Efficient and Self-Balanced ROLLUP Aggregates for Large-Scale Data Summarization

Duy-Hung Phan
EURECOM
phan@eurecom.fr

Quang-Nhat Hoang-Xuan
EURECOM
hoang@eurecom.fr

Matteo Dell'Amico*
EURECOM
della@linux.it

Pietro Michiardi
EURECOM
michiard@eurecom.fr

***Abstract*—Data summarization queries that compute aggregates by grouping datasets across several dimensions are essential to help users make sense of very large datasets. In this work, we focus on ROLLUP, an important operator that has been recently added to the Hadoop MapReduce ecosystem. However, its current implementation suffers from very large communication costs, leading to inefficient executions. We thus proceed with the design of a new ROLLUP operator for high-level languages. Our operator is self-optimizing, which means that it automatically performs load-balancing and determines a suitable operating point to achieve the highest performance. We have implemented our ROLLUP operator for Apache Pig, a popular high-level language in the Hadoop ecosystem. Our experimental results, obtained on both synthetic and real datasets, indicate that our new operator outperforms the current ROLLUP implementation in Pig by at least 50%.**

I. Introduction

Users that interact with "*big data*" constantly face the problem of extracting insight and obtain value from their data assets. Of course, humans can not be expected to parse through terabytes of data. In fact, typical user interaction with big data happens through *data summaries*. A summary is obtained by grouping data along various dimensions (*e.g.*, by location and/or time), and then showing aggregate functions of those data (*e.g.*, count, sum, mean, *etc.*). Even graphical and interactive visualizations of data very often show aggregated results.

On-Line Analytical Processing (OLAP) tools and techniques exist to facilitate exploration of data, allowing to perform "slicing and dicing" [1] by grouping data along multiple dimensions. In relational databases, this is facilitated by extending the traditional SQL GROUP BY clause with constructs such as ROLLUP, CUBE, and GROUPING SETS. These operations have seen extensive efforts to optimize their implementation in relational databases [2]–[4], which only apply to single servers or small clusters.

Despite the importance of data summarization, the field of data-intensive scalable computing (DISC) systems – where data can reach petabytes and be distributed on clusters of thousands of machines – has not seen much effort toward efficient implementations of the above concepts. In the Hadoop MapReduce ecosystem [5], [6], high-level query languages such as Pig Latin [7] and HiveQL [8] offer simple implementations of the above constructs, which do not perform aggressive optimizations. In enterprise workloads, jobs coming from queries written in high-level languages are the majority [9]; an optimized implementation of these operators is therefore truly desirable.

In this work, we focus on the design and implementation of a new ROLLUP operator for high-level languages. The motivations for focusing on ROLLUP are two: *(i)* it is very useful in frequent cases where data dimensions are naturally hierarchical (*e.g.*, day-month-year or city-region-country); *(ii)* it is used as a fundamental building block to compute CUBE and GROUPING SETS [10].

Existing implementations are not satisfying: as we discuss in Section II, current ROLLUP algorithms are naively biased toward extreme levels of parallelism. As a consequence, these approaches trade a theoretical possibility of scaling several orders of magnitude beyond the scale attainable by real-world clusters with a very significant overhead in terms of communications.

There are alternative ROLLUP algorithms that allow tuning the level of parallelism and the communication overhead of their implementation: with a proper setting, these algorithms perform better than naive implementations. Such approaches are appealing in the abstract, but they are practically very difficult for users to apply, since to determine the proper setting, they would require users to know: *(i)* the internals of the algorithm's implementation; *(ii)* statistical information about data distribution; *(iii)* size and performances of the cluster on which the algorithm is implemented. These requirements are difficult to meet at once, and they are essentially prohibitive in the context of high-level DISC languages, whose very reason of existence is to hide this kind of complexities and allowing users to concentrate on extracting meaning from data.

As a result, in this paper we design (in Section III) and implement a new ROLLUP operator that is completely transparent to users. Our operator automatically collects statistics about data and cluster performance. Subsequently it uses this information to: *(i)* balance load across different nodes in the cluster; *(ii)* determine an appropriate operation point of ROLLUP algorithms using a lightweight cost-based optimizer.

We perform an extensive experimental campaign (Section IV) to assess the validity of our design choices, using both real and synthetic datasets, and comparing the performance of a variety of different ROLLUP algorithms. Results indicate that our ROLLUP operator, that relies on automatically tuned algorithms, delivers superior performance when compared to current ROLLUP implementations for the Apache Pig system. Our operator is released as open source software[1], and is currently in the process of being integrated in the upstream Apache Pig Latin language implementation.

*The author is currently working in Symantec Research Labs

[1]https://bitbucket.org/bigfootproject/rollupmr

978-1-4673-7278-7/15 $31.00 © 2015 IEEE
DOI 10.1109/BigDataCongress.2015.31

IEEE computer society

We conclude our work in Section V with a summary of our future research directions.

II. Preliminaries & Related Work

Efficient computation of data summaries is an important topic that has received wide attention by the database community. Recently, such interest has led to several works to bring the benefits of data summarization to MapReduce systems as well.

