* type is supported, to facilitate unknown data types and lazy conversion of types, in addition to three collection types: *map, tuple* and *bag*. The Pig *word count* query is given in Listing 1.2.

Listing 1.2. Pig Word Count Benchmark

```
myinput = LOAD 'input.dat' USING PigStorage();
grouped = GROUP myinput BY \$0;
counts = FOREACH grouped GENERATE group,
         COUNT(myinput) AS total;
STORE counts INTO 'PigOutput.dat' USING PigStorage();
```

Hive - A Data Warehouse Infrastructure for Hadoop. *Hive QL* provides a familiar entry point for data analysts, minimizing the pain to migrate to the Hadoop infrastructure for distributed data storage and parallel query processing. Hive supports queries expressed in a SQL-like declarative language - HiveQL, which are compiled into MR jobs, much like the other Hadoop HLQLs. HiveQL provides a subset of SQL, with features like *from* clause subqueries, various types of *joins*, *group bys, aggregations* and *create table as select* all make HiveQL very SQL like.

Hive structures data into well-understood database concepts like tables, columns, rows, and partitions. It supports all the major primitive types: *integers, floats, doubles* and *strings*, as well as collection types such as *maps, lists* and *structs*. Hive also includes a system catalogue, a *metastore*, that contains schemas and statistics, which are useful in data exploration, query optimization and query compilation [23]. Just like with the Pig compilation process, Hive

includes a *query compiler*, which is a component that compiles HiveQL into a directed acyclic graph of MR tasks. The Hive *word count* query is given in Listing 1.3.

Listing 1.3. Hive Word Count Benchmark

```
CREATE EXTERNAL TABLE Text(words STRING)
LOCATION 'input.dat';
FROM Text INSERT OVERWRITE DIRECTORY 'HiveOutput.dat'
SELECT words, COUNT(words) as totals
GROUP BY words;
```

JAQL - A JSON Interface to MapReduce. *JAQL* is a functional data query language, which is built upon JavaScript Object Notation Language (JSON) [6]. JAQL is a general purpose data-flow language that manipulates semi-structured information in the form of abstract JSON values. It provides a framework for reading and writing data in custom formats, and provides support for common input/output formats like CSVs, and like Pig and Hive, provides operators such as *filtering, transformations, sort, group bys, aggregation,* and *join* [2].

JSON supports atomic values like numbers and strings, and has two container types: arrays and records of name-value pairs, where the values in a container are arbitrary JSON values. Databases and programming languages suffer an *impedance mismatch* as both their computational and data models are so different [1]. As the JSON model provides easy migration of data to and from some popular scripting languages like Javascript and Python, JAQL is extendable with operations written in many programming languages because JSON has a much lower impedance mismatch than XML for example, yet much richer datatypes than relational tables [2]. The JAQL *word count* query is given in Listing 1.4.

Listing 1.4. JAQL Word Count Benchmark

```
$input = read(lines('input.dat'));
$input -> group by $word = $
into { $word, num: count($) }
->write(del('JAQLOutput.dat',{fields:[''words'',''num'']}));
```

3.2 HLQL Comparison

Language Design. The language design motivations are reflected by the contrasting features of each high level query language. Hive provides Hive QL, a SQL like language, presenting a declarative language (Listing 1.3). Pig by comparison provides Pig Latin (Listing 1.2), a dataflow language influenced by both the declarative style of SQL (it includes SQL like functions), and also the more procedural MR (Listing 1.1). Finally, JAQL is a functional, higher-order programming language, where functions may be assigned as variables, and later evaluated (Listing 1.4). In contrast, Pig and Hive are strictly evaluated during the compilation process, to identify type errors prior to runtime.

Computational Power. Figure 2 illustrates the computational power of Pig, Hive, and JAQL, the MR model, and the MapReduceMerge model [27]. MapReduceMerge was designed to extend MR, by implementing relational algebra operators, the lack of which signifies that MR itself is not relationally complete. Indeed, MR is a simple model for applications to be encapsulated for computation in distributed environments.

