A Performance Comparison of Parallel DBMSs and MapReduce on Large-Scale Text Analytics

Fei Chen, Meichun Hsu
HP Labs
fei.chen4@hp.com, meichun.hsu@hp.com

ABSTRACT

Text analytics has become increasingly important with the rapid growth of text data. Particularly, *information extraction* (IE), which extracts structured data from text, has received significant attention. Unfortunately, IE is often computationally intensive. To address this issue, MapReduce has been used for large scale IE. Recently, there are emerging efforts from both academia and industry on pushing IE inside DBMSs. This leads to an interesting and important question: Given that both MapReduce and parallel DBMSs are for large scale analytics, which platform is a better choice for large scale IE? In this paper, we propose a benchmark to systematically study the performance of both platforms for large scale IE tasks. The benchmark includes both statistical learning based and rule based IE programs, which have been extensively used in real-world IE tasks. We show how to express these programs on both platforms and conduct experiments on real-world datasets. Our results show that parallel DBMSs is a viable alternative for large scale IE.

1. INTRODUCTION

Recently we have witnessed the rapid growth of text data, including Web pages, emails, social media, etc. Such text data contain valuable knowledge. To tap such knowledge from text data, text analytics has become increasingly important. Particularly, *information extraction* (IE), which extracts structured data from text, has received significant attention [27].

Unfortunately, IE is often computationally intensive [26, 28, 34]. The fast growing amount of machine-generated and user-generated data, the majority of which is unstructured text, makes the need of highly scalable tools even more appealing. To address this issue, MapReduce has been used for large scale IE [20, 35].

On the other hand, there are emerging efforts from both academia and industry on pushing IE inside DBMSs [33, 34, 22, 18]. These works encapsulate IE inside *user defined functions* (UDFs), and then leverage the DBMS engines to scale up these in-memory IE solutions to disk resident data. Furthermore, several new-generation DBMSs, equipped with *massive parallel processing* (MPP) architectures, automatically parallelize UDFs and queries using multiple independent machines.

Given that both MapReduce and parallel DBMSs are options for large scale analytics, it is both theoretically and practically important to understand which platform is a better choice for large scale IE. This is the question we ask in this paper. This study can help research community and industry vendors to understand how to improve both platforms to support IE tasks, or how to combine the advantages of both platforms to build an even better hybrid platform [29]. While there are many aspects (such as fault tolerance, elasticity, etc) to be considered when comparing MapReduce and parallel DBMSs, response time is one of the most important factors. Therefore, we focus on response time comparisons in this paper.

While there are a few recent works [16] [23] on comparing MapReduce and parallel DBMSs, they mainly focused on *relational* queries. In contrast, our work focus on *IE workflows*. In terms of benchmarks on text analytics, previous benchmarks either focused on the *quality* of the text analytics approaches [1] instead of *response time*, or focused on the *document retrieval* task [12], where the goal is to retrieve the most relevant documents given a query, instead of the *IE task*.

In order to define the benchmark of large scale IE tasks, we first categorize 3 types of IE operators which have been widely used as building blocks in real-world IE tasks. Then we consider several IE workflows consisting of these IE operators. These workflows have also been extensively used in typical IE tasks such as event extraction and entity reconciliation [11].

We choose Hadoop implementation of MapReduce and a leading commercial MPP DBMS, Vertica, for testing. In order to express the IE workflows, we use PigLatin, a high level language on Hadoop and SQL accordingly. The implementations on both platforms leverage both built-in operators such as the relational operators and UDFs.

We acknowledge that the evaluation in this paper only considered one system in parallel DBMSs and one system in MapReduce. We also understand that using other systems may produce different results. However, both Vertica and Hadoop/Pig are representative and leading systems. Therefore, the evaluation results from the two systems can gain us some initial understanding of the emerging big data analytics. In the future, we will extend our work to include

EDBT/ICDT '13 March 18 - 22 2013, Genoa, Italy

other systems.

Contributions: To summarize, we have made the following contributions in this paper:

- As far as we know, we are the first to propose a benchmark to systematically study large scale IE on parallel DBMSs and MapReduce.
- We categorize the fundamental building blocks of IE and design IE workflows which have been widely used in real-world IE tasks.
- We show how to express these workflows on both platforms using built-in operators and UDFs.
- Our results show that UDF performance can significantly impact the performance of overall IE workflows, suggesting UDF-centric optimizations as a future research direction.
- Our results also show that while UDFs run on DBMSs at least as efficiently as on MapReduce, complex workflows with relational operators run far more efficiently on DBMSs than on MapReduce. This demonstrates that parallel DBMSs is a viable alternative for large scale IE.

2. RELATED WORK

MapReduce and Parallel DBMSs Benchmarks: There have been a few works [16, 23] on comparing the performance of MapReduce and parallel DBMSs recently. However, these works mainly focused on relational queries, instead of IE workflows. Expressing IE workflows on both platforms involves features such as text manipulation operators and UDFs, which is not a focus of the previous benchmarks.

Text Analytics Benchmarks: Both IR and Database communities have created text analytics benchmarks [1, 12]. These works differ from ours mainly in two aspects. First, most of these works focused on the task of *document retrieval* instead of *information extraction*. Second, these benchmarks are either not targeted at measuring the response times or only targeted at response time on single node systems. In contrast, our benchmark focuses on evaluating the *response time of systems on multiple nodes*.

Pushing Analytics into DBMSs: There are emerging efforts on pushing analytics such as sophisticated machine learning algorithms into DBMSs [17, 33, 34, 22, 13, 18]. These efforts mainly focused on developing individual in-DBMS solutions for different analytics algorithms, which is complementary to our benchmark work.

Large Scale Information Extraction: The problem of IE has received much attention. Recent work [28, 35, 32, 9, 8] has considered how to improve the runtime in large-scale IE applications. Our work falls into this direction. However, these previous efforts either focused on single node solutions or only focused on MapReduce solutions.

3. BACKGROUND

3.1 Parallel DBMSs and Vertica

In parallel DBMSs, tables are partitioned over nodes in a cluster and the system uses an optimizer that translates SQL commands into a query plan executed on multiple nodes. The new generation of parallel DBMSs are equipped with MPP architectures. Such MPP architectures consist of independent processors executing in parallel, and are mostly implemented on a collection of shared nothing machines, allowing better scale-out capability.

Vertica is one of the leading commercial MPP RDBMSs. Besides the MPP architecture, its another feature is storing data by columns, which enables more efficient compression of data and better I/O performance.

Like many DBMSs, Vertica supports text manipulations in several ways. First, it supports character data type. Furthermore, Vertica provides several built-in string manipulation operators, including regular expression functions compatible with Perl 5. These features make writing IE workflows easier for users.

Besides built-in operators, Vertica also allows users to write their own operators/functions as UDFs. Such UDFs allow users to execute more sophisticated data operations such as statistical learning based IE, which are hard to expressed as native SQL. The Vertica execution engine leverages MPP architectures to automatically run UDFs among multiple nodes where data are distributed. We will discuss more about Vertica UDFs in Section 3.3.

3.2 MapReduce, Hadoop and Pig

MapReduce is a programming model for processing large scale of data on multiple nodes. Hadoop is an open-source implementation of MapReduce. Besides providing the programming language support, Hadoop provides a distributed file system called HDFS.