In this Section, we review several ROLLUP algorithms, for both traditional databases and MapReduce systems, and motivate the need for a substantially different approach. ROLLUP is a data operator first introduced by Gray *et al.* [1] as a special case of CUBE. The ROLLUP operator aggregates data along a dimension by "climbing up" the dimension hierarchy. For example, to aggregate the volume of cars sold in the last several years, it is possible to use the ROLLUP operator to obtain sales along the time dimension, including multiple levels: `total`, `year`, `month`, and `day`. These *levels* form a hierarchy, of which `total` is the top level that includes aggregates from all records.

A. ROLLUP in Parallel Databases

Traditionally, the ROLLUP operator was studied through the lenses of its generalized operator, CUBE, that groups data along all combinations (called *views*) of different levels in hierarchies. In ROLLUP, a view is equivalent to a level of the rollup hierarchy. In our car sales example, we have `4` different views.

Harinaryan *et al.* [2] introduced a model to evaluate the cost of executing CUBE and a greedy algorithm to select a near-optimal execution plan. Agarwal *et al.* [3] proposed *top-down* algorithms (*PipeSort*, *PipeHash*) to optimize CUBE computation: using finer group-bys to compute coarser ones (*i.e.* using `month` to compute `year`). Beyer and Ramakrishnan [4] suggested a *bottom-up computation* (BUC) to construct results from coarser to finer group-bys by reusing as much as possible the previously computed sort orders. All of these works are sequential algorithms which focus on single servers.

Scalable algorithms to handle the ROLLUP and CUBE operators in parallel databases also received considerable attention. They are divided into two main groups: *work partitioning* [11], [12] and *data partitioning* [13], [14]. In work partitioning, each processor (or node) of the cluster computes aggregates for a set of one or many views independently. To do that, all processors access concurrently the entire dataset. Typically such an access is offered by a shared-disk array that is both expensive and difficult to scale in term of performance and size. Instead, data partitioning algorithms divide the input data set into various subsets. A node computes all views associated to the subsets of data it hosts. To obtain global aggregates, a subsequent *merge* phase is required. The main advantage of such a *Two Phase* algorithm is that nodes need not to have access to the whole data set but work on a small portion that can be easily stored on local memory/disks.

B. ROLLUP in MapReduce

In what follows, we assume the reader to be familiar with the MapReduce paradigm and its well-known open-source implementation Hadoop [5], [15].

Currently, MapReduce high-level languages such as Apache Pig and Hive borrow from parallel databases the Two Phase algorithm described previously. Its MapReduce variant is straightforward: each mapper computes aggregates of the whole ROLLUP hierarchy from its local data, then sends the partial results to reducers which merge and return the final aggregates. The MapReduce Two Phase (MRTP) algorithm, which can use combiners as an optimization, produces a large quantity of intermediate data that impose strain on network resources. In addition, a large number of intermediate data increases noticeably mapper overheads to sort and eventually spill them to disk. Similarly, since reducers compute aggregates in each view separately, the MRTP algorithm presents significant overheads due to redundant computation.

These overheads are the main reason for the MRTP inefficiency, and they can be avoided by employing the *in-reducer grouping* (IRG) design pattern [16]. IRG computes aggregates in a top-down manner by exploiting custom partitioning and sorting in MapReduce to move the grouping logic from the shuffle phase to reducers. However, IRG severely lacks parallelism: all processing is performed by a *single* reducer.

More recent works explore the design space of one-round MapReduce ROLLUP algorithms by studying the trade-off between communication cost (amount of data sent over network) and parallelism [17]. Through a parameter called pivot position, the Hybrid IRG+IRG algorithm (HII) provides users with the flexibility of choosing a sweet spot to balance parallelism and communication cost. For one-round MapReduce algorithms, the HII algorithm is shown (analytically and experimentally) to achieve the best performance, if and only if the pivot position is set to the optimal value. Otherwise, the HII algorithm represent a valid theoretical contribution, albeit impractical to be used as a baseline for a high-level language operator.

For multi-round MapReduce ROLLUP algorithms, the work in [17] also proposes the ChainedIRG algorithm, which splits a ROLLUP operator into two chained MapReduce jobs based on a parameter (again, a pivot position). The pivot position divides the ROLLUP hierarchy between two jobs. The first job computes a subset of the hierarchy and the second job takes the first job's results to compute the rest.

Finally, MRCube, proposed by Nandi *et al.* [18], implements CUBE and ROLLUP operators in three rounds. The first round performs record random sampling on input data and estimates the cardinality of reducer keys. It serves identifying large groups of keys whose cardinality exceeds the capacity of a reducer, and set a partition factor N. In the second round, MRCube splits these large groups into N sub-groups using *value partitioning*: two keys belong to the same sub-groups if and only if their values of the aggregated attribute are congruent modulo N. Each sub-

group is computed as a partial aggregate by some reducer using the BUC algorithm [2]. The third round merges partial aggregates to produce final results. As a consequence, reducers do not handle excessive amounts of data, but the execution plan requires multi-rounds of MapReduce jobs. The authors also note that the value partitioning may be problematic when data is skewed on the aggregated attribute, as it can create sub-groups that exceed the reducer capacity and slow down the job. Also, as the number of large-groups may be high, the first step of MRCube can create a significant overhead and make MRCube slower than the MapReduce Two Phase algorithm [18].

