An Efficient MapReduce Cube Algorithm for Varied Data Distributions

Tova Milo
Tel-Aviv University
Tel-Aviv, Israel
milo@post.tau.ac.il

Eyal Altshuler
Tel-Aviv University
Tel-Aviv, Israel
eyalalts@post.tau.ac.il

ABSTRACT

Data cubes allow users to discover insights from their data and are commonly used in data analysis. While very useful, the data cube is expensive to compute, in particular when the input relation is very large. To address this problem, we consider cube computation in MapReduce, the popular paradigm for distributed big data processing, and present an efficient algorithm for computing cubes over large data sets. We show that our new algorithm consistently performs better than the previous solutions. In particular, existing techniques for cube computation in MapReduce suffer from sensitivity to the distribution of the input data and their performance heavily depends on whether or not, and how exactly, the data is skewed. In contrast, the cube algorithm that we present here is resilient and significantly outperforms previous solutions for varying data distributions. At the core of our solution is a dedicated data structure called the *Skews and Partitions Sketch* (SP-Sketch for short). The SP-Sketch is compact in size and fast to compute, and records all needed information for identifying skews and effectively partitioning the workload between the machines. Our algorithm uses the sketch to speed up computation and minimize communication overhead. Our theoretical analysis and thorough experimental study demonstrate the feasibility and efficiency of our solution, including comparisons to state of the art tools for big data processing such as Pig and Hive.

1. INTRODUCTION

Data cube [23] is a powerful data analysis tool, allowing users to discover insights from their data by computing aggregate measures over all possible dimensions. Imagine an analyst that is given a database relation describing products sold by a company in various cities in the world over the years. Using the data cube, it is possible to group the data by every combination of attributes and compute the aggregate over the different groups (e.g. product, year, location, and subsets thereof) and discover interesting trends as well as anomalies.

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA

 ISBN 978-1-4503-3531-7/16/06...$15.00

DOI: http://dx.doi.org/10.1145/2882903.2882922

While very useful, the data cube is expensive to compute, in particular when the input relation is large. Consequently, much research has been devoted for developing efficient algorithms for cube computation [15, 22, 31, 36, 30]. Such efficient computation becomes even more intricate and more critical in a big data scenario, where information is spread over many machines in dedicated platforms employed for enabling parallel computation over huge amounts of data. New cube algorithms that best exploit the properties of these platforms must be developed. Our work focuses on one such popular platform - MapReduce [19]. *We present here a new efficient algorithm for computing data cubes over large data sets in MapReduce environments and show that it consistently performs better than the previous solutions.*

Before describing our results, let us briefly explain how the MapReduce framework operates and what are the weaknesses of previously developed cube algorithms for this framework. MapReduce programs use two functions, map and reduce, that are executed in a cluster in two phases. In the first phase, all machines run the map function and generate intermediate data which is delivered to the appropriate reducers. Then, in the second phase, all machines run the reduce function on their input to compute output. A MapReduce algorithm is typically built using a series of such MapReduce rounds. The key difficulty in efficient programming in MapReduce is to minimize network traffic between the machines while at the same time balancing their workload. This is particularly challenging in cube computation because some of the aggregated groups as well as the computed cube itself may be very large and thus balancing computation and avoiding the generation of large intermediate data is not trivial.

Several algorithms for data cube using MapReduce have been developed, e.g. [26, 8, 33, 25]. Some have even been implemented in Pig [5] and Hive [4] - engines that provide a database interface over MapReduce and allow users to query data in an SQL-like syntax. For instance, Pig implements the data cube algorithm from [26]. However, to our knowledge, all previous solutions, including those in Pig and Hive, suffer from sensitivity to the distribution of the input data and their performance heavily depends on whether or not, and how exactly, the data is skewed. Intuitively, skews yield large groups that need to be aggregated. For example, if an extremely large number of laptops were sold in 2012, they may not all fit (e.g. for their costs to be aggregated) in a single machine's main memory. Managing the aggregation of such big groups alongside many smaller ones is challenging. While some of these algorithms, e.g. [26], make an attempt

to handle skews, the solution that they propose (to be explained later) is insufficient, and still existence of skewed groups affects the performance.

