On the design space of MapReduce ROLLUP aggregates

Duy-Hung Phan
EURECOM
phan@eurecom.fr

Matteo Dell'Amico
EURECOM
dellamic@eurecom.fr

Pietro Michiardi
EURECOM
michiard@eurecom.fr

ABSTRACT

We define and explore the design space of efficient algorithms to compute ROLLUP aggregates, using the MapReduce programming paradigm. Using a modeling approach, we explain the non-trivial trade-off that exists between parallelism and communication costs that is inherent to a MapReduce implementation of ROLLUP. Furthermore, we design a new family of algorithms that, through a single parameter, allow to find a "sweet spot" in the parallelism vs. communication cost trade-off. We complement our work with an experimental approach, wherein we overcome some limitations of the model we use. Our results indicate that efficient ROLLUP aggregates require striking the good balance between parallelism and communication for both one-round and chained algorithms.

1. INTRODUCTION

Online analytical processing (OLAP) is a fundamental approach to study multi-dimensional data involving the computation of, for example, aggregates on data that are accumulated in traditional data warehouses. When operating on massive amounts of data, it is typical for business intelligence and reporting applications, to require data summarization, which is achieved using standard SQL operators such as GROUP BY, ROLLUP, CUBE, and GROUPING SETS.

Despite the tremendous amount of work carried out in the database community to come up with efficient ways of computing data aggregates, little work has been done to extend these lines of work to cope with massive scale. Indeed, the main focus of prior works in this domain has been on single server systems or small clusters executing a distributed database, implementing efficient implementations of CUBE and ROLLUP operators, in line with the expectations of low-latency access to data summaries [6, 8, 11, 13, 14, 19]. Only recently, the community devoted attention to solve the problem of computing data aggregates at massive scales using data intensive, scalable computing engines such as MapReduce [10]. In support of the growing interest in computing data aggregates on batch-oriented systems, several high-level languages built on top of MapReduce, such as PIG [3] and HIVE [2], support simple implementations of, for example, the ROLLUP operator.

The endeavor of this work is to take a systematic approach to study the design space of the ROLLUP operator: besides being widely used on its own, ROLLUP is also a fundamental building block used to compute CUBE and GROUPING SETS [7]. We study the problem of defining the design space of algorithms to implement ROLLUP through the lenses of a recent model of MapReduce-like systems [4]. The model explains the trade-offs that exist between the degree of parallelism that is possible to achieve and the communication costs that are inherently present when using the MapReduce programming model. In addition, we overcome current limitations of the model we use (which glosses over important aspects of MapReduce computations) by extending our analysis with an experimental approach. We present instances of algorithmic variants of the ROLLUP operator that cover several points in the design space, implement and evaluate them using an Hadoop cluster.

In summary, our contributions are the following:

- We study the design space that exists to implement ROLLUP and show that, while it may appear deceivingly simple, it is not a straightforward embarrassing parallel problem. We use modeling to obtain bounds on parallelism and communication costs.
- We design and implement new ROLLUP algorithms that can match the bounds we derived, and that swipe the design space we were able to define.
- We pinpoint the essential role of *combiners* (an optimization allowing pre-aggregation of data, which is available in real instances of the MapReduce paradigm, such as Hadoop [1]) for the practical relevance of some algorithm instances, and proceed with an experimental evaluation of several variants of ROLLUP implementations, both in terms of their performance (runtime) and their efficient use of cluster resources (total amount of work).
- Finally, our ROLLUP implementations exist in Java MapReduce and have been integrated in our experimental branch of PIG, which are available in a public repository.[1]

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

The remainder of this paper is organized as follows. Section 2 provides background information on the model we use in our work and presents related work. Section 3 illustrates a formal problem statement and Section 4 presents several variants of ROLLUP algorithms. Section 5 outlines our experimental results to evaluate the performance of the algorithms we introduce in this work. Finally, Section 6 concludes our work and outlines future research directions.

2. BACKGROUND AND RELATED WORK

We assume the reader to be familiar with the MapReduce [10] paradigm and its open-source implementation Hadoop [1, 20]. First, we give a brief summary of the model introduced by Afrati *et al.* [4], which is the underlying tool we use throughout our paper. Then, we present related works that focus on the MapReduce implementation of popular data analytics algorithms.

The MapReduce model. Afrati *et al.* [4] recently studied the MapReduce programming paradigm through the lenses of an original model that elucidates the trade-off between parallelism and communication costs of single-round MapReduce jobs. The model defines the design space of a MapReduce algorithm in terms of *replication rate* and *reducer-key size*. The replication rate r is the average number of $\langle \text{key}, \text{value} \rangle$ pairs created from each input in the map phase, and represents the communication cost of a job. The reducer-key size q is the upper bound of the size of list of values associated to a reducer-key. Jobs have higher degrees of parellelism when q is smaller. For some problems, parallelism comes at the expense of larger communication costs, which may dominate the overall execution time of a job.

Afrati *et al.* show how to determine the relation between r and q. This is done by first bounding the amount of input a reducer requires to cover its outputs. Once this relation is established, a simple yet effective "recipe" can be used to relate the size of the input of a job to the replication rate and to the bounds on output covering introduced above. As a consequence, given a problem (*e.g.*, finding the Hamming distance between input strings), the model can be used to establish bounds on r and q, which in turn define the design space that instances of algorithms solving the original problem may cover.

Related work. Designing efficient MapReduce algorithms to implement a wide range of operations on data has received considerable attention recently. Due to space limitations, we cannot give justice to all works that addressed the design, analysis and implementation of graph algorithms, clustering algorithms and many other important problems: here we shall focus on algorithms to implement SQL-like operators. For example, the relational JOIN operator is not supported directly in MapReduce. Hence, attempts to implement efficient JOIN algorithms in MapReduce have flourished in the literature: Blanas *et al.* [9] studied *Repartition Join*, *Broadcast Join*, and *Semi-Join*. More recent work tackle more general cases like theta-joins [17] and multi-way-joins [5].