Analytics workflows often consist of multiple MapReduce jobs. To help users write such series of MapReduce jobs, there are many higher level languages developed. Pig is a platform on top of Hadoop which provides a high level language called PigLatin. It provides built-in operators similar to those provided by DBMSs, with which users can encode complex tasks comprised of multiple interrelated data transformations as data flow sequences, making them easy to write and maintain. Furthermore, like DBMSs, Pig automatically optimizes the execution of these complex programs, allowing users to focus on semantics instead of efficiency. Finally, Pig also allows users to create their own operators as UDFs, which we will discuss in detail in Section 3.3.

Although there are other high level language platforms such as Hive [30], we choose Pig in our benchmark studies because (1) it shares several similarities with DBMSs, and (2) it is one of the most popular platforms. We will consider other platforms in future works.

3.3 UDFs

There are two kinds of Vertica UDFs: *scalar* UDFs and *transform* UDFs. A UDF is a scalar UDF if it takes in a *single* row and outputs a *single* value. Otherwise, it is a transform UDF. Our benchmark includes both types.

Users develop Vertica UDFs by instantiating 3 interfaces: *setup*, *processBlock* and *destroy*. Setup and destroy are used to allocate and release resources used by UDFs respectively. processBlock is where users specify their processing logics. Vertica partitions data into blocks as basic units of invoking UDFs. Setup and destroy are invoked once for each block, and processBlock is invoked repeatedly within a block.

input	Tom	Cruise	was	born	in	NY
output	P	P	O	O	O	L

Figure 1: Using CRFs to extract named entities.

Similar to Vertica UDFs, Pig also has simple *eval* UDFs and *aggregation* UDFs which operate on a single row and a set of rows respectively. Writing Pig UDFs is mainly by instantiating the *exec* interface, which is similar to processBlock in Vertica. Although Pig does not explicitly provide interfaces similar to setup and destroy in Vertica, there are workarounds which allow users to achieve the same goals.

3.4 Information Extraction (IE)

IE and Extractors: IE is to extract structured data from text. We call programs used to achieve this goal *extractors*. Formally, given a predefined schema, an extractor takes in a piece of text and outputs tuples to populate the given schema. Each output tuple contains at least one attribute value which is a substring of the given text. Extractors are often a set of handcrafted rules or learning based models. We single out 3 types of extractors extensively used in many real world IE tasks.

1. Learning-based Extractors and Conditional Random Fields (CRFs): Learning-based extractors employ a learning model such as hidden markov models and support vector machines for extraction. Usually, these learning models are first trained using a sample of data. Then they are deployed and applied repeatedly on large-scale datasets. We focus on model application in our benchmark, as this is the phase which often involves big data.

Particularly, we focus on a state-of-the-art learning model in IE, conditional random fields (CRFs). CRF-based extractors is a workhorse of the many real world IE systems [36, 19], and has been used on many IE tasks, including *named entity extraction* [15, 21], *table extraction* [25], and *citation extraction* [24].

Figure 1 illustrates the input and output of a CRF-based extractor. Given as input a sequence of tokens (e.g. tokens from a single sentence) and a set of labels, CRFs are probabilistic models that tag each token with one label from the given set of labels. In this example, the CRF model has been trained to extract named entities, therefore the set of labels include People (P), Locations (L) and Others (O).

CRFs is a very powerful statistical model because it considers the dependency between tokens in order to determine their labels. To this end, it employs a global optimization algorithm called *Viterbi* for inference. We refer readers to [21] for details.

2. Regular Expression Based Extractors: Regular expressions are often used in rule-based extractors. The regular expressions are often hand crafted by domain experts to capture the patterns in text. Matching strings with regular expressions is often time consuming. To address this challenge, there have been extensive research works on constructing efficient regular expression matching engines [10, 26, 28]. Instead of implementing these customized solutions, our benchmark focuses on the built-in regular expression operators provided by both Vertica and Hadoop/Pig to understand how well they support large scale IE tasks.

3. Dictionary Matching Based Extractors: Another kind of rule-based extractors match strings with a set of strings in a given dictionary. If two strings are "similar" enough, a match is produced. Such dictionary matching based extraction has been widely used for IE tasks such as *entity reconciliation* [11]. Since a straightforward implementation incurs quadratic number of string comparisons, many solutions [17, 32] have been proposed to improve its efficiency. In our benchmark, we choose an implementation [17] which is relatively easy to express using both SQL and PigLatin, leaving other solutions for future studies.

IE Workflows: For complex IE tasks, enclosing the entire IE program as a singleton is often hard to debug and maintain. Therefore, the common practice is to decompose a complex IE task into smaller subtasks, apply off-the-shelf IE modules or write hand-crafted code to solve each subtask, and then "stitch" them together and conduct final processing [14]. Besides extractors, relational operators have been used to compose such complex IE workflows [26, 28]. We include two IE workflows consisting of relational operators and extractors in our benchmark.

4. EVALUATIONS

We first introduce the setup and datasets of our benchmark evaluations in Section 4.1 and Section 4.2 respectively. Next, we present our evaluation results, including the performance evaluations of loading data (Section 4.3), IE tasks using only simple workflows (Section 4.4) and those using complex IE workflows (Section 4.5). Finally, we present the summary and discussions of our results in Section 4.6.

4.1 Benchmark Environment

Clusters Setup: We used a 16-node cluster that runs Red Hat Enterprise Linux, rel. 5.8, (kernel 2.6.18-308.e15 x86 64) and each node has 4 Quad Core Intel Xeon Processors X5550 (8M Cache, 2.66 GHz, 6.40 GT/s), 48GB of RAM, and 8x275GB SATA disks. 8 nodes were used for Hadoop/Pig, and the other 8 were used for Vertica.

Software Setup: We installed Hadoop version 0.20 and Pig version 0.9.1 running on Java 1.6.0. We deployed all systems with the default configuration settings, except that we set the maximum JVM heap size in Hadoop to 2GB per task to satisfy the memory requirement of all the UDFs. Particularly, Vertica compresses data by default and Hadoop HDFS replication factor is 3 by default. We kept these default settings as they are used in typical deployment.

4.2 Data

We downloaded 100,000 Wikipedia articles from the March 2012 dump [2]. In order to support various IE tasks on these articles, we preprocessed these articles as follows. First, we tokenized each article, and then used a sentence splitter [3] to segment the articles into sentences. Finally, given each sentence, a Part-of-Speech (POS) tagger [3] was used to tag each token in the sentence.

Data Schema: The above process resulted in two tables: sentences and tokens. The sentences table contains one tuple for each sentence detected in the corpus. Each sentences tuple contains *did*, the ID of the article from which the sentence is extracted, *sid*, the sentence ID which indicates its position in the sentence sequence of that article, and the text of the sentence. The tokens table contains one tuple for each

Attr. Name	Attr. Type
did	int
sid	int
sentence	varchar (25000)

(a) sentences

Attr. Name	Attr. Type
nid	int
name	varchar (128)

(c) dictionary

Attr. Name	Attr. Type
did	int
sid	int
tid	int
token	varchar (24)
pos	varchar (8)

(b) tokens

Figure 2: Vertica schema definitions. The numbers indicate string sizes in bytes.

Table Name	Meta Data	
sentences	# of tuples	2.5M
	file size	1.1G
	ave length of a sentence	76 characters
	min length of a sentence	1 character
	max length of a sentence	23K characters
tokens	# of tuples	193M
	file size	3.9G
	ave length of a token	4 characters
	min length of a token	1 character
	max length of a token	24 characters
dictionary	# of tuples	453K
	file size	10.5 M
	ave length of a name	14 characters
	min length of a name	7 characters
	max length of a name	74 characters

Figure 3: Meta data of sentences, tokens and dictionary.

token in the corpus. Similar to the sentences tuple, each tokens tuple contains the IDs of the article and sentence from which the token is extracted. Additionally, it also contains *tid*, *token* and *pos*. They indicate its position in the token sequence of the sentence, the token string, and the POS tag of the token respectively.