C. Other Related Works

It is well known that MapReduce algorithms may suffer from poor performance if data is skewed. SkewReduce [19] is a framework that manages data and computation skew by using user domain knowledge to break the map and reduce tasks into smaller tasks; then it finds the optimal partition plan to achieve load balancing. SkewTune [20] is a dynamic skew mitigation system: by modifying the MapReduce architecture, it detects stragglers in reducers and pro-actively repartitions the unprocessed data to other idle reducers. Another approach is to design skew-resistant operators: data skew is handled at the algorithmic level. This approach does not require additional components, user interventions or modifications to the Hadoop framework. For instance, [21] proposes an algorithmic approach to handle set-similarity join. Apache Pig also supports a skew-resistant equi-join operator. For the MapReduce data summarization, to the best of our knowledge, our work is the first to tackle a skew-resistant ROLLUP operator.

Finally, we consider related works that use a cost model to find optimal execution plans. SkewReduce [19] uses a cost model that requires two user-supplied cost functions. MRShare [22] proposes work-sharing optimizations based on a cost model to predict merged job's runtime. This model requires users to provide a set of constant, static parameters, that represent the underlying cluster performance. Both practices are not transparent to users. The latter is problematic in practice because cluster performance changes overtime, and in many cases, clusters are dynamically allocated (*e.g.* Amazon EC2, Google App Engine) which means unpredictable performance.

Instead, as we show in the next Section, our approach automatically measures cluster performance. These measurements are fed to a regression model that predicts the ROLLUP runtime. This is completely transparent to users, and adaptable to any cluster configuration.

III. A New ROLLUP Operator

In this Section, we describe our design of an efficient and skew-resistant ROLLUP operator. Our approach can be integrated directly in current MapReduce high-level languages such as Apache Pig and Hive, and it is completely transparent to users. Our design avoids any modifications to the Hadoop framework or the MapReduce programming model, thus making our work directly applicable to any MapReduce-like systems. Our approach can also handle a stateless design (*i.e.* no historical execution statistics), which is the current standard practice for systems that provide high-level languages on top of MapReduce.

A. ROLLUP Operator Design

Our ROLLUP operator has two main components: the *tuning job* and the *ROLLUP query*. The ROLLUP query can implement one of several ROLLUP algorithms, such as MRTP, HII, ChainedIRG, MRCube as described in Section II-B. We discuss the choice of an appropriate algorithm in Section III-B.

The heart of our work is the *tuning job*, a primary component with two main goals. The first is to determine how to achieve load-balancing taking into account skewed data when executing the ROLLUP query, which is clearly beneficial to any ROLLUP algorithm. The second goal is to determine the most suitable operating point of the ROLLUP query, provided that the underlying ROLLUP algorithm requires tuning. For example, ROLLUP algorithms like HII and ChainedIRG both rely on an essential parameter that, if not properly tuned, can lead to inefficient executions and bad performance. In this case, the tuning job automatically sets the parameter of such algorithms to the proper value, such that performance is maximized.

In addition, to obtain an efficient ROLLUP operator, our goal is to minimize the overhead caused by the tuning job. Consequently, we propose a single, light-weight tuning job that simultaneously carries out all the following tasks to produce efficient ROLLUP queries:

1) It produces representative samples of the input data;
2) It balances reducer loads using information on key distribution estimated from sample data;
3) If required, it determines appropriate parameters of a ROLLUP algorithms using a cost-based optimizer.

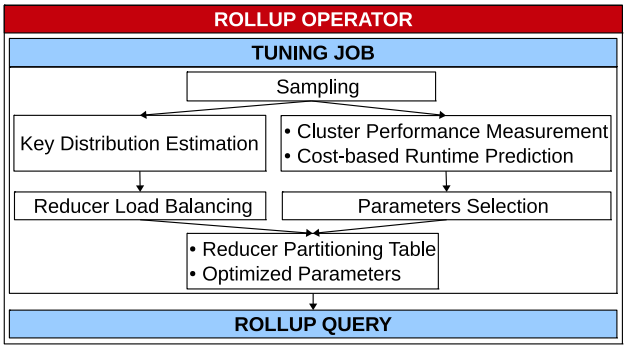

Figure 1. Overview of the ROLLUP operator design.

Figure 1 shows a sketch of our ROLLUP operator. The tuning job runs before the execution of the ROLLUP query. It produces a balanced partitioning table, and tunes parameters appropriately, when needed. The ROLLUP query uses these outcomes to optimize its performance.

In the rest of this Section, we describe in detail how our tuning job can fulfill its goals using a top-down presentation.

First, we discuss our choice for the ROLLUP algorithm that implements the ROLLUP query in Section III-B. Then, we present the internals of our tuning job in Section III-C. We examine load balancing on reducers in Section III-C1, which uses statistics collected through sampling (Section III-C2). Finally, we present our cost model (Section III-C3) that steers the ROLLUP query toward an optimized operating point.

B. The ROLLUP Query: Algorithmic Choice

The design of our ROLLUP operator is generic enough to employ any ROLLUP algorithm. However, in this work, we focus on the two algorithms that proved to offer, consistently, superior performance when compared to alternative approaches, namely the Hybrid IRG+IRG (HII) and ChainedIRG [17]. Indeed, unlike MRTP, MRCube and IRG, such algorithms have the flexibility to adjust the level of parallelism to exploit, which translates into a much lower communication cost compared to MRTP and MRCube. Previous results in the literature [17] corroborate our choice, which we confirm in our extensive performance evaluation in Section IV.