In contrast, our new cube algorithm gracefully handles mixtures of skewed and non skewed groups and consistently outperforms previous solutions for varying data distributions. The key idea underlying our solution is that to optimize performance, data must be analyzed at the *group* granularity. Note that by the definition of the cube, every subset of tuples that agree on the value of some group-by attributes contributes (after aggregation of its measure attribute) one tuple to the cube. We call each such tuple a *cube group* (c-group for short). While skewed c-groups have many tuples belonging to them, this large data is "compressed" by the aggregation and yields relatively small output. On the other hand, non-skewed c-groups are each small, but the cube computation may generate very many of them. To reduce network traffic it is thus beneficial to perform as much of the aggregation of skewed c-groups already at the map phase, whereas for non skewed ones it is better to postpone their materialization to the reduce phase. Our algorithm employs careful optimization to factorize, when possible, the computation across multiple c-groups, determine which c-groups should be computed where, and partition the work in a balanced manner among the available machines.

To support this we have developed a novel data structure called the *Skews and Partitions Sketch* (SP-Sketch for short). The SP-Sketch has two nice properties. On the one hand, it is very compact in size and fast to compute. On the other hand, it captures all needed information for identifying skewed c-groups and for effectively partitioning the processing of skewed and non skewed c-groups between the machines.

To understand the novelty of our approach, let us briefly contrast with the state of the art algorithm in [26]. The algorithm in [26] also uses sampling to obtain information about the data properties. However, a major disadvantage is that it makes a decision about the existence of skews *at the granularity of a full cuboid*. If a skewed group is detected, it aborts computation for the cuboid that contains this group, and recursively splits the cuboid. It then checks again, at each recursive iteration, for skews *at the granularity of the full given partition*, and if detected aborts the computation and splits the data again, and so on. The number of rounds thus depends of the skewness level, which makes the algorithm sensitive to data distribution. In contrast, our novel approach, and the use of SP-Sketch, allows us to determine, *in a single round*, skewness *at the c-group granularity*, for all c-groups in all cuboids. Neither recursion nor aborts are thus required by our algorithm, making it faster and resilient to any data distribution. The crux of our efficient solution is that we managed to prove that the for practical settings the number of skewed c-groups is in fact bounded and information on all of them can be kept simultaneously in main memory. While working at the c-group granularity was indeed mentioned in [26] as a desirable future direction that may allow for performance improvement, it is only this result of ours that made this idea practically feasible. The SP-Sketch, our use of it for data partitioning, the particular split of work between mappers and reducers, as well as factorized processing *at the c-group granularity* in the reducers, are all novel ideas.

Our main contributions are the following:

Formal Model. Our first contribution is a formal model defining the notion of (skewed) c-groups. We highlight the challenges that must be addressed by an efficient cube algorithm by analyzing a naive MapReduce algorithm. We formally define the notion of (skewed) c-groups and show why skewed c-groups need to be specially treated and why the relationship between c-groups must be exploited to avoid redundant network traffic.

SP-Sketch. Our second contribution is a novel data structure called SP-Sketch, that summarizes the important information for efficiently computing the cube. We start by presenting a utopian view of the SP-sketch, which is too expensive to compute. Then we describe an algorithm for building an approximated variant of the SP-Sketch. The algorithm is based on data sampling and is used to efficiently compute an approximated variant of the SP-Sketch. *We prove the sketch to have high accuracy as well as being small enough to entirely fit in a single machine main-memory.*

SP-Cube Algorithm. Our third contribution is an efficient algorithm that utilizes the previosly computed SP-Sketch, to efficiently split work between mappers and reducers: mappers will perform partial aggregation of skewed c-groups and determine which non-skewed c-groups may be processed together (and by which reducer). Reducers then process efficiently their assigned non-skewed c-groups and globally aggregate the (partially aggregated) skews.

Theoretical analysis. Our fourth contribution is a theoretical demonstration of the efficiency of our algorithm in terms of the size of machines' memory and the intermediate data transferred between the mappers and the reducers. We show that in an extreme (synthetic) case the amount of data that is transferred may be exponential in the number of cube dimensions. However, we prove that in common cases the size of the transferred data is only polynomial in the number of dimensions thereby enabling efficient processing. Regarding memory, we prove that the SP-Cube workload is balanced between the machines.