Besides the Wikipedia corpus, we also downloaded a list of entity names from Freebase [4], a *structured* Wikipedia-like portal. The resulting dictionary table contains one tuple for each name. Each tuple contains a name ID and the name string.

Figure 2 and Figure 3 list the Vertica schema and the metadata of the three tables respectively. Note that, the text data include both short text, such as tokens and dictionary names, and relatively long text, such as the sentences (the longest sentence is about 23000 characters). Using these different varieties of text data, we can study how MapReduce and parallel DBMSs handle text more comprehensively.

To study scaling-up factors, we duplicated both sentences and tokens and increased their sizes to 2, 4, 8 and 16 times of the original tables. We denote the sentence (token) table which is N times of the original one as sentencesNX (tokensNX). We did not increase the size of dictionary, since typically the size of document corpora may increase while the size of dictionaries often remains the same.

	Vertica			Hadoop/Pig
Table	Segment Attributes	Segment Function	Replication	Replication
sentences	doc_id, sent_id	hash	none	3
tokens	doc_id, sent_id, token_id	hash	none	3
dictionary	name_id	hash	none	3

Figure 4: Data layout of sentences, tokens and dictionary.

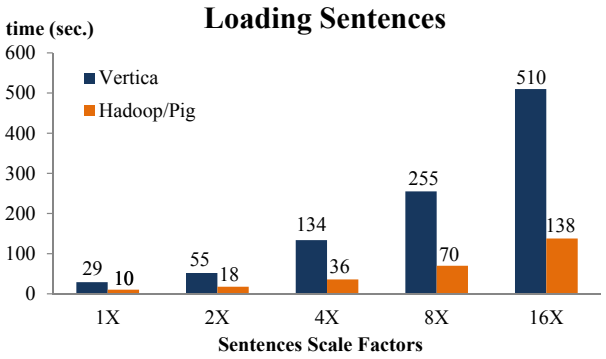

Figure 5: Loading sentences at 5 scale factors.

Data Layout: Figure 4 lists the data layout in Vertica and Hadoop HDFS. Vertica can either horizontally partition tables or replicate tables. In partitioning tables (or so called "creating segments" in Vertica), users need to specify (1) the attribute(s) on which the segments are created; and (2) functions (i.e. hashing or range) used to create the segments. We chose to segment all the tables on their *primary keys* using *hashing* functions. Furthermore, we did not replicate tables in Vertica. Finally, since Pig does not automatically generate query plans which use indices yet, for the fairness of comparison we did not create indices in Vertica either.

In Hadoop HDFS, we cannot explicitly specify the segment attributions and functions as we did in Vertica. Instead, Hadoop HDFS horizontally segments the files into blocks, creates replications for each block, and then randomly distributes all blocks among multiple data nodes.

4.3 Data Loading

Vertica: We used a `copy` command provided by Vertica which loads a file from the file system into the DBMS. This `copy` command is issued from a single node and coordinates the loading process among multiple nodes. Specifically, Vertica creates a new tuple for each line in the input file, and distributes the tuple to one of the nodes according to the segment attributes and functions defined together with the table schema.

Hadoop: In Hadoop, we used the same input files as we used to load tables into Vertica. Then we used a `copyFrom Local` Hadoop command to load the files from local files systems to HDFS.

Results: Figure 5 and 6 illustrate the times of loading sentences and tokens respectively. On each figure, we contrasted the time of loading the same file in Vertica with that in Hadoop. Furthermore, we scaled up both tables and

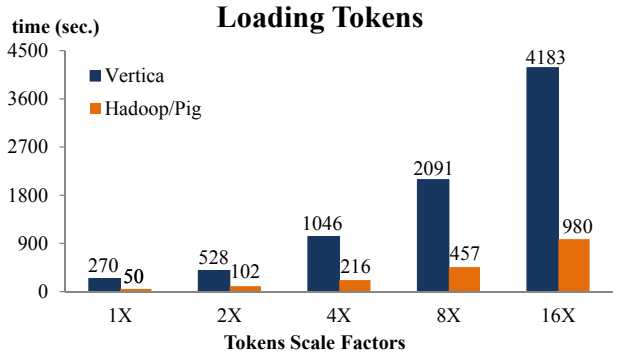

Figure 6: Loading tokens at 5 scale factors.

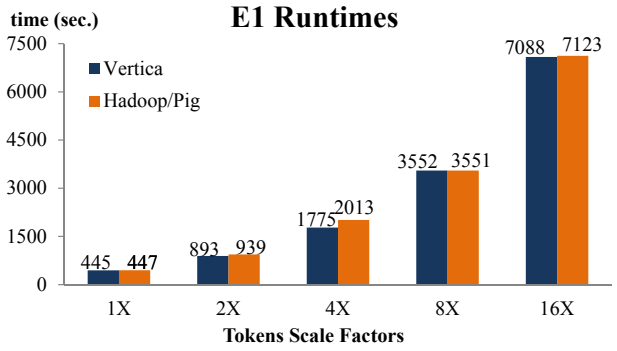

Figure 7: $E1$ execution time at 5 scale factors.

recorded the loading times.

We have the following observations. First, Vertica spent far more time loading both tables on all scale factors than Hadoop. Overall, the sentencesNX loading times of Vertica is 2-3 times larger than that of Hadoop, and the tokensNX loading time of Vertica is 3-4 times larger than that of Hadoop. The overhead of loading in Vertica is mainly caused by parsing files according to the schema and compressing data. However, as we will show later, the query execution performance gains of Vertica offset such upfront loading costs.

Furthermore, we observed when the data sizes were doubled, the loading times were also roughly doubled for both Vertica and Hadoop. This suggests that both Vertica and Hadoop scaled well in terms of loading large text data.

4.4 IE Tasks Using Simple IE Workflows

4.4.1 CRF Based Named Entity Extraction (E1)

The first IE task is to identify *named entities* from the Wikipedia articles using a CRF model. This CRF model takes as input the sequence of tokens (and their associated POS tags) within a single sentence from tokens, and outputs a named entity tag, for each token in the sequence, as either People (P), Organization (R), Location (L), or Other (O).

We chose to implement the CRF based named entity extractor in C++ for Vertica UDFs and in Java for Hadoop/Pig jobs, because these two languages are either the only or the main language supported by the two platforms. For both languages, we used the CRF APIs provided by popular CRF open sources [5, 6].

The CRF model was first trained using the stand-alone versions from the above packages. We now discuss the implementations of $E1$ in Vertica and Hadoop/Pig.

Vertica: The implementation of $E1$ in Vertica consists of two parts: (1) the implementation of a CRF UDF which takes a sequence of tokens (and their POS tags) within a sentence as input and generates named entity tags for the input tokens, and (2) the implementation of a SQL query which applies the CRF UDF to the entire tokens table.

We implemented the CRF UDF as a *transform* UDF by instantiating the Vertica UDF interfaces as follows. In the setup function, the CRF model is loaded into memory. The processPartition is the main body of UDF, where we parse the set of input tuples, construct an in-memory data structure storing the sequence of input tokens and their POS tags, apply the CRF model, and output a set of tuples containing the named entity tags produced by the CRF model. Finally, we release the memory consumed by the CRF model in the destroy function.