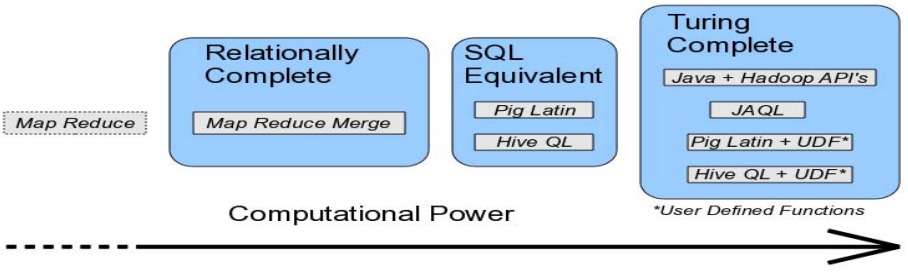

Fig. 2. Computational Power Comparison

Relational Completeness. Relational algebra is a mathematical notation that defines relations in standardized algebraic syntax which are combined to create the operators and functions required for relational completeness [7]. The structured query language, SQL, is regarded as relationally complete as it provides all operations in relational algebra set, and in addition offers a set of aggregation functions, such as *average, count*, and *sum.*

Turing Completeness. A language that contains conditional constructs; recursion capable of indefinite iteration; and a memory architecture that emulates an infinite memory model is said to be Turing Complete.

User Defined Functions. Pig, Hive and JAQL are all extendable with the use of user defined functions (UDF). These allow programmers to introduce custom data formats and bespoke functions. They are always written in Java, a Turing complete language.

4 Conciseness

4.1 Application Kernels

The HLQLs are benchmarked using three applications: two that are canonical and published, and a third that is a typical MR computation. The word count in Figure 3 is the canonical MR example [10]. The dataset join in Figure 4 is another canonical example, with published implementations for Java, Pig, Hive and JAQL [9,26,11,16]. The web log aggregation example in Figure 5 is a new benchmark that is typical of webserver log manipulation. Each can be described concisely in SQL, as:

```
SELECT words, COUNT(words) as totals
GROUP BY words;
```

Fig. 3. Word Count

```
SELECT t1.*
FROM TABLE1 t1
JOIN TABLE2 t2
  ON (t1.field = t2.field);
```

Fig. 4. Dataset Join

```
SELECT userID, AVG(timeOnSite) as averages, COUNT(pageID)
GROUP BY userID;
```

Fig. 5. Web Log Processing

4.2 Comparative Code Size

Table *1* shows the source lines of code count to satisfy the requirements used in the word count, join, and web log processing applications.

Table 1. Source Lines of Code Comparison

	Java	Pig	*Pig/Java Ratio*	JAQL	*JAQL/Java Ratio*	Hive	*Hive/Java Ratio*
Word Count	45	4	8.9%	6	13.3%	4	8.9%
Join	114	5	4.4%	5	4.4%	13	11.4%
Log Processing	165	4	2.4%	3	1.8%	11	6.7%
Mean Ratio	(100%)		**5.2%**		**6.5%**		**9%**

The aim of the high level languages Pig, Hive and JAQL is to provide an abstract data querying interface to remove the burden of the MR implementation away from the programmer. However, Java MR applications requires the programmer to write the *map* and *reduce* methods manually, and hence require program sizes of a magnitude of at least 7.5 (Table 1). It would appear that conciseness is thus achieved for the equivalent implementations in the HLQLs, and Section 6 will determine whether or not programs pay a performance penalty for opting for these more abstract languages. Refer to further language discussion, in Section 7.

5 Performance Experiment Design

This section outlines the runtime environment used for the performance comparisons, justifies the number of reducers used in each experiment, and explains the importance of key distribution for MapReduce applications.

5.1 Language Versions, Hardware and Operating Systems

The experiments utilize Pig version 0.6, Hive version 0.4.0, an *svn* snapshot of JAQL (r526), and Hadoop version 0.20.1. The benchmarks were run on a cluster

comprising of 32 nodes running Linux CentOS 5.4; Kernel 2.6.18 SMP; Intel Pentium 4, 3Ghz dual core, 1MB cache, 32bit; 1GB main memory; Intel Pro 1Gbps Full Duplex; HDD capacity 40GB.

5.2 Reducer Tasks

Guidelines have been outlined for setting a sensible value for the number of reducers for Hadoop jobs. This value is determined by the *number of Processing Units* (nodes) **PU**s in the cluster, and *the maximum number of reducers* per node **R**, as $\mathbf{PU} \times \mathbf{R} \times \mathbf{0.9}$ [17].

5.3 Key Distribution

A potential bottleneck for MR performance can occur when there exists a considerable range in the number of values within the set of keys generated by the *map* function (see Section 2.1). In Section 6.1, we measure the performance of each language when processing input data with skewed key distribution. The skewed dataset was generated using an existing package [22], developed by the Pig language design team at Yahoo.