Experimental Analysis. Our fifth contribution is a thorough experimental analysis that matches our theoretical results. We experiment with two real-life data sets as well as synthetic data and examine the various steps of our algorithm and its performance as a whole. We compare the performance of SP-Cube to existing algorithms implemented in Pig and Hive. Working on varying data distributions, we show that SP-Cube consistently outperforms other algorithms, achieving between 20%-300% better running time and smaller network traffic.

Throughout the paper, assume the aggregate function to be applied is *count*, i.e the cardinality of every c-group . In Section 7 we discuss different types of aggregate functions to charactierize our algorithm in the general case.

Outline. We start in Section 2 by providing the necessary background and definitions. Section 3 then highlights the challenges in cube computation via a naive cube algorithm. Section 4 presents the SP-Sketch and analyzes it. Section 5 then describes our SP-Cube algorithm that uses the sketch and studies its properties. Our experimental study is presented in Section 6 and related work is discussed in Section 7. Finally, we conclude is Section 8. For space constraints, some proofs and experiments are deferred to the Appendix.

2. PRELIMINARIES

We start by providing the basic definitions for data cube and the additional notions that will be used in the rest of the paper.

2.1 Data Cube, Cuboids, and Cube Groups

Given a domain $\mathcal{A}$ of attribute names, consider a relation $R(A_1, A_2, \ldots, A_d, B)$ with a set $A = \{A_1, \ldots, A_d\} \subseteq \mathcal{A}$ of attributes called *dimensions*, and an additional disjoint attribute $B \in \mathcal{A}$ called the *measure* attribute. W.l.o.g we will assume that the set of attributes names in R is ordered, and denote a tuple in R by $t = (a_1, \ldots, a_d, b)$, meaning that the value of attribute A_i (resp. B) in t is a_i (b). We also assume that the measure attribute takes a numeric value. Finally, we assume that the value of the attributes (as well as computed aggregates over the measure attribute) can fit in a fixed number of memory bytes, and take this as a constant in our complexity analysis.

We will often be interested in only a subset $A' \subseteq A$ of the dimensions, and then replace the attribute names of R that are not in A' by $*$. Similarly, we will often be interested in the projection of a tuple t to a subset A' of the dimensions. We will then replace the value of the dimension attributes not in A' by $*$ and omit the measure value b.

EXAMPLE 2.1. *As a simple running example, we consider a relation R describing the products sold by a company in various cities in Europe over the years. R has three dimension attributes: $A_1 = name$, $A_2 = city$, $A_3 = year$, and a measure attribute $B = sales$. A tuple in R records the number of sales for a given product name, in a specific city and year. For instance, the tuple $t = (laptop, Rome, 2012, 2000)$ describes the fact that that 2000 laptops where sold in Rome at 2012. The projection of t to the dimensions name and year is denoted $(laptop, *, 2012)$.*

First introduced in [23], the *data cube* of a relation R is a set of relations, capturing of all possible group-by's that can be computed over a subset $A' \subseteq A$ of the dimensions of R, w.r.t some given aggregate function. Examples of common aggregate functions include *sum*, *count*, and *max*.

The result of each such group-by is given in a separate table called a *cuboid*. We often overload notation and denote a given cuboid by the set A' of dimensions on which its group-by was performed. Each subset of tuples in R that agree on the value of the group-by attributes contributes (after aggregation of its measure attributes) one tuple to the cuboid. We call each such tuple a *cube group* (c-group for short). For a given c-group g, we will often be interested only in the values of its dimension attributes, and will then denote g by its projection to these attributes. Finally, we refer to the set of tuples of R that was grouped together to generate g as *the set of* g, denoted $set(g)$. We say that a tuple t in R contributes to a cube group g if $t \in set(g)$. Note that by definition each tuple t contributes to multiple such cube groups, each corresponding to a projection over some subset of its dimension attributes.