Now, both ChainedIRG and HII require a fundamental parameter P called *pivot position*. Let us consider the ROLLUP dimension with n levels: $\{d_1, d_2, \ldots, d_n\}$ in which d_1 is the top-level of the hierarchy. P divides the ROLLUP dimensions in two subsets: $S_1 = \{d_1, \ldots, d_{P-1}\}$ and $S_2 = \{d_P, \ldots, d_n\}$. Each subset now represents a sub-ROLLUP query. The difference between ChainedIRG and HII is that, while HII computes each sub-ROLLUP query independently and has only one job, ChainedIRG exploits the aggregates on S_2 to compute aggregates on S_1 (hence, it requires two jobs). Nonetheless, both algorithms use the IRG design pattern to compute each sub-ROLLUP query.

Although the experimental results in [17] indicate that ChainedIRG has the best runtime in an isolated system, in this section we cast our ROLLUP operator on the HII algorithm because it is less prone to delays due to scheduling when the cluster is loaded. Instead, the runtime of multi-phase algorithms (such as ChainedIRG) could be inflated since the job scheduler can dedicate resources to other jobs in between the phases. Nonetheless, we note that our tuning job can easily accommodate alternative algorithms, and we show this in our experimental evaluation, where we present results of our ROLLUP operator with instances of the ROLLUP query implementing all the algorithms we discussed in Section II.

Continuing our example in Section II, if $P = 3$ the two subsets are $\{\texttt{total}, \texttt{year}\}$ and $\{\texttt{month}, \texttt{day}\}$. For each input record, the map phase of HII generates 2 $\langle \text{key}, \text{value} \rangle$ pairs of the bottom level of each subset (*i.e.* `year` and `day`).

Then for each subset, the mappers partition their $\langle \text{key}, \text{value} \rangle$ pairs by the top level of this subset (*i.e.* `total` and `month`). Taking advantage of this partitioning scheme and the sorting done by the MapReduce framework, day-to-month and year-to-total aggregates can be processed independently in the reduce phase. In fact, HII applies a top-down approach: we compute aggregates for each day in a month; then combine results to obtain month aggregates. Similarly, we compute results for each year and construct the total aggregate.

Choosing an appropriate pivot position P is crucial in HII and ChainedIRG, because it determines the parallelism as well as the communication cost of such algorithms. Values of P that are too high can impose an excessive load on the reducer that is responsible for the `total` aggregates, and lessen the benefit of combiners (*i.e.* higher communication cost). Instead, low values of P could result in insufficient parallelism causing poor load balancing. Finding the proper value for P is a non-trivial problem.

C. The Tuning Job

We now discuss our main contribution, the tuning job, that balances the reducer load and finds the optimal value of P. We propose a novel mechanism that enables a single-round MapReduce job to collect statistical information about the input data and the underlying cluster performance, determine and impose a load balancing scheme and optimize the selection of the pivot position for the HII algorithm. The tuning job operates as follows:

- For each possible pivot position P, we estimate a *key distribution*: a mapping between each partitioning key and the number of records that correspond to that key (*i.e.* its cardinality).
- For each pivot position, we devise a greedy *key partitioning* scheme that balances as much as possible the load between reducers;
- For each pivot position, using performance measurements gathered throughout the tuning job execution, we use a cost model to *predict the runtime* associated to each pivot position. We then select the pivot position that yields the shortest job runtime.

Because the search space for an optimal pivot position is small and capped by n – the number of grouping sets – our mechanism can afford to evaluate all values of P. We minimize the overhead of the tuning job by executing it as a single MapReduce job, where the input data is read only once and all candidate values for P are evaluated in parallel. The rest of this section presents how each step of our tuning job is accomplished. Key partitioning described in Section III-C1, uses the estimation of key distribution using sampling (Section III-C2). From partitioning outputs, we extract the reducer loads to feed into our cost model, as shown in Section III-C3. The cost model uses a regression-based approach that predicts the runtime of both mappers and reducers for each value of P; this model uses performance measurements that we collect in the tuning job as a training set.

1) Balancing Reducer Load: Skewed data motivates the need for reducer load-balancing. In this step, we balance reducer load using cardinalities of partitioning keys obtained from the key distribution estimation discussed in Section III-C2. The input of this step is a set of keys K and an estimation of their respective cardinalities $C = \{C_k | k \in$

$K\}$. We then perform load balancing by partitioning the keys in K on r reducers to minimize the input keys on the most loaded reducer. This problem can be reduced to the multi-way number partitioning problem, which is NP-complete [23]. We thus opt for a greedy solution: sort all reduce input keys by descending cardinalities, and assign at each step a key to the least loaded reducer. Our algorithm uses smaller cardinalities to counter the imbalance created by large cardinalities that are allocated first. Its runtime is linear with respect to the number of keys. Thanks to the role of combiners, the number of input records sent to each reducer is quite small; for this reason, as we shall see in Section IV-B, our load balancing strategy is very effective in practice.

The output of this greedy algorithm is a reducer partitioning table, which we broadcast to all mappers to determine the key-reducer mapping. When the ROLLUP query is running, such a partitioning is kept in a hash table and used to "route" keys to reducers. If, due to sampling, a key does not appear in the hash table, it is "routed" to a pseudo-random reducer using hashing. We note that the impact of such missing keys is minimal on load balancing, as they correspond to infrequent input data.