The SQL query of applying the CRF UDF to the tokens table is listed below.

```
SELECT did, sid, CRF(token,pos)
       OVER (PARTITION BY did, sid
             ORDER BY tid)
FROM tokens;
```

The query first partitions the tokens table by the *did* and *sid* columns (i.e. grouping the tokens within the same sentences together). Then it sorts the tuples within each partition by *tid*. Finally, the CRF UDF, which takes input as the attributes *token* and *pos*, is applied to each sorted token sequence.

Hadoop: We implemented $E1$ using a single MapReduce job. The Mapper reads in the input file, the tokens table, parses each row r of the input file and identifies *did* and *sid* attributes within r. Then it emits a $(key, value)$ tuple for each r, where *key* is *did* and *sid* and the value is the rest of the content in r. Note that using *did* and *sid* as the key has the same effect as the `PARTITION BY` did, sid SQL expression.

Like Vertica UDFs, Reducers also have a *setup* and *cleanup* method. Similarly, we load the CRF model into memory in *setup* and release the memory in the *cleanup*. The main body of the Reducer is very similar to the main body of Vertica CRF UDF. The only difference is that in order to apply the CRF model to tokens in the order of their positions in the sentence, we need to implement what the `ORDER BY` tid SQL expression does in the Reducer.

It is important to note that our Vertica implementation and Hadoop/Pig implementation take the same input files (the *token* table) and output the same files which contain one tuple (row) for each token with *did*, *sid*, *tid* and the named entity tag of that token.

Results: Figure 7 plots the runtimes of the Vertica and Hadoop/Pig implementations of applying CRFs to tokensNX for $N = 1, 2, 4, 8$ and 16. The most interesting observation is that Vertica's runtimes were comparable to those of Hadoop/Pig in spite of the popular impression that Hadoop /Pig is more suitable for analyzing large scale text data. Furthermore, both Vertica and Hadoop/Pig scaled well. This indicates that both Vertica and Hadoop/Pig are equally reasonable options of applying CRFs on large scale data in terms of runtimes.

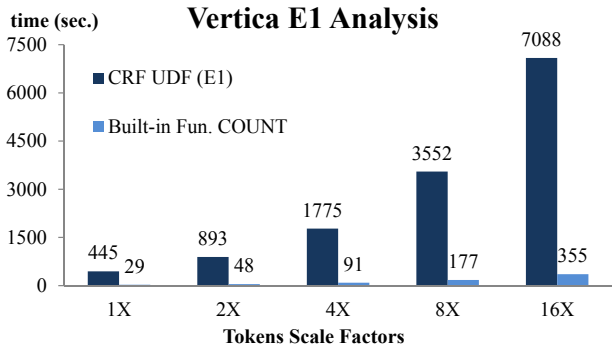

Figure 8: Vertica $E1$ analysis.

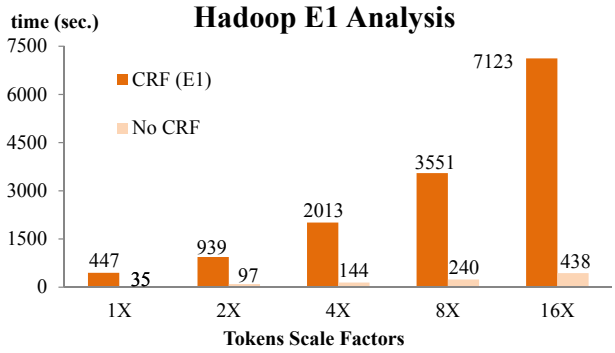

Figure 9: Hadoop $E1$ analysis.

To understand how Vertica CRF UDF performed, we modified the query by replacing CRF UDF, a transform UDF, with a built-in aggregation function `COUNT`. The query is listed below:

```
SELECT did, sid, tid,
       COUNT(*) OVER (PARTITION BY did, sid
                      ORDER BY tid)
FROM tokens;
```

For this modified query, we made sure that it read in the same table as the original query, and size of the table generated by this query was comparable to the size of the table generated by the original query.

Figure 8 plots the runtimes of the 2 SQL queries. First, the runtimes of the query with CRF UDF were about 14 to 19 times larger than the runtimes of the query with `COUNT`. Given that the input table and output table sizes of the two queries were comparable and their query plans picked by the optimizer were similar, the dramatic difference in runtimes suggests that CRF UDF incurred significant overheads, occupying about 94-95% of entire runtimes. This underscores the significance of UDFs in efficiently running statistical learning based extractors in parallel DBMSs like Vertica.

Similarly, to understand how CRF performed on Hadoop/Pig platform, we modified the original MapReduce job so that it *did not* perform any actual CRF work but partitioned tokens and sorted the tuples as the original MapReduce job did. Essentially we kept the I/O and communication costs about the same as the original MapReduce job.

Figure 9 plots the runtimes of the two MapReduce jobs. We observed that the MapReduce job without the CRF related code only took about 6-8% of the original MapReduce job runtimes. This suggests that running statistical learning based extractors also incurred significant overheads on Hadoop/Pig platform.

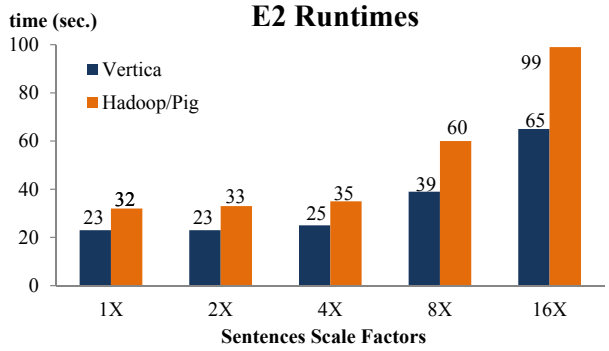

Figure 10: $E2$ execution time at 5 scale factors.

4.4.2 *Regular Expression Based Date Extraction (E2)*

The second IE task is to use a regular expression to extract date from the sentences.

Vertica: Vertica provides function `REGEXP_LIKE` to determine if a string matches a pattern, and function `REGEXP_SUBSTR` to extract a substring within a string that matches a pattern. The SQL for $E2$ is as follows[1]:

```
SELECT did, sid,
       REGEXP_SUBSTR(sentence,
                     DATE_REGEXP, 'i')
FROM   sentences
WHERE  REGEXP_LIKE(sentence,
                   DATE_REGEXP, 'i');
```

Pig: Pig also supports a set of built-in regular expression functions similar to those used in Java. We sketch the Pig implementation of the above SQL query as follows. First, we load data from the sentences by the `LOAD` function. We specify the sentences schema as the parameter of `LOAD` to produce the data conformed to the schema. Next, we go through all tuples and select those tuples where the sentences match the date regular expression. This is achieved by the `FILTER` function with the filtering condition specified by the regular expression matching operator `MATCHES`. Finally, we project on the filtered tuples to output the *did*, *sid* and the matched date substring using the `REGEX_EXTRACT` function.

Results: Figure 10 plots the runtimes of Vertica and Hadoop /Pig for $E2$. We observed that Vertica was consistently 40-52% faster than Hadoop/Pig over all scale factors. Furthermore, both platforms scaled well.

To understand why Vertica performed better than Hadoop /Pig on $E2$, we did the following analysis. We first removed `REGEXP_SUBSTR` and only kept `REGEXP_LIKE` in the original SQL query. This results in the following SQL.

```
SELECT did, sid
```

[1]We shorten the date regular expression used in our experiments, '(january|february|march|april|may|june|july|august|september|october |november|december)(\s+\d?\d\s*,?)?\s*\d{4}' as *DATE_REGEXP* in the following discussions.