EXAMPLE 2.2. *To continue with our running example, the data cube of R consists of 8 cuboids, including for instance the cuboids $C_1 = (name, *, year)$ and $C_2 = (*, *, *)$. The cuboid C_1 is obtained by grouping the tuples in R by product name and year, applying the aggregation function to the measure attribute of each group. Two c-groups in C_1 may be $c_1 = (laptop, *, 2012)$ and $c_1' = (laptop, *, 2015)$. c_1 (resp. c_1') is generated by aggregating the set of tuples $set(c_1)$ (resp. $set(c_1')$) of R that includes all tuples describing laptop sales in 2012 (2015). The cuboid C_2 consists of a single value c_2 obtained by aggregating the measure attribute of the all tuples in R. Here $set(c_2)$ consists of all the tuples in R. Note that the tuple $t = (laptop, Rome, 2012, 2000)$ contributes to both c_1 and c_2.*

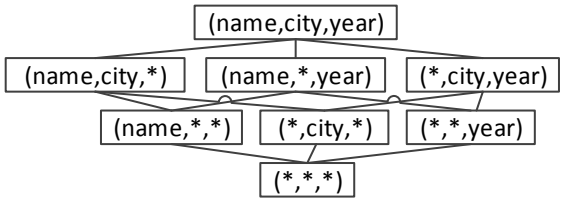


Figure 1: Cube lattice

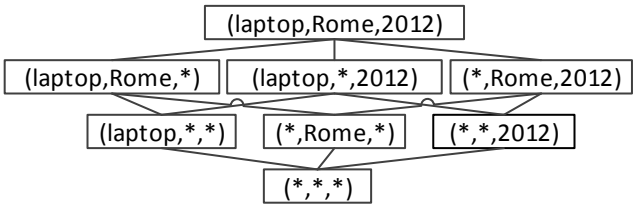


Figure 2: Tuple lattice

2.2 The Cube and Tuple Lattices

Inspired by [12] we employ here two notions of a lattice graph - the *cube lattice* and the *tuple lattice*, that capture respectively the relationships between different cuboids and cube groups.

DEFINITION 2.3 (THE CUBE LATTICE). *Given a relation R, the nodes of the cube lattice - lattice(R) are the cuboids of R. In this lattice, a cuboid C' is a descendant of a cuboid C iff its set of group-by attributes is obtained from that of C by omitting one attribute. We say that C is an ancestor of C' iff C' is a descendant of C.*

An example of the cube lattice for the relation in our running example is given in Figure 1.

DEFINITION 2.4 (THE TUPLE LATTICE). *Given a tuple t in R, the nodes of the tuple lattice - lattice(t) are all possible projections of t on subsets of the dimension attributes. A projection t'' is a descendant of a projection t' iff t'' is obtained from t' by omitting one attribute. t' is an ancestor of t'' iff t'' is a descendant of t'.*

Consider the tuple $t = (laptop, Rome, 2012, 2000)$. The lattice for t is given in Figure 2. Note that the nodes in the tuple lattice correspond precisely to the c-groups to which t contributes.

Two simple observations on theses lattices will be helpful in the sequel. The first, used already in previous work [26], concerns the cube lattice.

OBSERVATION 2.5. *For each cuboid C in the lattice and each descendant C' of C, C can be easily derived from the tuple sets of the cube groups in C' by partitioning the tuples in each set w.r.t the added attribute of C', then generating one aggregated tuple per partition.*

This observation is the basis for the traditional BUC data cube algorithm [15], which processes the lattice bottom up, computing each cuboid from one of its descendants. The specific descendant from which each cuboid is computed is chosen using some heuristics for optimizing performance.

We observe here that an analogous situation holds for the tuple lattice.

OBSERVATION 2.6. *For every c-group g in the tuple lattice and every descendant g' of g, the set of tuples that is aggregated for generating g, is a subset of the set used to generate g'. Namely, $set(g) \subseteq set(g')$.*

This observation will be useful in our group-focused algorithm. Recall that each node in $lattice(t)$ corresponds to some c-group g to which t contributes. If all tuples that contribute to a given cube group g (tuples in $set(g)$) are sent to the same machine (e.g. the tuples in $set((laptop, *, *))$), the machine can also use them to compute locally the ancestors c-groups in the lattice ($(laptop, *, 2012)$, $(laptop, Rome, *)$, and $(laptop, Rome, 2012)$), using e.g. the BUC algorithm, applied locally to the given tuples subset.