6 Performance Results

The application kernels outlined in Section 4.1 are used in this section to measure the language performance for scale-up, scale-out and runtime. It also shows the effect on runtime when tuning the number of reducers, for each language. A full specification of the Hadoop runtime parameters for all of the experiments that are described, can be found in Section 3.2 of [19].

6.1 Scaling Input Size

To measure the *scale-up* performance of each language, the size of the cluster was fixed at *20 PU's*. Scaling with uniform key distribution computation is shown in figures 6 and 7, and with a skewed key distribution computation is shown in figures 8 and 9.

A prominent feature of these measurements is that total runtime increases with input size. Two trends emerge, such that JAQL and Pig achieve similar scale up performance, though both Hive and Java perform considerably better. In the web log processing with the smallest input size, JAQL is the most efficient processing engine (Figure 7), achieving the lowest runtime of all of the languages - 26% quicker than the lower level Java implementation.

Skewed Keys. The word count and join applications were executed on datasets with skewed key distribution, in Figures 8 and 9. Both JAQL and Java performances are impaired by the skewed key distribution in the *join* application, scaling up less well. On feeding preliminary join results (Figure 9) back to the

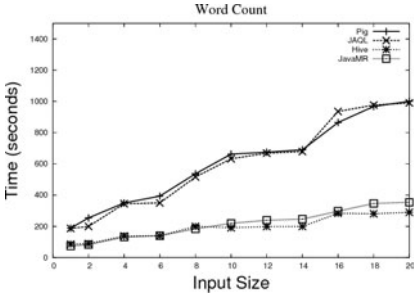

Fig. 6. Word Count Scale Up - Uniform Distribution

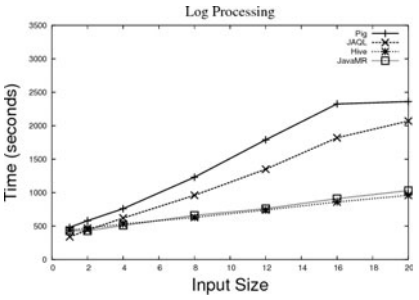

Fig. 7. Web Log Processing Scale Up - Uniform Distribution

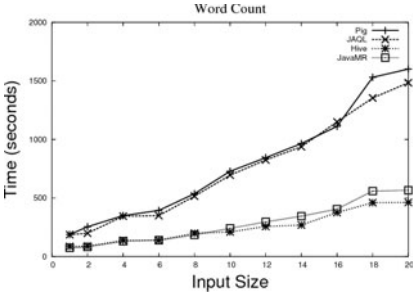

Fig. 8. Word Count Scale Up - Skewed Distribution

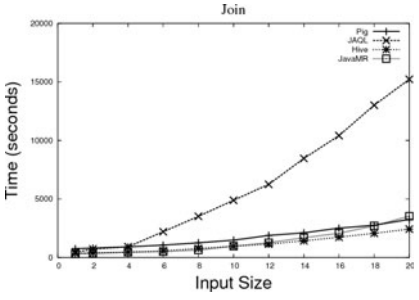

Fig. 9. Dataset Join Scale Up - Skewed Distribution

JAQL development team, a problem with the JAQL delimited output method was identified, and subsequently fixed [21]. Both Pig and Hive appear to handle the skewed key distribution more effectively. In Figure 9, where input size is *x20*, both Pig and Hive outperform the Java implementation. A complete result set from the skewed key runtime experiments can be found at [19].

6.2 Scaling Processing Units

To measure the *scale-out* performance of each language, the size of computation was fixed, and the number of worker nodes was increased from 1 to 20. The results of these *scale-out* benchmarks are depicted in Figures 10 and 11.

Beyond 4 PU's, there is no apparent improvement for JAQL, whereas Pig, Hive and Java are all able to utilize the additional processing capacity up to 16 PU's, at which point, no further cluster expansion is beneficial to runtime performance.

A fundamental contrast between the *join* and the *web log processing* application, in Figures 10 and 11 respectively, is the input size for the two applications.