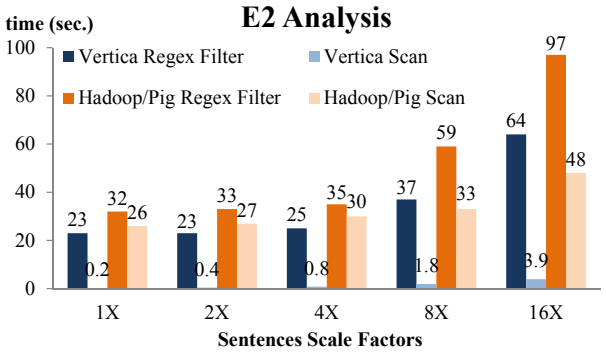

Figure 11: $E2$ **runtime analysis.**

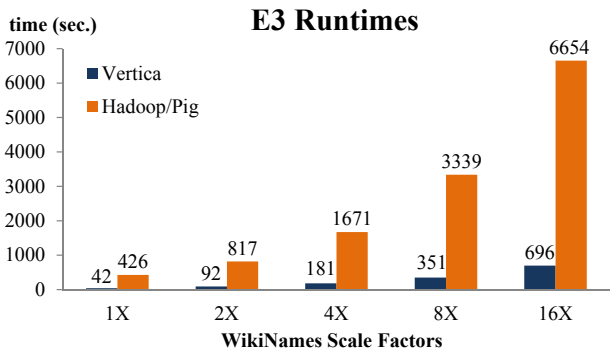

Figure 12: $E3$ **execution time at 5 scale factors.**

```
FROM sentences
WHERE REGEXP_LIKE(sentence,
                  DATE_REGEXP, 'i');
```

Next, we further removed `REGEXP_LIKE` and this results in the following SQL.

```
SELECT did, sid
FROM sentences;
```

We modified the original Pig script in the same way. Then we ran the modified 2 SQL queries and Pig scripts and compared their performance. Figure 11 illustrates their runtimes. First, we checked the query plans of the modified SQL queries and Pig scripts generated by Vertica and Hadoop/Pig optimizer respectively. We found that the modified SQL query (Pig script) plans were comparable with the query plans of the original SQL query (Pig script) in that (1) they accessed data in the same way, and (2) the shared operators between the modified SQL queries (Pig scripts) and the original one were applied in the same order.

Then we had the following observations. First, we observed that the runtimes of SQL queries and Hadoop/Pig scripts involving only the regular expression filter were almost the same as those of the original SQL query and those of the original Pig script respectively. This suggests that extracting the date substrings from the filtered sentences only occupied a negligible portion in the total runtimes of the original query/script.

Second, comparing the SQL query of filtering sentences table using the regular expression and the SQL query of scanning sentences table, we observed that the runtimes of former query were 16-115 times of that of scanning table, indicating that the operator `REGEXP_LIKE` dominated the runtimes. However, as the data size increased, the overheads of the regular expression matching dramatically decreased from 115 times of the runtimes of scanning table at $1X$ scale factor to 16 times of that of scanning table at $16X$ scale factor.

Comparing the Pig script of filtering sentences table using the regular expression and the Pig script of scanning sentences table, we also observed that the runtimes of the the former script were 1.3-2 times of that of scanning table. This indicates that although the runtimes of regular expression matching operator `MATCHES` occupied 19-51% of the runtimes of the regular expression filtering script, its runtime did not dominate the overall runtime as its Vertica counterpart did. Furthermore, in contrast to the Vertica regular expression matching operator, whose overheads relative to the overall runtimes *decreased* as data sizes increased, the overheads of Pig regular expression matching operator relative to the overall runtimes *increased* as data sizes increased.

Finally, we observed that while regular expression matching was faster in Pig than that in Vertica (about 11-17 seconds faster), scanning tables in Pig was much *slower* than that in Vertica (about 25-41 seconds slower). It was mainly the difference in table scanning time that caused the difference in the overall runtimes of the original Pig script and SQL query for $E2$.

4.4.3 *Dictionary Matching Based Entity Reconciliation (E3)*

The third IE task is to reconcile entity names based on dictionary matching. Specifically, we first obtained a set of entity names based on the CRF output, i.e. the output from $E1$. This results in 2.7 millions entity names extracted by CRF from the Wikipedia corpus. Next, we matched these extracted entity names with the Freebase dictionary dictionary using string edit distance.

Because matching 2.7 million entity names with a dictionary containing 453 thousand names in a straightforward way, which invokes 1.2×10^{12} string-to-string comparisons, is very time-consuming, in the experiments discussed below we randomly selected 2000 from 2.7 million entity names, and matched them with 10% dictionary names randomly selected from dictionary. We denote the table containing 2000 names extracted from Wikipedia as wikiNames and the small sample of dictionary table as smallDictionary.

wikiNames has 4 attributes: *did* and *sid* which indicate from which document and sentence the name is extracted, *nid* which indicates the name ID and *name* which is the name string. smallDictionary has the same schema as dictionary. To study the scalability, we replicated and increased wikiNames to $2, 4, 8, 16$ times of the original table as we did before. In section 4.5.2, we will discuss how to efficiently conduct dictionary matching over larger datasets.

Vertica: The Vertica implementation includes two parts: implementing the edit distance UDF and writing the query which invokes the UDF. The edit distance UDF takes input as two strings, and outputs a score indicating the edit distance between the two strings. In contrast to the CRF UDF discussed in Section 4.4.1, this UDF is a *scalar* UDF. The main algorithm of edit distance computation is in the processBlock function.

The SQL query is listed below, where it invokes the edit distance UDF over all pairs of names resulting from the cross

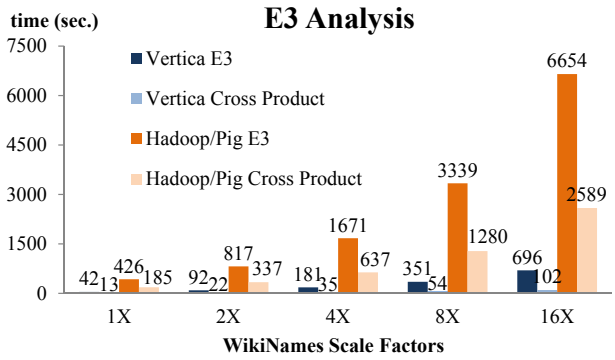

Figure 13: $E3$ vs cross product times.

product of smallDictionary and wikiNames.

```
SELECT D.name, N.name,
       EditDistance(D.name,N.name)
FROM   smallDictionary D, wikiNames N;
```

Pig: Similar to the Vertica implementation, the Pig implementation also includes two parts: implementing a Pig UDF and writing the Pig script. The Pig UDF is a separate java file, where the main algorithm of edit distance was implemented in a function called `exec`.

We sketch the Pig script as follows. It first loads the data from smallDictionary and wikiNames separately. Then it conducts a cross product between the smallDictionary data and wikiNames data. Finally, it works on columns of the cross product results, including projecting the columns to be output and applying the edit distance UDF.

Results: Figure 12 plots the runtimes. We observed that Vertica was significantly (about 8-9 times) faster than Pig. Both platforms scaled well as data size increased. This result again underscores the efficiency advantage of Vertica over Pig on dictionary matching based extraction tasks.

To understand why Vertica performed better than Pig, we removed the edit distance UDF from the original SQL query, resulting in a query purely conducting cross product. The SQL is listed below. It is important to notice the modified query is "comparable" to the original query in terms of their input and output data sizes. We also modified the Pig script in the similar way.

```
SELECT D.name, N.name, 1
FROM   smallDictionary D, wikiNames N;
```

Figure 13 plots the runtimes of the modified queries vs those of the original queries. Again, we checked the query plans of the modified queries generated by Vertica and Hadoop/Pig engines respectively and found that these query plans were comparable to their counterparts of the original queries.