2.3 MapReduce Settings

We specify here the MapReduce cluster settings that will be used in the rest of the paper. Consider a relation R with n tuples and $d+1$ attributes. Working in a distributed environment, suppose we have k machines, and assume each one can run a single map or a single reduce function in every map or reduce phase, respectively. We assume that the n tuples of the input are equally loaded to the machines at the beginning of the algorithm. We additionally assume that the machines have a main memory that is in the order of their input size. We mark $m = \frac{n}{k}$ and assume that a machine main memory size is $O(m)$. In addition, we assume that all machines share a distributed file system, in which R is read from and to which the output data cube will be written. To conclude this section, we formally define skewed c-groups in terms of their freuquency relative to a machine memory size.

DEFINITION 2.7. *a c-group g is* skewed *if the cardinality of its tuples set is larger than m, namely $|set(g)| > m$.*

Note that the threshold of skewed c-groups depends on a machine memory size, m. For performance considerations, we generally want c-groups to be computed in main memory. As skewed c-groups do not fit in a single machine main memory, efficient treatment for them is challenging, and we explain later in the paper how our novel algorithm overcomes these difficulties.

3. GUIDELINES FOR EFFICIENT CUBE COMPUTATION WITH MAPREDUCE

Before presenting our algorithm for cube computation, let us first consider a naive basic MapReduce-based algorithm, and then use it to highlight the challenges addressed by our solution. Note that the naive algorithm we present has been improved by previous work, but our goal here is not to serve as baseline but rather, because of its simplicity, for highlighting the challenges that a good algorithm needs to address.

3.1 MapReduce Cubing - Naive Approach

We assume that the tuples of the relation R are read from a distributed file system and are equally split among the given set of mappers in an arbitrary manner. A pseudo code of the naive cube algorithm is shown in Algorithm 1.

Algorithm 1: Naive MapReduce Cubing Algorithm

```
1 Map(t)
2   groups = Nodes(lattice(t))
3   measure = measure-attribute(t)
4   for g ∈ groups do
5     emit(g,measure);
6   end
7 Reduce(g,values)
8   result = agg(values)
9   emit(g,result)
```

The algorithm starts by a map phase (lines 1-6) where each tuple t is projected on every subset of its dimensions. This is done by constructing the tuple lattice $lattice(t)$ and retrieving its nodes, which are precisely these projections (line 2). For each such projection, a (key,value) pair is emitted (line 5), where the key is the projected tuple and the value is the measure attribute of t (retrieved by the $measure - attribute$ function). All pairs are then sent to a reducer that is in charge on all tuples with the same key. Observe that by this construction all tuples belonging to a given cube group are sent to the same reducer. The specific reducer for each group is implicitly chosen by the MapReduce framework, by hashing the key (c-group value).

Next, in the reduce phase (lines 7-9), for every c-group, the reducer that received its corresponding tuples applies the aggregate function on the set and writes the resulting tuple (the c-group and its corresponding aggregate value) - back to the distributed file system (line 9). Note that for simplicity of presentation, the algorithm writes the c-groups to the distributed file system in an arbitrary order. If one wishes to generate one file per cuboid, the code can be slightly modified so that tuples of distinct cuboids are written to distinct files (with the files generated by the different reducers concatenated to form the full cuboid). We omit this here.

Let us discuss the problems with this naive algorithm.

3.2 Skews

The first problem with the naive algorithm is its sensitivity to heavy skews in the data. Consider a relation R where many tuples agree on the value of some subset of the attributes. For instance, suppose that many tuples have the value $(*, Paris, 2010)$ when projected on the *city* and *year* attributes. If the cube group for $(*, Paris, 2010)$ is larger than what can fit in a reducer's main memory, the corresponding reducer will not be able to perform the aggregation in main-memory and performance will suffer.

As we have stated, m denotes the machines' partial input size. Recall that a c-group g is *skewed* if the cardinality of its tuples set is larger than m. Since skewed groups cannot fit in the reducer main memory, the computation in the reduce phase will involve I/Os between main-memory and disk, making the overall computation slower. To avoid this delay, our optimized algorithm detects all skewed groups and partially aggregates them already at the map phase, before being sent to the relevant reducer. Interestingly, we will show that the number of skewed groups is not large, and thus this partial aggregation does not weigh heavily on the

mappers. The formal upper bound for the number of skewed groups is proved in Section 4.