With respect to OLAP data analysis tasks such as CUBE and ROLLUP, efficient MapReduce algorithms have only lately received some attention. A first approach to study CUBE and ROLLUP aggregates has been proposed by Nandi *et al.* [16]; this algorithm, called "naive" by the authors, is called *Vanilla* in this work. MR-Cube [16] mainly focuses on algebraic aggregation functions, and deals with data skew; it implements the CUBE operator by breaking the algorithm in three phas-es. A first job samples the input data to recognize possible reducer-unfriendly regions; a second job breaks those regions into sub-regions, and generates corresponding $\langle \text{key}, \text{value} \rangle$ pairs to all regions, to perform partial data aggregation. Finally, a last job reconstructs all sub-regions results to form the complete output. The MR-Cube operator naturally implements ROLLUP aggregates. However in the special case of ROLLUP, the approach has two major drawbacks: it replicates records in the map phase as in the naive approach and it performs redundant computation in the reduce phase.

For the sake of completeness, we note that one key idea of our work (in-reducer grouping) shares similar traits to what is implemented in the Oracle database [7]. However, the architectural differences with respect to a MapReduce system like Hadoop, and our quest to explore the design space and trade-offs of ROLLUP aggregates make such work complementary to ours.

3. PROBLEM STATEMENT

We now define the ROLLUP operation as a generalization of the SQL ROLLUP clause, introducing it by way of a running example. We use the same example in Section 4 to elucidate the details of design choices and, in Section 5, to benchmark our results.

ROLLUP can be thought of as a hierarchical GROUP BY at various granularities, where the grouping keys at a coarser granularities are a subset of the keys at a finer granularity. More formally, we define the ROLLUP operation on an input *data set*, an *aggregation function*, and a set of n *hierarchical granularities*:

- We consider a *data set* akin to a database table, with M columns $c_1, \ldots, c_M$ and L rows $r_1, \cdots r_L$ such that each row r_i corresponds to the $(r_{i1}, \ldots, r_{iM})$ tuple.
- Given a set of rows $R \subseteq \{r_1, \cdots r_L\}$, an *aggregation function* $f(R)$ produces our desired result.
- n *granularities* $d_1, \ldots, d_n$ determine the groupings that an input data is subject to. Each d_i is a subset of $\{c_1, \cdots c_M\}$, and granularities are *hierarchical* in the sense that $d_i \subsetneq d_{i+1}$ for each $i \in [1, n-1]$.

The ROLLUP computation returns the result of applying f after grouping the input by the set of columns in each granularity. Hence, the output is a new table with tuples corresponding to grouping over the finest (d_n) up to the coarsest (d_1) granularity, denoting irrelevant columns with an ALL value [12].

Example. Consider an Internet Service provider which needs to compute aggregate traffic load in its network, per day, month, year and in overall. We assume input data to be a large table with columns (c_1, c_2, c_3, c_4) corresponding to (year, month, day, payload[2]). A few example records from this dataset are shown in the following:

```
(2012,  3, 14, 1)
(2012, 12,  5, 2)
(2012, 12, 30, 3)
(2013,  5, 24, 4)
```

[2]In Kilobytes

The aggregation function f outputs the sum of values over the c_4 (payload) column. Besides SUM, other typical aggregation functions are MIN, MAX, AVG and COUNT; it is also possible to consider aggregation functions that evaluate data in multiple columns, such as for example correlation between values in different columns.

Input granularities are $d_1 = \emptyset$, $d_2 = \{\text{year}\}$, $d_3 = \{\text{year}, \text{month}\}$, and $d_4 = \{\text{year}, \text{month}, \text{day}\}$. The highest granularity, $d_1 = \emptyset$, groups on no columns and is therefore equivalent to a SQL GROUP BY ALL clause that computes the overall sum of the payload column; such an overall aggregation is always computed in SQL implementations, but it is not required in our more general formulation. We will see in the following that "global" aggregation is problematic in MapReduce.

In addition to aggregation on hierarchical time periods as in this case, ROLLUP aggregation applies naturally to other cases where data can be organized in tree-shaped taxonomies, such as for example country-state-region or unit-department-employee hierarchies.

If applied on the example, the ROLLUP operation yields the following result (we use '*' to denote ALL values):

```
(2012,  3, 14,  1)
(2012,  3,  *,  1)
(2012, 12,  5,  2)
(2012, 12, 30,  3)
(2012, 12,  *,  5)
(2012,  *,  *,  6)
(2013,  5, 24,  4)
(2013,  5,  *,  4)
(2013,  *,  *,  4)
(   *,  *,  *, 10)
```

Rows with ALL values represent the result of aggregation at coarser granularities: for example, the `(2012, *, *, 6)` tuple is the output of aggregating all tuples from year 2012.

Aggregation Functions and Combiners. In MapReduce, it is possible to pre-aggregate values computed in mappers by defining *combiners*. We will see in the following that combiners are crucial for the performance of algorithms defined in MapReduce. While many useful aggregation functions are subsceptible to being optimized through combiners, not all of them are. Based on the definition by Gray *et al.* [12], when an aggregation function is *holistic* there is no constant bound on the size of a combiner output; representative holistic functions are MEDIAN, MODE and RANK.

The algorithms we define are differently subsceptible to the presence and effectiveness of combiners. When discussing the merits of each implementation, we also consider the case where aggregation functions are holistic and hence combiners are of little or no use.

4. THE DESIGN SPACE

We explore the design space of ROLLUP, with emphasis on the trade-off between communication cost and parallelism. We first apply a model to obtain theoretical bounds on replication rate and reducer key size; we then consider two algorithms (Vanilla and In-Reducer Grouping) that are at the end-points of the aforementioned trade-off, having respectively maximal parallelism and minimal communication cost. We follow up by proposing various algorithms that operate in different, and arguably more desirable, points of the trade-off space.