2) Sampling and Computing Key Distribution: The load balancing step we just discussed requires the number of reducer input records per grouping key, *i.e.* its cardinality. Without modifying the MapReduce architecture, we cannot count these cardinalities on the fly when processing data. Instead, we resort to *sampling*: we only read data from a small subset of the input data, and we gather key distributions and cardinalities that are the input of the load balancing step. These steps are counted as an overhead in our total job runtime.

In MapReduce, uniform record random sampling from the whole input would be inefficient, as it requires the whole data to be read and parsed. Instead, we employ *chunk random sampling*. Chunk sampling allows low overhead, as it reads a very small portion of input data. The dataset is split in chunks of data that have different sizes: this is also useful for the linear regression model we describe in Section III-C5. Each chunk is chosen with probability σ, a parameter we explore in our experimental evaluation.

However, chunk sampling introduces sampling bias. To reduce this bias, for each chunk, we output each record only once, regardless of its multiple appearances in that chunk. Such a technique can be easily achieved by using combiners. In the reducers, we collect records from different chunks and treat them like records gathered from uniform random sampling. This method, albeit in a different context, is described in the literature as the COLLAPSE approach in [24]. For large input data, the COLLAPSE approach is proved to be approximately as good as uniform random sampling.

For each value of P, we gather statistics from the sample data. The statistics are collected from both mappers and reducers in the tuning job, which are used for two goals: *(i)* to construct the key distribution by examining the histogram of partitioning key; *(ii)* to gather input of performance measurements in each phase for our cost model. In this way, the work done in this step is also used to benchmark system performance and provide input for cost model. This is an improvement to other standard sampling methods.

3) Cost-Based Pivot Selection: In Section III-B we have discussed qualitatively the impact of the pivot position over the runtime of map and reduce phases. We now describe our pivot selection technique in detail. Here, we present a cost model to predict the runtime of a ROLLUP query implementing the HII algorithm, and use it as a reference to optimize the value of P.

4) The Model: The ROLLUP query runtime T_J is defined as a function of P:

$$T_J(P) = \overline{T_M}(P) * \alpha + T_{R_{\text{max}}}(P), \quad (1)$$

where $\overline{T_M}$ represents the average runtime of a map task, α is the number of map *waves* and $T_{R_{\text{max}}}$ is the runtime of the slowest reduce task.

Waves are due to the fact that the number of map slots can be smaller than the number of input splits to read. The number of waves is computed as

$$\alpha = \left\lceil \frac{\text{input size}}{\text{input split size} \cdot \text{number of map slots}} \right\rceil. \quad (2)$$

We assume here that the number of reduce tasks will not be larger than the number of available reduce slots (indeed, such a setting is generally not recommended in MapReduce), therefore for simplicity, we consider the reduce phase to have one wave only.

We decompose the map and reduce phases into several steps. The mappers *read* the input, *replicate* the data, *write* map output records to memory, *sort* output records, *combine* and *spill* output to local disk. For some steps such as *read*, *parse* and *replicate*, their runtime does not depend on the pivot position. Since we are interested in finding the value of P that minimizes the running time rather than predicting the running time itself, we focus on minimizing the variable parts of T_M:

$$\begin{aligned} T'_M(P) = {} & c_{\text{write},P}(2\beta) + c_{\text{sort},P}(2\beta) \\ & + c_{\text{combine},P}(2\beta) + c_{\text{spill},P}(\beta'_P). \end{aligned} \quad (3)$$

Here, β is the number of input tuples of a mapper; β' is the number of output tuples of its combiner; $c_{\lambda,P}(x)$ is a cost function that returns the runtime of step λ (write, sort, combine, spill) and depends on a particular value of P. Since HII generates 2 outputs for each input tuple, we have 2β as the input of $c_{\lambda,P}(x)$.

Similarly to the map phase, the reducers *shuffle* and *merge* their input records, *process* the ROLLUP operator and write outputs to a distributed file system (*DFS*). We therefore estimate the reduce phase as:

$$T_R(P) = c_{\text{shuffle},P}(\gamma_P) + c_{\text{process},P}(\gamma_P) + c_{\text{DFS},P}(\gamma'_P), \quad (4)$$

where γ_P is the number of reducer input records and γ'_P is the number of reducer output records.

5) Regression-Based Runtime Prediction: We propose a novel approach to predict the runtime of each step. This approach is not only completely transparent to users but also flexible enough to deal with any cluster configuration. We achieve such flexibility and transparency by using the tuning job we introduced in Section III-C2 to obtain performance measurements. We apply regression on these measurements using the least squares method to predict the runtime of the ROLLUP query.

Performance Measurements: The tuning job can be thought of as a preliminary ROLLUP job, *i.e.* it is composed by the same steps of the ROLLUP query. For each step, we measure the number of its input records and its runtime as a data point (x, T_x). To gain more accurate runtime prediction, we generate several data points. In the map phase, the tuning job runs multiple mappers on small, different chunk sizes; each chunk size is a data point[2]. It also measures the runtime of the *process* phase and *DFS* I/O at several different points in time during the reduce phase.