3.3 Load Balancing

In the MapReduce framework, work is distributed among reducers by implicitly applying some partitioning function to the *key* of each $(key, value)$ pair and directing it to the resulting reducer. This can be a generic hash function, but also a pluggable customized one. Whether the resulting load is balanced or not depends on the compatibility of the partitioning function and the keys distribution. To alleviate the sensitivity to data distribution and assure that work is always evenly distributed, our optimized algorithm plugs a partitioning function that exploits the lexicographical order of cube groups. Intuitively, for a given cuboid, the relation tuples are (virtually) partitioned into k partitions of equal size, k being the number of reducers, such that the tuples in partition i, when projected on the cuboid dimensions, are lexicographically smaller than those of partition $i+1$. The (projected) tuples of partition i are then assigned to reducer i. We will explain in the next two sections how (together with skews detections) this partitioning is efficiently achieved and exploited by our algorithm for load balancing the reducers work.

3.4 Network Traffic

The naive algorithm treats each c-group independently and does not exploit the relationships between different groups. Indeed, for each of the n tuples of the input relation, Algorithm 1 generates 2^d projections, d being the number of dimensions, yielding an overall number of $n \cdot 2^d$ key-value pairs, sent over the network to the relevant reducers. As we will show in Section 5, much of this redundant communication may be avoided by exploiting the Observation 2.6. Intuitively, projections whose corresponding cube groups may be computed from descendants in the tuple lattice, need not be resent and instead can be computed by the reducer assigned to the smallest (non-skewed) descendant. We will explain this in more details in Section 5.

4. THE SP-SKETCH

We present our optimized cube algorithm in two steps. First we describe in this section the SP-Sketch data structure used by our algorithm. The sketch contains information that allows to (1) identify *skewed* cube groups (in section 3.2 we defined that a group g is skewed if $|set(g)| > m$) and (2) *partition* the relation tuples for balancing the reducers workload. Hence the name SP-sketch. Then, in the following section we explain how the SP-sketch is used by our cube algorithm to ensure efficient computation with reduced network traffic.

We start by presenting a utopian view of the SP-sketch, which is too expensive to compute. Then we describe an approximated yet accurate enough variant, that can be efficiently computed. We note that the SP-Sketch captures the properties of the input relation and is independent of the aggregate function to be used. Consequently, once constructed, the same SP-Sketch can be used to efficiently compute multiple aggregated functions.

4.1 Data Partitioning

We first define some useful auxiliary notions. Consider a relation R and a cuboid C of R. For two tuples t_1, t_2 in R, we say that $t_1 <_C t_2$ (resp. $t_1 =_C t_2$) if, when restricted to the dimension attributes of C, t_1 is lexicographically smaller than t_2 (resp. equals t_2). We denote by $sorted(R, C)$ a *sorted* version of R where the tuples are ordered w.r.t $<_c$ (equal tuples are ordered arbitrarily). Let n be the number of tuples in R, let k be the number of available machines and let m be the size of a machine's memory. Recall that we assume that $m \geq \frac{n}{k}$, namely that the relation tuples can altogether fit in the memories of the given k machines. We can partition the tuples in R into k subsets, using the $k-1$ *partition elements* in $sorted(R, C)$. We now define the partition elements of every cuboid.

DEFINITION 4.1. *The* partition elements *of R w.r.t C are the tuples in positions $i\frac{n}{k}$, $i = 1 \dots k-1$, in $sorted(R, C)$.*

Given partition elements $t_1, \dots, t_{k-1}$, the first partition contains all tuples t s.t. $t \leq t_1$. The i^{th} partition, $i = 2 \dots k-2$, contains the tuples t s.t. $t_i < t \leq t_{i+1}$. Finally, the k^{th} partition contains the tuples t s.t. $t_{k-1} < t$. The resulting split has two nice properties that will be useful in the sequel.

PROPOSITION 4.2. *The following two properties hold -*

1. *For each non-skewed c-group g in C, all its tuples fall in the same partition, and*
2. *Omitting the members of skewed c-groups , all partitions are of size $O(m)$.*

In our algorithm, each partition (excluding its skewed groups) is assigned to a machine. The first property will guarantee that for each c-group all its tuples will be sent to the same machine, while the second guarantees bounded machines load.

4.2 Building the SP-Sketch

The (utopian) **SP-Sketch** is a graph with a structure similar to the cube lattice described in Section 2. For each cuboid node C in the lattice, the SP-sketch records two items: The first, denotes $skews(C)$, records the set of skewed c-groups in this cuboid. The second, denoted $partition-elements(C)$ records the partitions elements of R w.r.t C.