4.1 Bounds on Replication and Parallelism

Here we adopt the model by Afrati *et al.* [4] to find upper and lower bounds for the replication rate. Note that the model, unfortunately, does not account for combiners nor for multi-round MapReduce algorithms.

First, we define the number of all possible inputs and outputs to our problem, and a function $g(q)$ that allows to evaluate the number of outputs that can be covered with i input records. To do this, we refer to the definitions in Section 3:

1. **Input set**: we call C_i the number of different values that each column c_i can take. The total number of inputs is therefore $|I| = \prod_{i=1}^{M} C_i$.

2. **Output set**: for each granularity d_i, we denote the number of possible grouping keys as $N_i = \prod_{C_i \in d_i} C_i$ and the number of possible values that the aggregation function can output as A_i.[3] Thus, the total number of outputs is $|O| = \sum_{i=1}^{n} N_i A_i$.

3. **Covering function**: let us consider a reducer that receives q input records. For each granularity d_i, there are N_i grouping keys, each one grouping $|I|/N_i$ inputs and producing A_i outputs. The number of groups that the reducer can cover at granularity d_i is therefore no more than $\lfloor qN_i/|I| \rfloor$, and the covering function is $g(q) = \sum_{i=1}^{n} A_i \left\lfloor \frac{qN_i}{|I|} \right\rfloor$.

Lower Bound on Replication Rate. We consider p reducers, each receiving $q_i \leq q$ inputs and covering $g(q_i)$ outputs. Since together they must cover all outputs, it must be the case that $\sum_{j=1}^{p} g(q_j) \geq |O|$. This corresponds to

$$\sum_{j=1}^{p} \sum_{i=1}^{n} A_i \left\lfloor \frac{q_j N_i}{|I|} \right\rfloor \geq \sum_{i=1}^{n} N_i A_i. \tag{1}$$

Since $q_j N_i/|I| \geq \lfloor q_j N_i/|I| \rfloor$, we obtain the lower bound of the replication rate r as:

$$r = \sum_{i=1}^{p} \frac{q_i}{|I|} \geq 1. \tag{2}$$

Equation 2 seems to imply that ROLLUP aggregates is an *embarassingly parallel* problem: the $r \geq 1$ bound on replication rate does not depend on the size q_i of reducers. In Section 4, we show – for the first time – an instance of an algorithm that matches the lower bound. Instead, known instances of ROLLUP aggregates have a larger replication rate, as we shall see next.

Limits on Parallelism. Let us now reformulate Equation 2, this time requiring only that the output of the coarsest granularity d_1 is covered. We obtain

$$\sum_{j=1}^{p} \left\lfloor \frac{q_j N_1}{|I|} \right\rfloor \geq N_1.$$

[3]For the limit case $d_i = \emptyset$, $N_i = 1$, corresponding to the single empty grouping key.

Clearly, the output *cannot be covered* (the left side of the equation would be zero) unless there are reducers receiving at least $q_j \geq |I|/N_1$ input records. Indeed, the coarsest granularity imposes hard limits on the parallelism, requiring to broadcast the full input on at most N_1 reducers. This is exacerbated if – as it is the case with the standard SQL ROLLUP – there is an overall aggregation, resulting in $d_1 = \emptyset$, $N_1 = 1$ and therefore $q_j \geq |I|$. A *single reducer* needs to receive *all the input*: it appears that no parallelism whatsoever is possible.

As we show in the following, this negative result however depends on the limitations of the model: by applying combiners and/or multiple rounds of MapReduce computation, it is indeed possible to compute efficient ROLLUP aggregates in parallel.

Maximum Achievable Parallelism. Our model considers parallelism as determined by the number of reducers p and the number of input records q_j each of them processes. However, one may also consider the number of *output* records produced by each reducer: in that case, the maximum parallelism achievable is when each reducer produces at most a single output value. This can be obtained by assigning each grouping key in each granularity to a different reducer; the aggregation function is then guaranteed to output only one of the A_i possible values. This, however, implies a replication rate $r = n$; an implementation of the idea is described in the following section.

4.2 Baseline algorithms

Next, we define two baseline algorithms to compute ROLLUP aggregates: Vanilla, which is discussed in [16], and *In Reducer Grouping*, which is our contribution. Then, we propose a hybrid approach that combines both baseline techniques.

Vanilla Approach. We describe here an approach that maximizes parallelism at the detriment of communication cost; since this is the approach which is currently implemented in Apache Pig [15] we refer to it as Vanilla. Nandi *et al.* [16] refer to it as "naive".

The ROLLUP operator can be considered as the result multiple GROUP BY operations: each of them is carried out at a different granularity. Thus, to perform ROLLUP on n granularities, for each record, the vanilla approach generates exactly n records corresponding to these n grouping sets (each grouping sets belongs to one granularity). For instance, taking as input the `(2012, 3, 14, 1)` record of the running example, this approach generates 4 records as outputs of the map phase:

```
(2012, 3, 14, 1) (day granularity)
(2012, 3,  *, 1) (month granularity)
(2012, *,  *, 1) (year granularity)
(   *, *,  *, 1) (overall granularity)
```

The Reduce step performs exactly as the reduce step of a GROUP BY operation, using the first three records (year, month, day) as keys. By doing this, reducers pull all the data that is needed to generate each output record (shuffle step), and compute the aggregate (reduce step). Figure 1 illustrates a walk-through example of the vanilla approach with just 2 records.