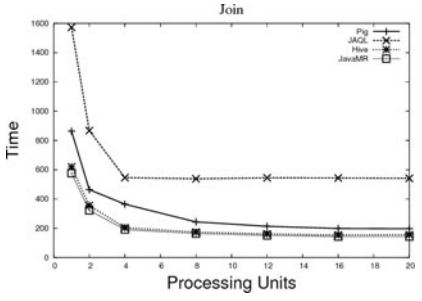

Fig. 10. Runtime for Dataset Join

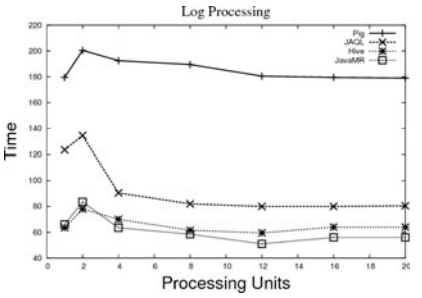

Fig. 11. Runtime for Web Log Processing

As a result, less work is required by every run of the *web log processing* application, and more for the *join* application. These results clearly illustrate one common design challenge for parallel systems - In the *join* application (Figure 10), a sufficient workload eluded to a diminished runtime, shifting from using 1 PU, to 2 - a trivial case study. However, there was only modest computational requirements for the *web log processing* application, Figure 11. It appears that, as a consequence, the communication and coordination overheads associated with work distribution (2 or more PU's) is detrimental to overall runtime performance.

6.3 Proportional Scale Out of Input Size and Processing Units

In this experiment, the required computation (**C**) is increased in proportion with the number of PU's **P** in the cluster, with a multiplying factor **m**. Ideal performance would achieve no increase in the computation time **T** as **m** increases.

$$\mathbf{T = mC \ / \ mP}$$

In reality however, this is not attainable, as communication and task coordination costs influence overall runtime, hindering runtime performance.

Whilst Java achieves the quickest runtime for the *join* operation when *scaling out* (Figure 12), Pig and Hive achieve performance parity with Java for the *relative* scale out performance, Figure 13. The exception is JAQL, which scales less well.

6.4 Controlling Reducers

All MR jobs are split into many *map* and *reduce* tasks. The computational granularity - the computation size per reduce task, may have a significant influence of performance.

Whilst the number of *map tasks* for a MR job is calculated by the Hadoop runtime engine, there is a degree of flexibility for the number of *reduce tasks*

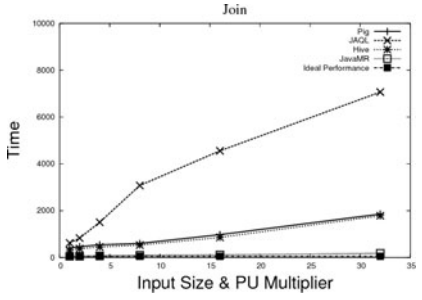

Fig. 12. Dataset Join Scale Out

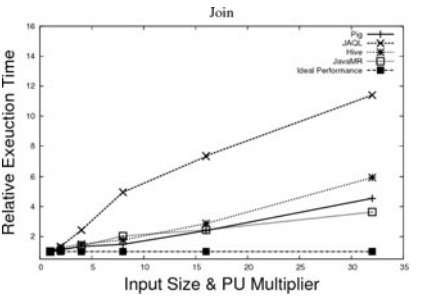

Fig. 13. Relative Dataset Join Scale Out

associated with a job. When these experiments were conducted in May 2010, Hive, Pig and the Hadoop Java API provided an additional performance tuning parameter to specify the number of reducers to be used, whilst JAQL did not. Figures 14 and 15 are annotated where the reducers parameter is set at 18, in accordance with the formula in Section 5.2.

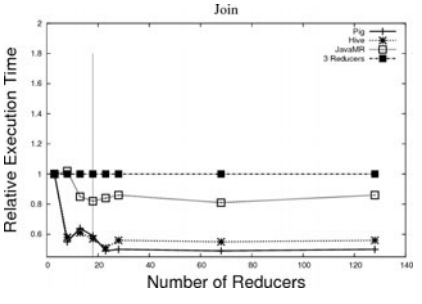

Fig. 14. Controlling Reducers: Dataset Join

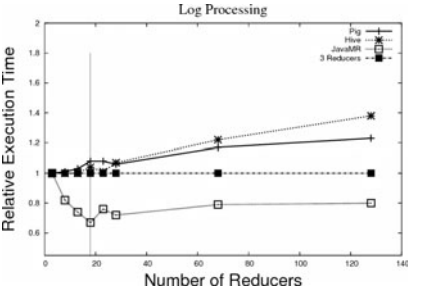

Fig. 15. Controlling Reducers: Log Processing

A consequence of too-fine granularity is in an increase in runtime as the reduce tasks parameter increases to 128, shown in Figure 15. In addition, poor runtime performance is seen where the *reduce tasks* parameter is set well below the recommended value, 18. The *join* application in Figure 14 shows that despite the parameter guidance value of 18, *Pig* and *Java* achieve quickest runtime at 23 reducers, after which, performance gradually degrades.