We have the following observations. First, the runtimes of the cross product SQL query were only about 15-31% of the runtimes of the original SQL query. Furthermore, as the data size increased, the ratio of the cross product runtimes to the runtimes of the original query decreased significantly, dropping from 31% at scale factor $1X$ to 15% at scale factor $16X$. This suggests that the edit distance UDF occupied a significant portion of the overall runtime and became more and more dominating in runtimes as data sizes increased.

Second, we have similar observations for the Pig scripts. The runtimes of the cross product Pig script were about 39-43% of the runtimes of the original Pig script. As the data size increased, the ratio of the cross product runtimes to the runtimes of the original script also decreased slightly, dropping from 43% at scale factor $1X$ to 39% at scale factor $16X$. This indicates that the edit distance UDF also occupied a significant portion of the overall runtimes, although it was not as dominating as its Vertica counterpart.

Finally, we observed that the runtime difference between the SQL and Pig cross product queries was significant, increasing from 172 seconds at scale factor $1X$ to 2487 seconds at scale factor $16X$. This difference was about 41-45% of the runtime difference between the original SQL and Pig script. The difference in the edit distance UDF runtimes may contribute the remaining overall runtime difference. This suggests that both efficient relational query processing and efficient UDF execution contributed to the efficiency advantage of Vertica over Pig on $E3$.

4.5 IE Tasks Using Complex IE Workflows

4.5.1 *Multi-join Based Event Extraction (E4)*

As discussed previously (see Section 3.4), many IE workflows for complex IE tasks consist of multi-joins, which "stitch" together the extraction results of subtasks. To study the performance of Vertica and Hadoop/Pig on such workflows, we study the task of extracting events regarding "Apple" company from Wikipedia articles. The goal is to extract an appleEvents table of schema (date, event), indicating on which date what event happened to Apple company.

The extraction rules we used for this tasks are as follows. We first extracted all Wikipedia sentences which mentioned "Apple" company. Then we extracted dates from all sentences. Next, we stitched together an "Apple" company token with a date if they appear in the same sentence. Finally, we output a tuple with a date that appears in the same sentence with "Apple" company and the entire sentence containing this date as the event.

Vertica: We consider two possible SQL implementations, which mainly differ in the way the named entity tags are obtained. The first implementation uses a *materialized* crfTags table of schema (did, sid, tid, tag), indicating the named entity tags of each token in the tokens table. In our experiments, crfTags is the table output by $E1$, i.e. the CRF named entity extractor. The advantages of this implementation are two aspects. First, it is more efficient for repeated extraction tasks based on the named entity tags. For example, there can be tasks which require extracting events of companies other than "Apple" or tasks regarding People entities instead of Company entities. Second, we can easily replace CRF with other types of named entity extractors, e.g., an off-shelf named entity extractors, without changing the event extraction workflow.

The second implementation uses an *in-line* construction of crfTags table. In our experiments, we used the SQL query for $E1$ as a sub-query to compute crfTags online. In contrast to the materialized implementation, the in-line construction is more suitable for one-shot extraction tasks.

The SQL query below lists the materialized SQL implementation. It uses 3 tables: tokens, crfTags, and sentences. Then it first filters 3 tables using the "Apple" on tokens, Company ("R") named entity tags on crfTags and the date regular expression on sentences respectively. Finally, it joins all three tables so that "Apple" and tag "R" are on the same

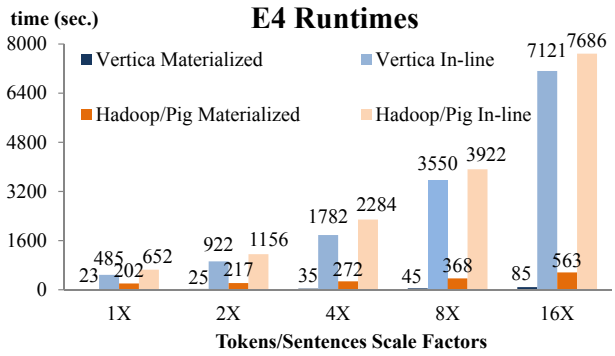

Figure 14: $E4$ execution time at 5 scale factors.

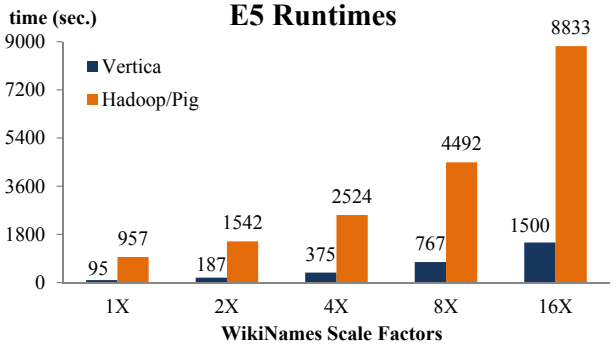

Figure 15: $E5$ execution time at 5 scale factors.

token, and this token is in the same sentence as the sentence containing the date.

```
SELECT S.did, S.sid,
       REGEXP_SUBSTR(S.sentence,
                     DATE_REGEXP, 'i'),
       S.sentence
FROM tokens T, crfTags C, sentences S
WHERE T.token ILIKE 'apple' AND
      C.tag = 'R' AND
      REGEXP_LIKE(S.sentence,
                  DATE_REGEXP, 'i') AND
      T.did = C.did AND C.did = S.did AND
      T.sid = C.sid AND C.sid = S.sid AND
      T.tid = C.tid
ORDER BY S.did,S.sid;
```

The in-line implementation just replaces crfTags with $E1$ SQL as a subquery.

Pig: Like SQL implementations, we also implemented both the materialized version and in-line version of Pig scripts for $E4$. Pig provides a set of operators similar to those provided by Vertica. So we can translate the above SQL implementation using all Pig built-in operators (together with the CRF UDF). However, multi-joins raises a challenge in writing Pig scripts. $E4$ involves joining 3 tables and there are many ways of joining these tables, depending on their join order. Each way results in a different runtime. Unlike a declarative language such as Vertica SQL, a procedure language like PigLatin requires specifying which way the joins are conducted. To address this issue, we tried all combinations of joining 3 tables, and chose the combination resulting in the fastest runtime. Our experiment results below are based on this manually selected "optimal" implementation.

Results: Figure 14 plots the runtimes of two implementations on both platforms. For the scalability experiments, we increased the sizes of 3 tables simultaneously: When we doubled the size of sentences, we also doubled the size of tokens and thus crfTags accordingly.

We first observed that the materialized implementation on Vertica was significantly (6-8 times) faster than that on Pig, indicating Vertica's the efficiency advantage of executing complex workflow involving multiple joins. Furthermore, both Pig and Vertica scaled well as data size increased.

Furthermore, we observed that the in-line implementation on Vertica was still 8-34% faster than that on Pig, although the difference was not as significant as that of the materialized implementation. The main reason is that for the in-line implementation, the CRF UDF runtimes dominated the total runtimes (CRF UDF occupied 90-99% of the total runtime on Vertica, and 69-93% of the total runtime on Pig), and its runtimes were similar on both Vertica and Pig. Furthermore, as data size increased, CRF UDF occupied more and more of the entire runtimes on both Vertica and Pig. This again underscores the significance of UDFs even for complex IE workflows.