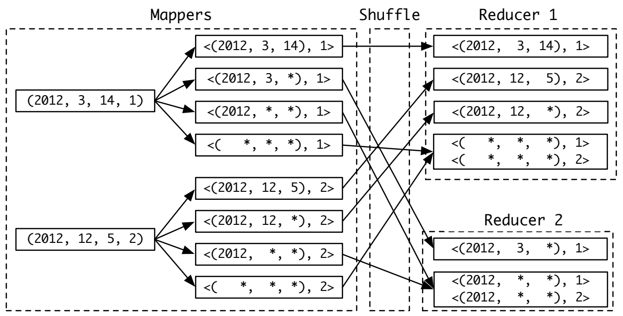


Figure 1: Example for the vanilla approach.

Parallelism and Communication Cost. The final result of ROLLUP is computed in a single MapReduce job. As discussed above, this implementation obtains the maximum possible degree of parallelism, since it can be parallelized up to a level where a single reducer is responsible of a single output value. On the other hand, this algorithm requires maximal communication costs, since for each input record, n map output records are generated. In addition, when the aggregation operation is *algebraic* [12], redundant computation is carried out in the reduce phase, since results computed for finer granularities cannot be reused for the coarser ones.
Impact of Combiners. This approach largely benefits from combiners whenever they are available, since they can compact the output computed at the coarser granularity (*e.g.*, in the example the combiner is likely to compute a single per-group value at the *year* and overall granularity). Without combiners, a granularity such as overall would result in shuffling data from *every input tuple* to a *single reducer*.

While combiners are very important to limit the amount of data sent along the network, the large amount of temporary data generated with this approach is still problematic: map output tuples need to be buffered in memory, sorted, and eventually spilled to disk if the amount of generated data does not fit in memory. This results, as we show in Section 5, in performance costs that are discernible even when combiners are present.

In-Reducer Grouping. After analyzing an approach that maximizes parallelism, we now move to the other end of the spectrum and design an algorithm that minimizes communication costs. In contrast to the Vanilla approach, where the complexity resides on the Map phase and the Reduce phase behaves as if implementing an ordinary GROUP BY clause, we propose an In-Reducer Grouping (IRG) approach, where all the logic of grouping is performed in the Reduce phase.

In-Reducer Grouping makes use of the possibility to define a *partitioner* in Hadoop [10, 20]. The mapper selects the columns of interest (in our example, all columns are needed, so the map function is simply the identity function). The keys are the finest granularity d_n (*day* in our example) but data is partitioned only by the columns of the coarsest granularity d_1. In this way, we can make sure that *1)* each reducer receives enough data to compute the aggregation function even for the coarsest granularity d_1; *2)* the intermediate keys are sorted [10, 20], so for every grouping key of any granularity d_i, the reducer will process consecutively

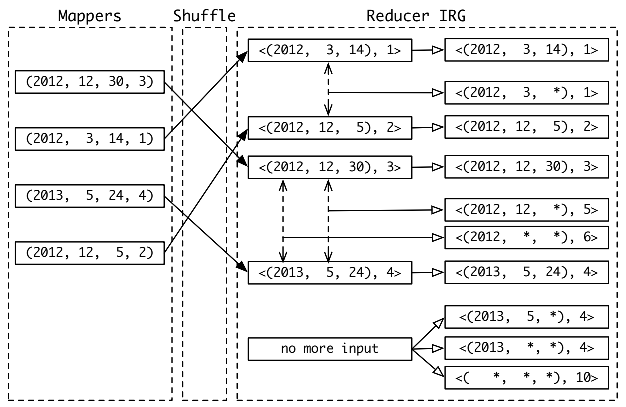


Figure 2: Example for the IRG approach.

all records pertaining to the given grouping key.

Figure 2 shows an example of the IRG approach. The mapper is the identity function, producing *(year, month, day)* as the keys and *payload* as the value. The coarsest granularity d_1 is overall, and $N_1 = 1$: hence, all $\langle \text{key}, \text{value} \rangle$ pairs are sent to a single reducer. The reducer groups all values of the same key, and processes the list of values associated to that key, thus computing the sum of all values as the total payload t. The grouping logic in the reducer also takes care of sending t to n grouping keys constructed from the reducer input key. For example, with reference to Figure 2, the input pair `(<2012, 3, 14>, 1)` implies that value $t = 1$ is sent to grouping keys `(2012, 3, 14)`, `(2012, 3, *)`, `(2012, *, *)` and `(*, *, *)`. The aggregators in these grouping keys accumulate all t values they receive. When there is no more t value for a grouping key (in our example, when *year* or *month* change, as shown by the dashed lines in Figure 2), the aggregator outputs the final aggregated value.

The key observation we exploit in the IRG approach is that a secondary, lexicographic sorting, is fundamental to minimize state in the reducers. For instance, at *month* granularity, when the reducer starts processing pair `(<2013, 5, 24>, 4)`, then we are guaranteed that all grouping keys of month smaller than `(2013, 5)` (*e.g.* `(2012, 12)`) have already been processed and should be output without further delay. This way reducers need not keep track of aggregators for previous grouping keys: reducers only use n aggregators, one for each granularity.

To summarize, the IRG approach extensively relies on the idea of an *on-line algorithm*: it makes a single pass over its input, maintaining only the necessary state to accumulate aggregates (both algebraic and holistic) at different granularities, and produces outputs as the reduce function iterates over the input.

Parallelism and Communication Cost. Since mappers output one tuple per input record, the replication rate of the IRG algorithm meets the lower bound of 1, as showed in Equation 2. On the other hand, this approach has limited parallelism, since it uses no more reducers than the number N_1 of grouping keys at granularity d_1. In particular, when an overall aggregation is required, IRG can only work on a *single reducer*. As a result, IRG is likely to perform less work and require less resources than the Vanilla approach described previously, but it cannot benefit from parallelization in the reduce phase.

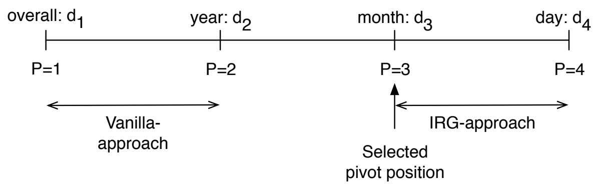


Figure 3: Pivot position example.