In summary, the additional expressive power of controlling the number of reducer tasks can optimize performance by as much as 48% (see Figure 14) compared to the default setting, set by the default Hadoop configuration, of just 1 reducer. In addition, the results illustrate that the guideline for the value of this parameter (detailed in Section 5.2) is approximately optimal for this *"control knob"* in achieving the quickest runtime for the application.

7 Language Discussion

A common feature of Pig, Hive and JAQL is that they can be extended with special-purpose Java functions. This means that they are not limited to the core functionality of each language, and the computational power is increased with the use of such user defined functions, as detailed in Section 3.2.

Pig is the most concise language, with an average ratio (relative to Java) of just 5.2%, as shown in Table 1. Pig has built-in optimization for joining skewed key distribution, which outperforms data handling found in a non-adaptive Java MR join application, shown in Figure 9. The discussion of tuning reduce tasks in Section 6.4, in Figure 14, shows that Pig takes advantage of an increasing number of reduce tasks, and handles an undesirably high level of specified reducer tasks comparatively well in Figure 15.

Hive was the best performer for the scale-up, scale-out and speed-up experiments for each of the three benchmarks, and throughout these experiments, overall runtime was only fractionally slower than Java. Like Pig, Hive took advantage of tuning the reducers parameter in the join application, in Figure 14. Both Pig and Hive are shown to have SQL equivalent computational power[1].

JAQL is a lazy higher-order functional language and is Turing Complete, whereas Pig Latin and Hive QL are not. Throughout the benchmark results, JAQL does not compete with the runtime of Hive or Java. However, it does achieve a speed-up of approximately 56% for the join benchmark, shown in Figure 10. The runtime improvements through controlling the number of reducers (Section 6.4) were not measured for JAQL at the date of experiment execution (May, 2010), though this feature has subsequently been added to the language [5]. This highlights one challenge for comparative studies such as this report - these languages are relatively new, and are moving targets when attempting to construct fair and relevant performance and feature comparisons.

One particularly apt challenge when writing comparative studies such as this, is that each project - each HLQL, is a "moving target". Active development continues to evolve each language, with new features being added with each release. Pig 0.8, for example, was released on the 17th December 2010, and provides support for UDFs in scripting languages, and also a safeguard against the omission of the "number of reducers" tuning parameter. Given such an omission, Pig uses its own heuristic to set this value, to avoid inefficient runtimes discussed in Section 6.4. Pig version 0.8 also introduces a *PigStats* utility, enabling "Pig to provide much better visibility into what is going on inside a Pig job than it ever did before" [12]. This utility would be an invaluable tool for further comparisons, and tuning, of Pig performance for more opaque and complex programs. Likewise, Hive 0.7 was released on the 29th March 2011, which adds data indexing, a

[1] With the minor exception that Pig does not currently provide support for arbitrary *theta* joins.

concurrency model, and an authentication infrastructure, amongst other things [13].

The complimentary nature of these HLQLs is discussed in [14], by a member of the Pig development team. The analogy outlines three separate tasks involved with data processing: *data collection*; *data preparation*, performed in "data factories"; and *data presentation*, performed in "data warehouses", and this study has focused on the latter two. The data engineers at Yahoo appear to believe that Pig, with its support for pipelines and iterative processing, is a good solution for data preparation. Similarly Hive, with its support for ad-hoc querying and its query support for business intelligence tools, is a natural fit for data presentation. The conclusion in [14] is that the combination of Pig and Hive therefore provides a complete data processing toolkit for Hadoop.

8 Conclusion

This report has presented a comparative study of three high level query languages, that focus on data-intensive parallel processing, built on top of the MapReduce framework in Hadoop. These languages are Pig, Hive and JAQL, and a set of benchmarks were used to measure the scale-up, scale-out and runtime of each language, and ease of programming and language features were discussed.

Hive was shown to achieve the quickest runtime for every benchmark, whilst *Pig* and *JAQL* produced largely similar results for the scalability experiments, with the exception of the *join* application, where *JAQL* achieved relatively poor runtime. Both *Hive* and *Pig* have the mechanics to handle skewed key distribution for some SQL like functions, and these abilities were tested with the use of skewed data in the *join* application. The results highlight the success of these optimizations, as both languages outperformed Java when input size was above a certain threshold.