EXAMPLE 4.3. *Figure 3 depicts the SP-Sketch for our running example. It details part of the information kept for three of the cuboids. For each of the three cuboids we see (part of) its skewed c-groups and partition elements. Note that as the cuboid $(*, *, year)$ is a descendant of the cuboid $(name, *, year)$, all instances of a skewed c-group in the latter are also instances of the corresponding c-group in the former, making it skewed as well. For instance, the groups $(keyboard, *, 2009)$, $(printer, *, 2011)$ and $(television, *, 2012)$ are skewed c-groups in the cuboid $(name, *, year)$, and thus $(*, *, 2009)$, $(*, *, 2011)$ and $(*, *, 2012)$ are also skewed in the cuboid $(*, *, year)$ (which also as an additional skewed c-group - $(*, *, 2014)$).*

A naive but too expensive algorithm for building the SP-Sketch would build $sorted(R, C)$ for all cuboids, then derive from the sorted lists the skewed cube groups and the partition elements.

Instead, we adopt here sampling techniques from [32] and [26] to build an approximated version to the SP-sketch. We

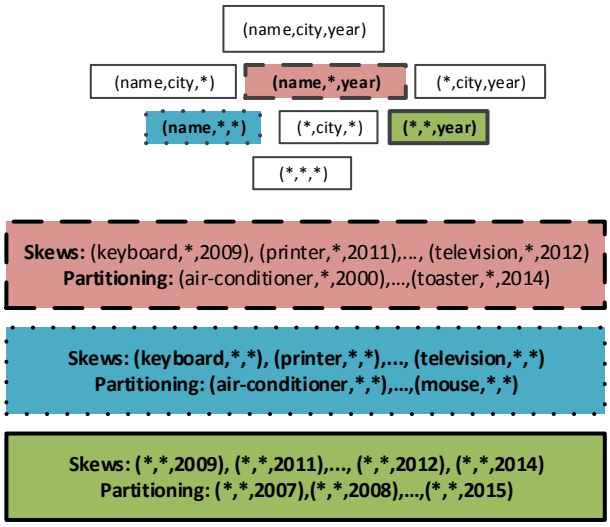


Figure 3: The SP-Sketch for our running example

sample tuples from R with probability α (to be defined below) for each tuple, independently of the others. Then, we build the SP-sketch based on this data sample. The sampling technique in [32] is a standard technique for sketching a data using uniform sampling. However, our use of the sample is novel. We use the sample to detect the skewed groups and partition elements of each cuboid (for building the SP-Sketch), whereas [32] uses the sketch to find bucketing elements for sorting. Algorithm 2 is a MapReduce-based implementation of this approximation algorithm, which we run in the first round in our cube algorithm (to be presented in the next section).

In the map phase (lines 2-5) the relation tuples are loaded from the distributed file system, equally split among the k mappers. Each mapper samples its tuples, and each tuple is taken into the sample independently of the others, with probability $\alpha = \frac{1}{m}ln(nk)$ (α is the probability to pass the *if* test in line 4). The specific value of $\alpha = \frac{1}{m}ln(nk)$ is chosen for ensuring a small sample size, for in-memory computations. However, we need the sample to be as informative as possible, as we use it to build the sketch. Therefore, it should contain enough tuples from the original data. Our mathematical development (see the proofs of Propositions 4.4, 4.5, and 4.6) have led us to derive that this chosen value of α achieves this desired tradeoff. The entire sample is then delivered to a single reducer that builds the SP-sketch (lines 7-10). Namely, this MapReduce algorithm uses k machines in the map phase, but only one is needed in the reduce phase. In the reduce phase, the SP-Sketch is built over the sample. The invoked *build-sketch* procedure (not detailed in the figure) implements a brute-force approach for building the SP-sketch, using in-memory computation, as follows (We will prove later that the sample is small enough to allow this):

Skews: For determining the skewed c-groups , we compute a cube over the sample and employ *count* as the aggregate function.[1] We then record as skewed the c-groups whose count value is larger than $\beta = ln(nk)$. Our choice of β is justified using the following tradeoff. If its value is too large, we might miss some skewed groups. If it is too small, we might consider too many groups as skewed, and the sketch would be too large to fit in a machine main memory. Our particular choice of β was designed such the SP-Sketch successfully captures all skewed groups in the cube (with high probability), but is still small to fit in all machines main memory. Its specific chosen value is formally justified in the proof of Proposition 4.5.