Impact of Combiners. Since the IRG algorithm minimizes communication cost, combiners only perform well if pre-aggregation at the finest granularity d_n is beneficial – *i.e.*, if the number of rows L in the data set is definitely larger than the number of grouping keys at the finest granularity, N_n. As such, the performance of the IRG approach suffers the least from the absence of combiners, *e.g.* when aggregation functions are not algebraic.

If the aggregate function is algebraic, however, the IRG algorithm is designed to re-use results from finer granularities in order to build the aggregation function *hierarchically*: in our running example, the aggregate of the total payload processed in a month can be obtained by summing the payload processed in the days of that month, and the aggregate for a year can likewise be computed by adding up the total payload for each month. Such an approach saves and reuses computation in a way that is not possible to obtain with the Vanilla approach.

Hybrid approach: Vanilla + IRG. We have shown that Vanilla and IRG are two "extreme" approaches: the first one maximizes parallelism at the expense of communication cost, the second one instead minimizes communication cost but does not provide good parallelism guarantees.

Neither approach is likely to be an optimal choice for such a tradeoff: in a realistic system, we are likely to have way less reducers than number of output tuples to generate (therefore making the extreme parallelism guarantees produced by Vanilla excessive); however, in particular when an overall aggregate is needed, it is reasonable to require an implementation that does not have the bottleneck of a single reducer.

In order to benefit at once from an acceptable level of parallelism and lower communication overheads, we propose an *hybrid* algorithm that fixes a *pivot* granularity P: all aggregate functions on granularities between d_P and d_n are computed using the IRG algorithm, while aggregates for granularities above d_P are obtained using the Vanilla approach. A choice of $P = 1$ is equivalent to the IRG algorithm, while $P = n$ corresponds to the Vanilla approach.

Let us consider again our running example, and fix the pivot position at $P = 3$, as shown in Figure 3. This choice implies that aggregates for the overall and *year* granularities d_1, d_2 are computed using the Vanilla approach, while aggregates for the other granularities d_3, d_4 (*month* and *day*) are obtained using the IRG algorithm. For example, for the `(2012, 3, 14, 1)` tuple, the hybrid approach produces *three* output records at the mapper:

```
(2012, 3, 14, 1) (day granularity)
(2012, *,  *, 1) (year granularity)
(   *, *,  *, 1) (overall granularity)
```

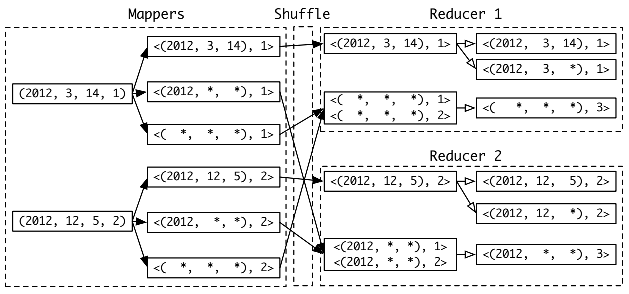


Figure 4: Example for the Hybrid Vanilla + IRG approach.

In this case the map output key space is partitioned by the month granularity, so that there is *1)* one reducer per each month in the input dataset, that computes aggregates for granularities up to the month level, and *2)* multiple reducers that compute aggregates for the *overall* and *year* granularities. Figure 4 illustrates an example with two reducers.

Some remarks are in order. Assuming a uniform distribution of the input dataset, the load on reducers of type *1)* is expected to be evenly shared, as an input partition corresponds to an individual month and not the whole dataset. The load on reducers of type *2)* might seem still prohibitive; however, we note that when combiners are in place they are going to vastly reduce the amount of data sent to the reducers responsible of the overall and *year* aggregate computation. For our example, the reducers of type *2)* receive few input records, because the overall and *year* aggregates can be largely computed in the map phase. Furthermore, we remark that the efficiency of combiners in reducing input data to reducers (and communication costs) is very high for coarse granularities, and decreases towards finer granularities: this is why the IRG algorithm applies the Vanilla approach from the pivot position, up to coarse granularities.
Parallelism and Communication Cost. The performance of the hybrid algorithm depends on the choice of P: the replication rate (before combiners) is P. The number of reducer that this approach can use is the total of *1)* N_P grouping keys that are handled with the IRG algorithm, and *2)* $\sum_{i=0}^{P-1} N_i$ grouping keys that are handled with the Vanilla approach. Ideally, an *a priori* knowledge of the input data can be used to guide the choice of the pivot position. For example, if the data in our running example is known to span over tens of years and we know we only have ten reducer slots available (*i.e.*, at most ten reducer tasks can run concurrently), a choice of partitioning by year ($P = 2$) would be reasonable. Conversely, if the dataset only spans a few years and hundreds of reducer slots are available, then it would be better to be more conservative and choose $P = 3$ or $P = 4$ to obtain better parallelism at the expense of a higher communication cost.
Impact of Combiners. The hybrid approach heavily relies of combiners. Indeed, when combiners are not available, all input data will be sent to the one reducer in charge of the overall granularity; in this case, it is then generally better to choose $P = 1$ and revert to the IRG algorithm. However, when the combiners are available, the benefit for the hybrid approach is considerable, as discussed above.

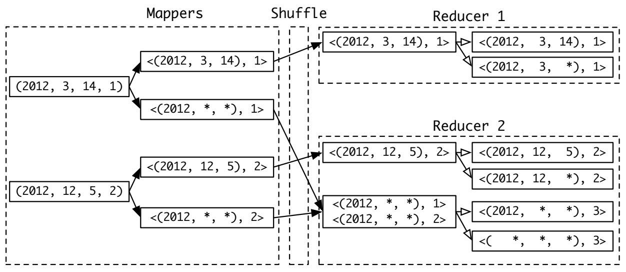


Figure 5: Example for the Hybrid IRG + IRG approach.