4.5.2 *Aggregation Based Efficient Dictionary Matching (E5)*

Finally, we look at an IE task which requires aggregations in addition to multi-joins. Recall that in Section 4.4.3 we describe how to implement dictionary matching in a straightforward way which requires quadratic number of string comparisons. Previous work [17] proposed a more efficient approach which relies on matching short substrings of length n, called *n-grams*, and taking into account both positions of individual matches and the total number of such matches.

Specifically, this approach first computes the n-grams of all names in both dictionary and wikiNames and stores them into auxiliary tables denoted as dicGrams and wikiGrams respectively. The dicGrams table is of schema $(nid, pos, gram)$, where nid is the name ID from dictionary, pos is the position of the n-gram within the name, and $gram$ is the corresponding n-gram. Similarly, wikiGrams is of schema $(did, sid, nid, pos, gram)$ where did, sid and together with nid uniquely associate the n-gram with a name in wikiNames.

Given these n-grams and a threshold of edit distance, the algorithm exploits three conditions to filter out name pairs upon which we will apply the expensive edit distance UDF. These conditions are: if the edit distance between two names is small, they must (1) share a large number of n-grams, (2) the positions of the shared n-grams are not far away, and (3) the lengths of the two names are similar. Please refer to the paper [17] for more rigorous descriptions.

Vertica: The paper [17] gave a SQL query expressing the above 3 conditions as a filter. We only apply the edit distance UDF to those name pairs which pass this filter. The following SQL query is to output all name pairs whose edit distance are within 3[2]:

```
SELECT A.name, B.name
FROM   dictionary A, wikiNames B,
```

[2]We altered the query described in paper [17] to satisfy the ANSI SQL-99 syntax constraint supported by Vertica.

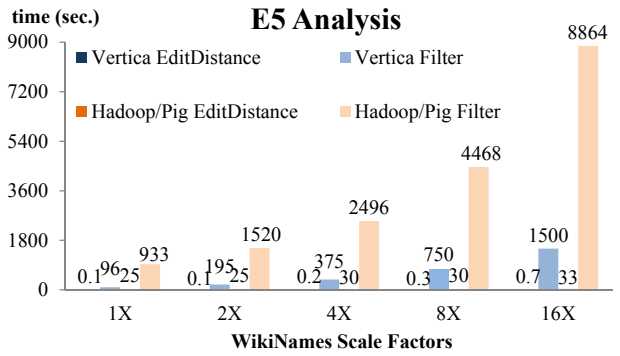

Figure 16: $E5$ execution time analysis.

```
(SELECT W.did, W.sid, W.nid,
        D.nid AS dnid
 FROM dictionary D, dicGrams DG,
      wikiNames W, wikiGrams WG
 WHERE W.did = WG.did AND
       W.sid = WG.sid AND
       W.nid = WG.nid AND
       D.nid = DG.nid AND
       WG.ngram = DG.ngram AND
       (ABS(WG.pos - DG.pos) < 3)
       (ABS (LENGTH(W.name)-LENGTH(D.
           name)) < 3)
 GROUP BY W.did, W.sid, W.nid, D.nid,
          LENGTH(W.name),
          LENGTH(D.name)
 HAVING COUNT(*) >=
        (LENGTH(W.name) - 4) AND
        COUNT(*) >=
        (LENGTH(D.name) - 4)) C
WHERE A.nid = C.dnid AND
      B.did = C.did AND
      B.sid = C.sid AND
      B.nid = C.nid AND
      EditDistance(A.name,B.name)<3;
```

Notice that subquery C expresses the 3 filtering conditions.

Pig: We translate the above SQL query using the built-in operators provided by Pig, including `GROUP` (similar to SQL `GROUP BY`). We followed the same procedure we did in Section 4.5.1 to manually choose the fastest implementation from all possible 4 table join combinations.

Results: Figure 15 plots the runtimes of $E5$ on Vertica and Hadoop/Pig. In this set of experiments, we used the entire dictionary table instead of the smaller smallDictionary used in Section 4.4.3. We observed that the implementation on Vertica was significantly (5-9 times) faster than that on Pig. This again underscores the efficiency advantage of Vertica over Pig on complex IE workflows. Both Vertica and Hadoop scaled well, although Pig had initial overheads not fully amortized at small scale factors and the run times only began to rise at an expected rate after scale factor 4.

To further understand why Vertica performed so well, we decomposed the execution time into the time used on the filtering subquery (subquery C in the above SQL and its counterpart in Pig script) and the remaining part(mainly the edit distance UDF). Figure 16 plots the decomposed times. We observed that the filtering query dominated the total runtime on both platforms (occupying at least 99% of total runtime on Vertica and 98% on Pig). They largely contributed to the difference in total runtime between the platforms. This suggests that although UDFs are important for IE workflows, relational query operators and query flows are as important as UDFs for complex IE workflows.

It is important to note that although the filtering subquery dominated the total runtime, without the filtering, it could have taken much longer to compare wikiNames against the entire dictionary table on both platforms in a straightforward way as we did in $E3$. Please refer to [17] for more details.

4.6 Summary and Discussions

We now summarize the benchmark results, comment on particular aspects of each system that the raw numbers may not convey, and present key conclusions of our studies.

Importance of UDFs: As we have shown in several simple and complex IE tasks ($E1$, $E3$ and $E4$), UDFs dominated the total runtimes on both Vertica and Hadoop/Pig. Therefore, it is important to optimize the execution of UDFs on both engines in two aspects.

The first direction is to make the optimizers of both DBMSs and Hadoop/Pig aware of UDFs. The current optimizers of both types of platforms make little efforts in understanding UDFs, including their selectivity and costs. Understanding UDFs, however, can make a big difference in runtime. For example, the execution plans generated by both Vertica and Hadoop/Pig for the in-line implementation of $E4$ applied the CRF UDFs to the entire tokens table. However, if the optimizers had known that CRF UDF is very expensive, it could first filter tokens table using "Apple". Then it could only apply CRFs to tokens in those sentences which contain token "Apple".

A possible solution along this direction is to understand certain properties of the UDFs and exploit these properties for query optimization. There are some recent works [33, 26, 28, 8] in this direction, but there is much more potential. Another possible solution is to develop tools which can semi-automatically collect UDF statistics and seek users' help in understanding UDFs.

The second direction is to make the execution engines better support running UDFs. In all of our experiments, we ran UDFs as "fenced-out" mode (e.g. running UDFs as a separate process from the query process) on Vertica, which is a safer but less efficient approach. We observed that for some UDFs on Vertica, in particular, UDFs which generated large size of output, the performance was improved by 20% if we switched to "fenced-in" mode. How to achieve the tradeoff between the efficiency and the safety of running UDFs is an important research direction.

Furthermore, the current execution engines, in particular the parallel DBMSs, are designed for I/O intensive tasks. However, as we have witnessed in our experiments and others' works [33, 28, 22, 7], many IE tasks, in particular, are also CPU-intensive tasks. Therefore, it is important to optimize CPU utilization to achieve better performance in executing UDFs.

Importance of Built-in Extraction Operators: Both Vertica and Hadoop/Pig provide built-in extraction operators such as regular expression matchers. Our benchmark showed that on both Vertica and Hadoop/Pig, regular ex-

pression matchers occupied a significant portion of total runtime for some IE tasks (more than 90% on Vertica and as much as 50% on Hadoop/Pig). There have been several works [10, 31] on efficiently matching regular expressions. Both DBMSs and Hadoop/Pig can consider incorporating these advanced techniques for regular expression matcher.