Linear Regression: Let us consider a step λ: we model its cost as a linear function $c_{\lambda,P}(x) = a + bx$. The sort step is an exception because of its $\mathcal{O}(n \log n)$ complexity: $c_{\text{sort},P}(x) = a + bx + cx \log x$. In the tuning phase, we obtain for each step a list of data points $[(x_1, T_{x_1}), (x_2, T_{x_2}), \ldots, (x_\mu, T_{x_\mu})]$; we use least square fitting to find the coefficients a and b such that our function $c(x)$ minimizes the error $S = \sum_{i=1}^{\mu}(T_{x_i} - c(x_i))^2$. This approach necessitates the chunk sizes x_i we use for sampling to be of variable size. Once a and b are known, we calculate the runtime T_λ on the original dataset.

Runtime Prediction and Pivot Selection: Now that all $c_{\lambda,P}$ functions are known, we need to evaluate β, β', γ and γ' in order to obtain runtime predictions from Equations 3 and 4. Again, using regression we can estimate β, β', γ and γ'. We have now all the required values; we can therefore evaluate Equations 3 and 4 for all possible values of P. The value P^* that minimizes $T'_M(P^*) + T_R(P^*)$ is our chosen value as pivot position.

IV. Experimental Evaluation

We now proceed with an experimental evaluation of our ROLLUP operator, implemented for Apache Pig. Our experimental evaluation is done on a Hadoop cluster of 20 machines with 2 map and 1 reduce slot each. The HDFS block size is set to 128MB. We execute ROLLUP aggregates over date-time dimensions using synthetic and real-life datasets; our reference performance metric is *job runtime*, with jobs being executed in an isolated cluster. We note that all results in the following are the average value of at least 10 runs: the standard error of results is smaller than 3%, so for the sake of readability, we omit error bars from our figures.

A. Datasets and ROLLUP Queries

In our experiments, we use 4 illustrative datasets. Each dataset has tuples with schema *year, month, day, hour, minute, second* and a value v. The ROLLUP operator computes the total value $V = \sum v$ per each date-time dimension and as a whole (*i.e.*, the `total` level). Specifically, we used the following datasets:

- **Synthetic Telco Logs (STL)**: 1 billion records ranged in 30- years with uniform distribution.
- **Skewed Synthetic Telco Logs (SSTL)**: 1 billion records. The tuples are in a 3-year time-frame, according to a power-law distribution with coefficient $\alpha = 2.5$.
- **Simplified Integrated Surface Database(ISD)**:[3] nearly 2.5 billion records in 114 years. This dataset is heavily skewed towards recent years, as the last 10 years contain 45% of the total records.
- **Reverse DNS (RDNS)**: a sub-set of the Internet Census2012 [4]. It has 3.5 billion records spanning 5 months.

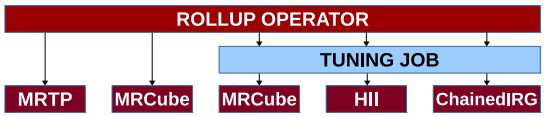

Figure 2. ROLLUP operator with our tuning job

B. Experimental Results

We now present our experimental evaluation that uses a prototype implementation of our ROLLUP operator for the Apache Pig system. Figure 2 illustrates the different flavors of our operator that we used in our experimental campaign.

First, we provide a comparative analysis of our operator using 4 ROLLUP algorithms: the standard MRTP, and our implementation of MRCube, HII and ChainedIRG. This series of experiments are an illustration of the versatility of our approach, that yields a ROLLUP operator that can achieve substantial performance gains over current standards.

Next, we focus on the tuning job, customized for the HII ROLLUP algorithm: we show that our cost-based optimizer can indeed find the most suitable operating point to achieve minimum job runtime. In addition, we measure the overhead imposed by the tuning job, and relate it to its optimization accuracy.

Finally, to validate the efficiency of our design, we compare the overhead of the tuning job customized for MRCube against the original 3-phase MRCube design, and show that even such an algorithm could benefit from our approach.

1) Comparative Performance Analysis: In this series of experiments, we execute a simple ROLLUP query, as shown below, on the four datasets described above, and measure the runtime required to complete the job using different flavors of the ROLLUP operator.

```
A = LOAD path/file AS (y, M, d, h, m, s, v);
B = CUBE A BY ROLLUP(y, M, d, h, m, s) RATE
    samplingRateValue;
C = FOREACH B GENERATE group, SUM(cube.v);
```

[2]In our experiments, the chunk sizes are 256KB, 512KB, 1MB, 2MB.