4.3 Alternative hybrid algorithms

We now extend the hybrid approach we introduced previously, and propose two alternatives: a single job involving two parallel IRG instances, and a chained job involving a first IRG computation and a final IRG aggregation.

Hybrid approach: IRG + IRG. In the previous section, we have shown that it is possible to design an algorithm aiming at striking a good balance between parallelism and replication rate, using a single parameter, *i.e.* the pivot position. In the baseline hybrid approach, parallelism is an increasing function of the replication rate, so that better parallelism is counterbalanced by higher communication costs in the shuffle phase.

Here, we propose an alternative approach that results in a constant replication rate of 2: the "trick" is to replace the Vanilla part of the baseline hybrid algorithm with a second IRG approach. Using the same running example as before, for the tuple `(2012, 3, 14, 1)`, and selecting the pivot position $P = 3$, the *two* map output records are:

```
(2012, 3, 14, 1) (day granularity)
(2012, *,  *, 1) (year granularity)
```

Figure 5 illustrates a running example. In this case, the map output key space is partitioned by the *month* granularity, such that there is one reducer per month that uses the IRG algorithm to compute aggregates; in addition, there is *one reducer* receiving all tuples having ALL values taking care of the *year* and overall granularities, using again the IRG approach. As before, the role of combiners is crucial: the amount of `(year, *, *, payload)` tuples that are sent to the single reducer taking care of *year* and *overall* aggregates is likely to be very small, because opportunities to compute partial aggregates in the map phase are higher for coarser granularities.
Parallelism and Communication Cost. This algorithm has a constant replication rate of 2. As we show in Section 5, the choice of the pivot position P is here much less decisive than for the baseline hybrid approach: this can be explained by the fact that moving the pivot to finer granularities does not increase communication costs, as long as the load on the reducer taking care of the aggregates for coarse granularities remains low.
Impact of Combiners. Similarly to the baseline hybrid approach, this algorithm relies heavily on combiners; if combiners are not available, then, a simple IRG approach would be preferable.

Chained IRG. It is possible to further decrease the replication rate and hence the communication costs of computing ROLLUP aggregates by adopting a multi-round approach composed of two chained MapReduce jobs. In this case, the first job pre-aggregates results up to the pivot position P using the IRG algorithm; the second job uses partial aggregates from the first job to produce – on a single reducer – the final aggregate result, again using IRG. We note here that a similar observation, albeit for computing matrix multiplication, is also discussed in detail in [4].

Parallelism and Communication Cost. The parallelism of the first MapReduce job is determined by the amount N_P of grouping keys at the pivot position; the second MapReduce job, has a single reducer. However, the input size of the second job is likely to be orders of magnitude smaller than the first one, so that the runtime of the reduce phase of the second job – unless the pivot position puts too much effort on the second job – is generally negligible. The fact that the second reducer operates on a very small amount of input, results in a replication rate very close to 1.

The main drawback of the chained approach is due to job scheduling strategies: if jobs are scheduled in a system with idle resources, as we show in Section 5, the chained IRG algorithm results in the smallest runtime. However, in a loaded system, the second (and in general very small) MapReduce job could be scheduled later, resulting in artificiously large delays between job submission and its execution.

Impact of Combiners. This approach does not rely heavily on combiners *per se*. However, it requires the aggregation function to be algebraic in order to make it possible for the second MapReduce job to re-use partial results.

5. EXPERIMENTAL EVALUATION

We now proceed with an experimental approach to study the performance of the algorithms we discussed in this work. We use two main metrics: *runtime* – *i.e.* job execution time – and *total amount of work*, *i.e.* the sum of individual task execution times. Runtime is relevant on idle systems, in which job scheduling does not interfere with execution times; total amount of work is instead an important metric to study in heavily loaded systems where spare resources could be assigned to other pending jobs.

5.1 Experimental Setup

Our experimental evaluation is done on a Hadoop cluster of 20 slave machines (8GB RAM and a 4-core CPU) with 2 map and 1 reduce slot each. The HDFS block size is set to 128MB. All results shown in the following are the average of 5 runs: the standard deviation is smaller than 2.5%, hence – for the sake of readability – we omit error bars from our figures.

We compare the five approaches described in Section 4: baseline algorithms (Vanilla, IRG, Hybrid Vanilla + IRG) and alternative hybrid approaches (Hybrid IRG + IRG Chained IRG). We evaluate a single ROLLUP aggregation job over (*overall*, *year*, *month*, *day*, *hour*, *minute*, *second*) that uses the SUM aggregate function which, being algrebraic, can benefit from combiners. Our input dataset is a synthetic log-trace representing historical traffic measurements taken by an Internet Service Provider (ISP): each record in our log has *1)* a time-stamp expressed in (*year*, *month*, *day*, *hour*, *minute*, *second*); and *2)* a number representing the payload (e.g. number of bytes sent or received

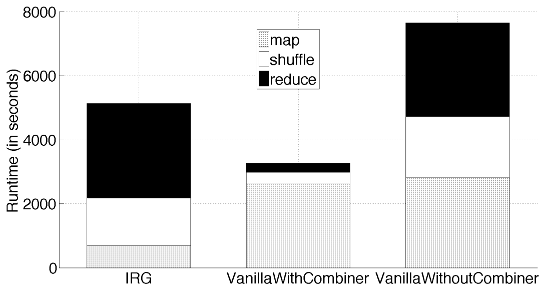


Figure 6: Impact of combiners on runtime for the Vanilla approach.

over the ISP network). The time-stamp is generated uniformly at random within a variable number of years (where not otherwise specified, the default is 40 years). The payload is a uniformly random positive integer. Overall, our dataset comprises 1.5 billion binary tuples of size 32 bytes each, packed in a SequenceFile [20].