Parallel DBMSs as a Viable Alternative for Large Scale IE: One of the most important lessons we learnt from the benchmark is that parallel DBMSs is a viable alternative for large scale IE to Hadoop in several aspects. First, as we have shown that DBMSs like Vertica provide many built-in extraction related operators such as regular expression functions. This not only makes it easier for users to write IE programs without coding from scratch, but also enables more efficient execution (as shown in task $E2$ and $E4$).

Second, several DBMSs including Vertica use MPP architecture. This feature together with UDFs make it as easy to parallelize applications on DBMSs as that on Hadoop. In terms of performance, as we have observed from our benchmark results, for IE workflows which were dominated by expensive UDFs such as CRFs (e.g. $E1$ and in-line $E4$), Vertica ran at least as fast as Hadoop/Pig, while for other IE workflows (e.g., $E3$ and $E5$), Vertica outperformed Hadoop/Pig by 5-9 times.

Finally, many workflows for complex IE programs consist of a significant number of relational operators used to stitch together workflows for sub-tasks. This is where parallel DBMSs really shine. As our experiment results have shown, for such complex IE workflows (e.g. materialized $E4$ and $E5$), Vertica were significantly (5-9 times) faster than Hadoop
/Pig. We cover more details about this aspects in the following paragraph.

General Performance Issues in DBMSs and Hadoop /Pig: Besides issues specific to IE, we also observed a few performance issues which appeared in previous performance studies [23, 16] on relational queries in DBMSs and Hadoop. First, loading times in DBMSs were slower than those in Hadoop. We showed that loading data in Vertica was about 3-5 times slower than that in Hadoop. This is mainly caused by the upfront overheads of parsing files and compression. So Hadoop/Pig may be more suitable for one-shot analytics tasks while DBMSs may be more suitable for repeated analytics over the same data.

Second, joins in DBMSs were significantly faster than those in Hadoop/Pig. Unlike the streamlining execution of multiple joins in DBMSs, Hadoop/Pig must materialize the results for each join before it begins the next one. This turns out to be great overheads (paid by Hadoop/Pig for fault tolerance). These observations suggest that optimizing workflows consisting of relational operators is also important for large scale IE tasks.

5. CONCLUSIONS AND FUTURE WORK

We propose a benchmark to systematically study the performance of parallel DBMSs and Hadoop for large scale IE tasks. Our results show that parallel DBMSs is a viable alternative for large scale IE. The future works include (1) extending the studies to other high level languages over Hadoop such as Hive; and (2) leveraging our benchmark results to categorize IE workflows and build hybrid execution engines on DBMSs and Hadoop for large scale IE.

6. REFERENCES

[1] `http://trec.nist.gov/`.
[2] `http://dumps.wikimedia.org/enwiki/20120307/`.
[3] `http://cogcomp.cs.illinois.edu/page/software/`.
[4] `http://www.freebase.com/`.
[5] `http://crfpp.sourceforge.net/`.
[6] `http://crf.sourceforge.net/`.
[7] A. Alexandrov, M. Heimel, V. Markl, D. Battré, F. Hueske, E. Nijkamp, S. Ewen, O. Kao, and D. Warneke. Massively parallel data analysis with pacts on nephele. *PVLDB-10*.
[8] F. Chen, X. Feng, C. Ré, and M. Wang. Optimizing statistical information extraction programs over evolving text. *ICDE-12*.
[9] F. Chen, B. Gao, A. Doan, J. Yang, and R. Ramakrishnan. Optimizing complex extraction programs over evolving text data. *SIGMOD-09*.
[10] J. Cho and S. Rajagopalan. A fast regular expression indexing engine. *ICDE-02*.
[11] X. Dong, A. Halevy, and J. Madhavan. Reference reconciliation in complex information spaces. *SIGMOD-05*.
[12] V. Ercegovac, D. DeWitt, and R. Ramakrishnan. The texture benchmark: measuring performance of text queries on a relational DBMS. In *VLDB-05*.
[13] X. Feng, A. Kumar, B. Recht, and C. Ré. Towards a unified architecture for in-RDBMS analytics. *SIGMOD-12*.
[14] D. Ferrucci and A. Lally. UIMA: An architectural approach to unstructured information processing in the corporate research environment. *Nat. Lang. Eng.*, 10(3-4), 2004.
[15] J. R. Finkel, T. Grenager, and C. Manning. Incorporating non-local information into information extraction systems by gibbs sampling. *ACL-05*.
[16] A. Floratou, N. Teletia, D. DeWitt, J. Patel, and D. Zhang. Can the elephants handle the NoSQL onslaught? *PVLDB-12*.
[17] L. Gravano, P. Ipeirotis, H. Jagadish, N. Koudas, S. Muthukrishnan, D. Srivastava, et al. Approximate string joins in a database (almost) for free. *VLDB-01*.
[18] J. Hellerstein, C. Ré, F. Schoppmann, D. Wang, E. Fratkin, A. Gorajek, K. Ng, C. Welton, X. Feng, K. Li, et al. The MADlib analytics library, or MAD skills, the SQL. *PVLDB-12*.
[19] G. Kasneci, M. Ramanath, F. Suchanek, and G. Weikum. The YAGO-NAGA approach to knowledge discovery. *SIGMOD Record*, 37(4), 2008.
[20] J. Lin and C. Dyer. Data-intensive text processing with mapreduce. *Syn. Lec. on Human Lang. Tech.-10*.
[21] A. McCallum and W. Li. Early results for named entity recognition with conditional random fields, feature induction and web-enhanced lexicons. *CoNLL-03*.
[22] F. Niu, C. Ré, A. Doan, and J. Shavlik. Tuffy: Scaling up statistical inference in markov logic networks using an RDBMS. *PVLDB-11*.
[23] A. Pavlo, E. Paulson, A. Rasin, D. Abadi, D. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD-09*.

[24] F. Peng and A. McCallum. Accurate information extraction from research papers using conditional random fields. In *HLT-NAACL-04*.

[25] D. Pinto, A. McCallum, X. Wei, and W. B. Croft. Table extraction using conditional random fields. *SIGIR-03*.

[26] F. Reiss, S. Raghavan, R. Krishnamurthy, H. Zhu, and S. Vaithyanathan. An algebraic approach to rule-based information extraction. *ICDE-08*.

[27] S. Sarawagi. Information extraction. *Foundations and Trends in Databases*, 1(3):261–377, 2008.

[28] W. Shen, A. Doan, J. Naughton, and R. Ramakrishnan. Declarative information extraction using datalog with embedded extraction predicates. *VLDB-07*.

[29] A. Simitsis, K. Wilkinson, M. Castellanos, and U. Dayal. Optimizing analytic data flows for multiple execution engines. *SIGMOD-12*.

[30] A. Thusoo, J. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy. Hive-a petabyte scale data warehouse using hadoop. *ICDE-10*.

[31] D. Tsang and S. Chawla. A robust index for regular expression queries. *CIKM-11*.

[32] R. Vernica, M. Carey, and C. Li. Efficient parallel set-similarity joins using MapReduce. *SIGMOD-10*.

[33] D. Wang, M. Franklin, M. Garofalakis, and J. Hellerstein. Querying probabilistic information extraction. *PVLDB-10*.

[34] D. Wang, M. Franklin, M. Garofalakis, J. Hellerstein, and M. Wick. Hybrid in-database inference for declarative information extraction. *SIGMOD-11*.

[35] G. Weikum, J. Hoffart, N. Nakashole, M. Spaniol, F. Suchanek, and M. Yosef. Big data methods for computational linguistics. *IEEE Data Eng. Bulletin*, 35(3), 2012.

[36] F. Wu, R. Hoffmann, and D. S. Weld. Information extraction from Wikipedia: moving down the long tail. *SIGKDD-08*.