[3]http:// www.ncdc.noaa.gov/oa/climate/isd/

[4]http://internetcensus2012.bitbucket.org/

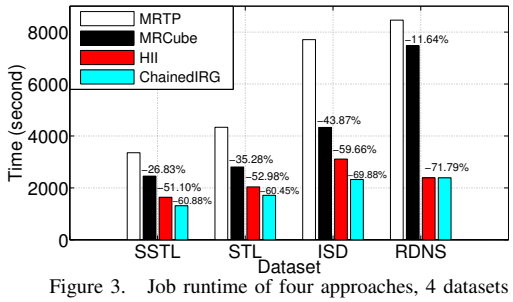

Figure 3. Job runtime of four approaches, 4 datasets

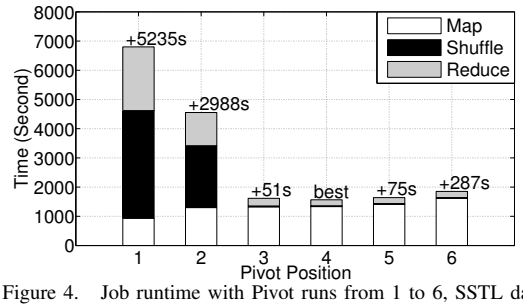

Figure 4. Job runtime with Pivot runs from 1 to 6, SSTL dataset

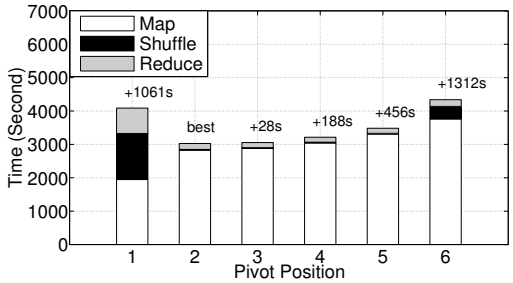

Figure 5. Job runtime with Pivot runs from 1 to 6, ISD dataset

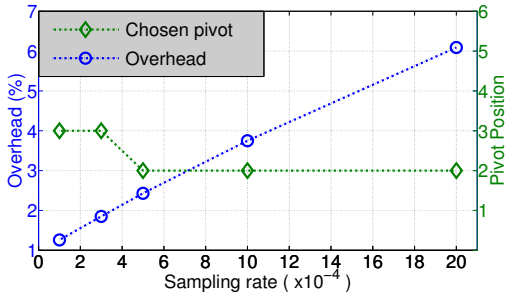

Figure 6. Overhead and accuracy trade-off, ISD dataset

Figure 3 compares the runtime of different ROLLUP operators using MRTP, MRCube, HII and ChainedIRG algorithms for all datasets. On top of the bars of MRCube, HII, and ChainedIRG, we indicate the the gain of each algorithm with respect to MRTP: for example, -26.83% on top of MRCube means that the corresponding jobs terminate 26.83% quicker than with the current Apache Pig implementation.

For all datasets, our operator for the HII algorithm outperforms the MRTP implementation by at least 50%. Indeed, the map phase of MRTP generates 7 output tuples for each input one, while HII only generates at most 2 tuples. When customized for the ChainedIRG algorithm, our operator runs faster than HII, as its map phase generates only 1 tuple. For RDNS dataset, both HII and ChainedIRG degenerate to $P = 1$ which is the IRG algorithm, which explains the identical runtime.

When compared to MRCube, both the HII and ChainedIRG variants perform better. The first reason is that the mappers in MRCube generate more tuples than HII and ChainedIRG: 3 for ISD and STL, 4 for SSTL and even 5 for RDNS datasets. The second reason is that the reduce phase of MRCube incurs redundant computation. The third reason is that our tuning job runs faster than the first job of MRCube (that would correspond only to a fraction of our tuning job's mission: data sampling and cardinality estimation).

2) Cost-based Parameter Selection Validation: In this series of experiments, we override the automatic selection of the pivot position, but allow the load balancing at reducers, to proceed with a "brute- force", experimental approach to validate the cost-based optimizer of our tuning job.

For all datasets, we run with the full-fledged tuning job and compare the pivot position output by the cost-based optimizer to that yielding the smallest job runtime, for all possible (manually set) positions. Due to the lack of space, we only present results for the SSTL and ISD datasets, but we obtain similar results for all other datasets. Also our ChainedIRG operator exhibits the same pattern we present here for the HII algorithm. In the following Figures, we report (on top of each histogram) the increment (in seconds) in query runtime for sub-optimal pivot positions.

Query runtime as a function of P for the SSTL and ISD dataset are reported in Figures 4 and Figure 5, respectively. First, we note that $P = 1$ is essentially IRG. It has the fastest map phase, but the worst performance. This is due to the lack of parallelism which results in longer shuffle and reduce runtime. At the other extreme, $P = 6$ also results in performance loss for several reasons, including low combiner efficiency (and hence longer shuffle runtime) and longer reducer runtime. The SSTL dataset presents a peculiar case: since the data covers only 3 years, the query runtime corresponding to $P = 2$ can only utilize 3 of the 20 available reducers in our cluster. As a result, both shuffle and reduce phases take a long time to process, due to the lack of parallelism. Instead, in the ISD dataset, the data covers 114 years. It means that $P = 2$ can fully utilize parallelism. Finally, we verified that the pivot chosen in our tuning job correctly specified $P = 4$ and $P = 2$ as the optimal pivot positions for the SSTL and ISD datasets respectively.

3) Overhead and Accuracy trade off of the Tuning Job: We define the overhead as the runtime of the tuning job divided by the total runtime of the ROLLUP operator. We examine this overhead as a function of sampling rate. We also study the trade-off between the overhead and the accuracy of the pivot position determined by our cost model. We plot the overhead of the tuning job, along with the pivot

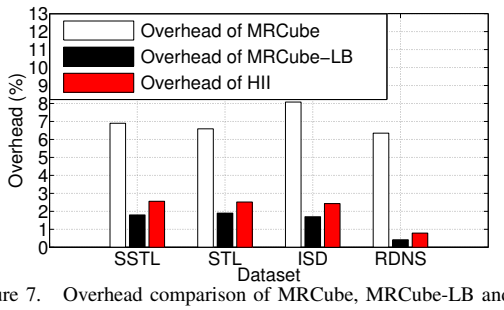

Figure 7. Overhead comparison of MRCube, MRCube-LB and HII.

position that the cost model selects with sampling rate from 0.0001 to 0.002. Again due to lack of space, we only show the plot for the ISD dataset as a representative result.