5.2 Results

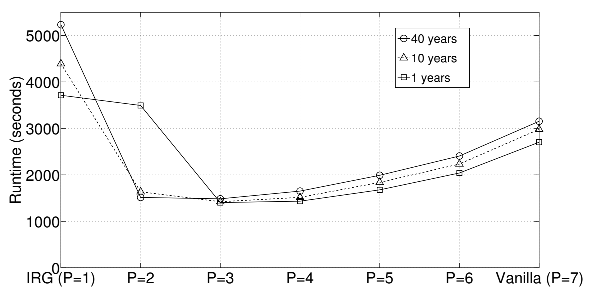
This section presents a range of results we obtained in our experiments. Before delving into a comparative study of all the approaches outlined above, we first focus on studying the impact of combiners on the performance of the Vanilla approach. Then, we move to a detailed analysis of runtime and amount of work for baseline algorithms (Vanilla, IRG, and Hybrid), and we conclude with an overview to outline merits and drawbacks of alternative hybrid approaches.

The role of combiners. Figure 6 illustrates a break-down of the runtime for computing the ROLLUP aggregate on our dataset, showing the time a job spend in the various phases of a MapReduce computation. Clearly, combiners play an important role for the Vanilla approach: they are beneficial in the shuffle and reduce phases. When combiners cannot be used (e.g. because the aggregation function is not algebraic), the IRG algorithm outperforms the Vanilla approach. With combiners enabled, the IRG algorithm is slower (larger runtimes) than the Vanilla approach: this can be explained by the lack of parallelism that characterizes IRG, wherein a single reducer is used as opposed to 20 reducers for the Vanilla algorithm. Note that, in the following experiments, combiners are systematically enabled. Finally, Figure 6 confirms that the IRG approach moves algorithmic complexity from the map phase to the reduce phase.

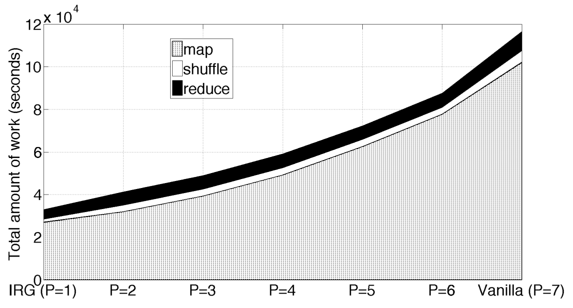
Baseline algorithms. In Figure 7(a) we compare the runtime of Vanilla, IRG, and the hybrid Vanilla + IRG approach. In our experiments we study the impact of the pivot position P, in lights of the "nature" of the input dataset: we synthetically generate data such that they span 1, 10 and 40 years worth of traffic logs.[4]

Clearly, IRG (which corresponds to $P = 1$) is the slowest approach in terms of runtime. Indeed, using a single reducer incurs in prohibitive I/O overheads: the amount of data shuffled into a single reducer is too large to fit into mem-

[4]Note that the size – in terms of number of tuples – of the input data is kept constant, irrespectively of the number of represented years.



(a) Runtime



(b) Amount of work

Figure 7: Comparison of baseline approaches.

ory, therefore spilling and merging operations at the reducer proceed at disk speeds. Although no redundant computations are carried out in IRG, I/O costs outweigh the savings in computations.

A hybrid approach ($2 \leq P \leq 6$) outperforms both IRG and Vanilla algorithms, with runtime as little as half that of the Vanilla approach. Communication costs make the runtime grow slowly as the pivot position moves towards finer granularities, suggesting that in doubt, it is better to position the pivot to the right (increased communication costs) rather than to the left (lack of parallelism). In our case, where a maximum of 20 reduce tasks can be scheduled at any time, our results indicate that P should be chosen such that N_P is larger than the number of available reducers. As expected, experiments with data from a single year indicate that the pivot position should be placed further to the right: the hybrid approach with $P = 2$ essentially performs as badly as the single-reducer IRG.

Now, we present our results under a different perspective: we focus on the *total amount of work* executed by a ROLLUP aggregate implemented according to our baseline algorithms. We define the total amount of work for a given job as the sum of the runtime of each of its (map and reduce) tasks. Figure 7(b) indicates that the IRG approach consumes the least amount of work. By design, IRG is built to avoid redundant work: it has minimal replication rate, and the single reducer can produce ROLLUP aggregates with a single pass over its input.

As a general remark, that applies to all baseline algorithms, we note that the total amount of work is largely determined by the map phase of our jobs. The trend is tangible as P moves toward finer granularities: despite communication costs (the shuffle phase, which accounts for the replication rate) do not increase much with higher values of P thanks to the key role of combiners, map tasks still need to materialize data on disk before it can be combined and shuffled, thus contributing to a large extent to higher amounts of work.

Alternative Hybrid Approaches. We now give a compact representation of our experimental results for variants of the Hybrid approach we introduce in this work. Figure 8 offers a comparison, in terms of job runtime, of the Hybrid Vanilla + IRG approach to the Hybrid IRG + IRG and the Chained IRG algorithms. For the sake of readability, we omit from the figure experiments corresponding to $P = 1$ and $P = 7$.

Figure 8 shows that the job runtime of the Hybrid Vanilla + IRG algorithm is sensitive to the choice of the pivot position P. Despite the use of combiners, the Vanilla "component" of the hybrid algorithm largely determines the job runtime, as discussed above. The IRG + IRG hybrid algorithm obtains lower job runtime and is less sensitive to the pivot position, albeit $3 \leq P \leq 5$ constitutes an ideal range in which to place the pivot. The best performance in terms of runtime is achieved by the Chained IRG approach: in this case, the amount of data shuffled through the network (aggregated over each individual job of the chain) is smaller than what can be achieved by a single MapReduce job. We further observe that placing P towards finer granularities contributes to small job runtime: once an appropriate level of parallelism can be achieved in the first job of the chain, the computation cost of the second job in the chain is negligible, and the total amount of work (not shown here due to space limitations) is almost constant and extremely close to the one for IRG.