The report also highlighted the abstract nature of these languages, showing that the source lines of code for each benchmark is much smaller than the Java implementation for the same benchmark, by a factor of at least 7.5. *Pig* and *Hive* enable programmers to control the number of reducers within the language syntax (as of May 2010), and this report showed that the ability to tune this parameter can greatly improve the runtime performance for these languages. JAQL was shown to be the most computationally powerful language, and it was argued that JAQL was *Turing Complete*.

References

1. Atkinson, M.P., Buneman, P.: Types and persistence in database programming languages. ACM Comput. Surv. 19(2), 105–190 (1987)
2. Beyer, K.S., Ercegovac, V., Krishnamurthy, R., Raghavan, S., Rao, J., Reiss, F., Shekita, E.J., Simmen, D.E., Tata, S., Vaithyanathan, S., Zhu, H.: Towards a scalable enterprise content analytics platform. IEEE Data Eng. Bull. 32(1), 28–35 (2009)

3. Borthakur, D.: The Hadoop Distributed File System: Architecture and Design (2007), http://www.hadoop.apache.org
4. Borthakur, D.: The Hadoop Distributed File System: Architecture and Design. The Apache Software Foundation (2007)
5. code.google.com/p/jaql. Jaql developers message board, http://groups.google.com/group/jaql-users/topics
6. Crockford, D.: The application/json media type for javascript object notation (json). RFC 4627 (Informational) (July 2006)
7. Date, C.J.: An Introduction to Database Systems. Addison-Wesley Longman Publishing Co., Inc., Boston (1991)
8. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Commun. ACM 51(1), 107–113 (2008)
9. The Apache Software Foundation. Hadoop — published java implementation of the join benchmark, http://goo.gl/R4ZRd
10. The Apache Software Foundation. Hadoop — wordcount example, http://wiki.apache.org/hadoop/WordCount
11. The Apache Software Foundation. Hive — language manual for the join function, https://cwiki.apache.org/confluence/display/Hive/LanguageManual+Joins
12. The Apache Software Foundation. Pig 0.8 — release notes (December 2010), http://goo.gl/ySUln
13. The Apache Software Foundation. Hive 0.7 — release notes (March 2011), http://goo.gl/3Sj67
14. Gates, A.: Pig and hive at yahoo (August 2010), http://goo.gl/OVyM1
15. Gates, A.F., Natkovich, O., Chopra, S., Kamath, P., Narayanamurthy, S.M., Olston, C., Reed, B., Srinivasan, S., Srivastava, U.: Building a high-level dataflow system on top of map-reduce: the pig experience. In: Proc. VLDB Endow., vol. 2, pp. 1414–1425 (August 2009)
16. IBM. Jaql — language manual for the join function, http://code.google.com/p/jaql/wiki/LanguageCore#Join
17. Murthy, A.C.: Programming Hadoop Map-Reduce: Programming, Tuning and Debugging. In: ApacheCon US (2008)
18. Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A.: Pig latin: a not-so-foreign language for data processing. In: SIGMOD 2008: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, pp. 1099–1110. ACM, New York (2008)
19. Stewart, R.J.: Performance and programmability comparison of mapreduce query languages: Pig, hive, jaql & java. Master's thesis, Heriot Watt University, Edinburgh, United Kingdom (May 2010), http://www.macs.hw.ac.uk/~rs46/publications.php
20. Stewart, R.J.: Slideshow presentation: Performance results of high level query languages: Pig, hive, and jaql (April 2010) http://goo.gl/XbsmI
21. JAQL Development Team. Email discussion on jaql join runtime performance issues. private communication (September 2010)
22. Pig Development Team. Pig DataGenerator, http://wiki.apache.org/pig/DataGeneratorHadoop
23. Thusoo, A., Sarma, J.S., Jain, N., Shao, Z., Chakka, P., Zhang, N., Anthony, S., Liu, H., Murthy, R.: Hive - a petabyte scale data warehouse using hadoop. In: ICDE, pp. 996–1005 (2010)

24. Thusoo, A., Sarma, J.S., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P., Murthy, R.: Hive: a warehousing solution over a map-reduce framework. In: Proc. VLDB Endow., vol. 2(2), pp. 1626–1629 (2009)
25. White, T.: Hadoop — The Definitive Guide: MapReduce for the Cloud. O'Reilly, Sebastopol (2009)
26. Yahoo. Pigmix — unit test benchmarks for pig, http://wiki.apache.org/pig/PigMix
27. Yang, H.-c., Dasdan, A., Hsiao, R.-L., Parker, D.S.: Map-reduce-merge: simplified relational data processing on large clusters. In: SIGMOD 2007: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, pp. 1029–1040. ACM, New York (2007)