In Figure 6, due to data skew, a low sampling rate does not produce sufficiently accurate cardinalities, and as a consequence, the cost model and the load-balancing phase under-perform: they cannot determine the optimal pivot position. Nevertheless, the pivot position chosen by our tuning job quickly stabilizes to the optimal value ($P = 2$). As a consequence, the sampling phase only imposes less than 3% overhead of the whole operator runtime.

4) Efficiency of Tuning Job: Finally, to conclude our experimental evaluation, we verify our tuning job efficiency in comparison to that of MRCube. We integrate load balancing into MRCube and call it MRCube-LB. In MRCube-LB, we replace the first round of the MRCube algorithm by our tuning job: the cost-based optimizer is disabled, while data sampling, cardinality estimation and load balancing are active. Next, we present a notable improvement of the MRCube-LB with respect to the originial MRCube. Figure 7 shows, for all datasets, the *overhead* of our operator in three versions: MRCube, MRCube-LB and HII algorithms. Both the overhead of the HII and MRCube-LB versions are always smaller than that of the original MRCube by at least 60%. Comparing the HII and MRCube-LB versions, the marginally higher overhead of the former is due to the execution of the cost-based optimizer.

In conclusion, this series of experiments indicate that our tuning job is versatile, lightweight, and accurate.

V. Conclusion

Summarization queries such as the ones using ROLLUP are crucial to let users explore very large datasets; however, existing high-level languages in the Hadoop ecosystem provide simple and quite inefficient implementations.

The main contribution of this work was the design of an efficient, skew- resilient ROLLUP operator for MapReduce high-level languages. Its principal component, the tuning job, is a lightweight mechanism that materializes in a small job executed prior to the ROLLUP query. The tuning job performs data and performance sampling to achieve, at the same time, cost-based optimization and load balancing of a range of ROLLUP algorithms.

Our extensive experimental validation illustrated the flexibility of our approach, and showed that – when appropriately tuned – some ROLLUP algorithms dramatically outperform the one used in current implementations of the ROLLUP operator for the Apache Pig system. In addition, we showed that the tuning job is lightweight yet accurate: cost-based optimization determines the best parameter settings with small overheads, which are mainly dictated by the data sampling scheme. Our work is available as an open-source project, and it is currently under consideration as a contribution to Apache Pig.

We conclude by noting that the ROLLUP operator constitutes a suitable building block for implementing more generic and expensive aggregations using directives such as GROUPING SETS or CUBE; our future work aims at designing efficient CUBE and GROUPING SETS operators for MapReduce systems.

References

[1] J. Gray *et al.*, "Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals," *DMKD*, 1997.
[2] V. Harinarayan *et al.*, "Implementing data cubes efficiently," in *ACM SIGMOD*, 1996.
[3] S. Agarwal *et al.*, "On the computation of multidimensional aggregates," in *PVLDB*, 1996.
[4] K. Beyer and R. Ramakrishnan, "Bottom-up computation of sparse and iceberg cubes," in *ACM SIGMOD*, 1999.
[5] J. Dean and S. Ghemawat, "MapReduce : Simplified Data Processing on Large Clusters," in *ACM OSDI*, 2004.
[6] "http://hadoop.apache.org."
[7] "http://pig.apache.org."
[8] "http://hive.apache.org."
[9] Y. Chen *et al.*, "Interactive analytical processing in big data systems: A cross-industry study of MapReduce workloads," in *PVLDB*, 2012.
[10] S. Bellamkonda *et al.*, "Adaptive and Big Data Scale Parallel Execution in Oracle," in *PVLDB*, 2013.
[11] F. Dehne *et al.*, "Parallelizing the data cube," *DPDB*, 2002.
[12] R. T. Ng *et al.*, "Iceberg-cube computation with PC clusters," *ACM SIGMOD Record*, 2001.
[13] Y. Chen *et al.*, "Parallel ROLAP data cube construction on shared-nothing multiprocessors," *DPDB*, 2004.
[14] S. Goil *et al.*, "A parallel scalable infrastructure for OLAP and data mining," *IDEAS*, 1999.
[15] T. White, *Hadoop - The Definitive Guide: Storage and Analysis at Internet Scale*. O'Reilly, 2012.
[16] A. Baer *et al.*, "Two parallel approaches to network data analysis," in *LADIS*, 2011.
[17] H. D. Phan *et al.*, "On the design space of MapReduce ROLLUP aggregates." in *EDBT/ICDT Workshops*, 2014.
[18] A. Nandi *et al.*, "Distributed cube materialization on holistic measures," in *IEEE ICDE*, 2011.
[19] Y. Kwon *et al.*, "Skew-resistant parallel processing of feature-extracting scientific user-defined functions," in *ACM SoCC*, 2010.
[20] Y. Kwon *et al.*, "Skewtune: mitigating skew in MapReduce applications," in *ACM SIGMOD*, 2012.
[21] R. Vernica *et al.*, "Efficient parallel set-similarity joins using MapReduce," in *ACM SIGMOD*, 2010.
[22] T. Nykiel *et al.*, "MRShare: sharing across multiple queries in MapReduce," in *PVLDB*, 2010.
[23] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, 1979.
[24] S. Chaudhuri *et al.*, "Effective use of block-level sampling in statistics estimation," in *Proc. ACM SIGMOD*, 2004.