We can now summarize our findings as follows:

- All the approaches that we examined greatly benefit from the, fortunately common, property that aggregation functions are algebraic and therefore enable combiners and re-using partial results. If this is not the case, approaches based on the IRG algorithm are preferable.

- If total amount of work is the metric to optimize, IRG is the best solution because it minimizes redundant work. If low latency is also required, hybrid approaches offer a good trade-off, provided that the pivot position P is chosen appropriately.

- Our alternative hybrid approaches are the best performing solutions; both are very resilient to bad choices of the P pivot position, which can therefore be chosen with a very rough a-priori knowledge of the input dataset. Chained IRG provides the best results due to its minimal communication costs. However, chained jobs may suffer from bad scheduling decisions in a heavily loaded cluster, as the second job in the chain may "starve" due to large jobs being scheduled first. The literature on MapReduce scheduling offers solutions to this problem [18].

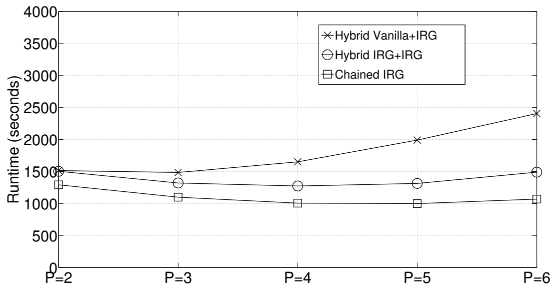


Figure 8: Comparison between alternative hybrid approaches.

6. CONCLUSION & FUTURE WORK

In this paper we have studied the problem of the efficient computation of ROLLUP aggregates in MapReduce. We proposed a modeling approach to untangle the available design space to address this problem, by focusing on the trade-off that exists between the achievable parallelism and communication costs that characterize the MapReduce programming model. This was helpful in identifying the limitations of current ROLLUP implementations, that only cover a small portion of the design space as they concentrate solely on parallelism. We presented an algorithm to meet the lower bounds of the communication costs we derived in our model, and showed that minimum replication can be achieved at the expenses of parallelism. In addition we presented several variants of ROLLUP implementations that share a common trait: a single parameter (the pivot) allows tuning the parallelism vs. communication trade-off for finding a reasonable "sweet spot".

Our work was enriched by an experimental evaluation of several ROLLUP implementations. The experimental approach revealed the importance of optimizations currently available in systems such as Hadoop, which could not be taken into account with a modeling approach alone. Our experiments showed, in addition to the performance of each ROLLUP variant in terms of runtime, that the efficiency of the new algorithms we designed in this work was superior to what is available in the current state of the art.

Our plan is to extend our experimental evaluation to consider skewed datasets: we believe that our hybrid algorithms exhibit the distinguishing feature that the pivot position can be used not only to gauge parallelism and replication, but also to mitigate the possible uneven computational load distribution when data is not uniform. We also consider a data-dependent pivot, which is an even more refined pivot than our current schema-dependent one. Furthermore, we plan to extend our work by designing an automatic mechanism to select an appropriate pivot position, depending on the nature of the data to process.

Acknowledgments

The authors would like to thank Antonio Barbuzzi for his valuable comments. This work has been partially supported by the EU project BigFoot (FP7-ICT-317858).

7. REFERENCES

[1] `http://hadoop.apache.org`.
[2] `http://hive.apache.org`.
[3] `http://pig.apache.org`.
[4] F. N. Afrati et al. Upper and lower bounds on the cost of a map-reduce computation. In *VLDB*, 2013.
[5] F. N. Afrati and J. D. Ullman. Optimizing Multiway Joins in a Map-Reduce Environment. *IEEE Transactions on Knowledge and Data Engineering*, 2011.
[6] S. Agarwal et al. On the Computation of Multidimensional Aggregates. In *VLDB*, 1996.
[7] S. Bellamkonda et al. Adaptive and Big Data Scale Parallel Execution in Oracle. In *VLDB*, 2013.
[8] K. Beyer and R. Ramakrishnan. Bottom-up computation of sparse and iceberg cubes. In *ACM SIGMOD*, 1999.
[9] S. Blanas et al. A comparison of join algorithms for log processing in MapReduce. In *ACM SIGMOD*, 2010.
[10] J. Dean and S. Ghemawat. MapReduce : Simplified Data Processing on Large Clusters. In *ACM OSDI*, 2004.
[11] M. Fang et al. Computing Iceberg Queries Efficiently. In *VLDB*, 1998.
[12] J. Gray et al. Data Cube : A Relational Aggregation Operator Generalizing Data Cube : A Relational Aggregation Operator Generalizing Group-By , Cross-Tab , and Sub-Totals. *Data Mining and Knowledge Discovery*, 1997.
[13] J. Hah, J. Pei, and G. Dong. Efficient Computation of Iceberg Cubes with Complex Measures. In *ACM SIGMOD*, 2001.
[14] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *ACM SIGMOD*, 1996.
[15] P. Jayachandran. Implementing RollupDimensions UDF and adding ROLLUP clause in CUBE operator. PIG-2765 JIRA.
[16] A. Nandi et al. Distributed cube materialization on holistic measures. In *IEEE ICDE*, 2011.
[17] A. Okcan and M. Riedewald. Processing Theta-Joins using MapReduce. In *ACM SIGMOD*, 2011.
[18] M. Pastorelli et al. HFSP: Size-based scheduling for hadoop. In *IEEE BigData*, 2013.
[19] K. A. Ross and D. Srivastava. Fast computation of sparse datacubes. In *VLDB*, 1997.
[20] T. White. *Hadoop - The Definitive Guide: Storage and Analysis at Internet Scale.* O'Reilly, 2012.