[1]Our implementation employs here the classic BUC algorithm [15] but any cube algorithms will do.

Algorithm 2: Approximated SP-Sketch

```
1  // k mappers
2  Map(t)
3  |  pick at random α ∈ [0,1]
4  |  if α ≤ (1/m)ln(nk) then
5  |  |  emit(0,t);
6  // 1 reducer, only 0 key, values - sampled tuples
7  Reduce(key,values)
8  |  sample = values
9  |  sketch = build-sketch(sample)
10 |  emit(0,sketch)
```

Partitions: For determining the partition elements, we sort the sampled tuples w.r.t to each cuboid C, and compute for each sorted list its $k-1$ partition elements: If n' is the number of samples, then the partition elements here are the tuples in positions $i\frac{n'}{k}$, $i = 1 \ldots k-1$.

Once computed, the SP-Sketch is stored in the distributed file system (to be later cached, in the second MapReduce phase of our cube algorithm, by all machines).

4.3 SP-Sketch Properties

We conclude this section by analyzing the size of the sample and the generated sketch, as well as its accuracy.

As mentioned above, the sampling technique that we use is inspired by [32] and [26]. [32] uses sampling to devise efficient sorting in MapReduce, whereas [26] uses samples to determine whether a given cuboid contains some skews. Our use of the samples, as dictated by the needs of the SP-Sketch, is slightly different - we only wish to partition the data and not to fully sort it, and we are interested in identifying all skewed c-groups and not just in determining whether some exist. Nevertheless, we can adapt some of the proofs from [32, 26] to our context, as we show below.

We first show that with high probability, the sample size is small enough to fit entirely in a machine's memory. Recall that we use n to denote the number of tuples in R, k the number of machines and m the size of a machine's memory.

PROPOSITION 4.4. *Assume every tuple is independently taken into the sample with probability $\alpha = \frac{1}{m}ln(nk)$. Then, with probability of at least $1-O(\frac{1}{n})$ the sample size is $O(m)$.*

The proof follows the same simple probabilistic analysis as in [32] where elements are sampled independently with the same probability α. We thus omit this here.

We next show that the SP-sketch generated from the sample has high accuracy. The first proposition shows that the SP-Sketch successfully captures skewed groups.

PROPOSITION 4.5. *Let d be the number of dimensions and assume $2^d \cdot k = O(m)$. Then, with probability of at least $1-O(\frac{1}{k})$, the algorithm detects all skewed groups.*

We note that the assumption that $2^d * k = O(m)$ is reasonable and typically holds in a MapReduce environment:

The memory of a reducer, m, is in the order of Gigabytes, whereas d is a small constant, and k, the number of machines, is in the order of hundreds or thousands.

The second proposition shows that the SP-Sketch successfully captures the partitioning elements of every cuboid.

PROPOSITION 4.6. *Let d be the number of dimensions and assume $2^d \cdot k = O(m)$. Then, with probability of at least $1 - O(\frac{2^d}{n})$, omitting the members of skewed c-groups , all partitions are of size $O(m)$.*

Finally, we show that the computed SP-sketch is small enough and can entirely fit in the main memory of every machine in the cluster.

PROPOSITION 4.7. *Let d be the number of dimensions and assume $2^d * k = O(m)$. Then, with probability of at least $1 - \frac{1}{k}$, the SP-sketch size is $O(m)$.*

5. THE SP-CUBE ALGORITHM

We are now ready to describe our cube algorithm, called *SP-Cube*. The algorithm is composed of two MapReduce rounds. In the first round, the SP-Sketch is built. In the second round the cube is computed efficiently using the sketch. We now explain this second round.

Recall that the SP-Sketch records k partitions for each cuboid, k being the given number of machines. Correspondingly, when sent to reducers, tuples from the i^{th} partition, $i = 1 \dots k$, of any cuboid will be assigned to reducer i. For simplicity of presentation, we assume an additional given reducer, numbered 0, that will be in charge of aggregating the partial aggregates of skewed groups. (Otherwise this work can be assigned to one of the existing reducers).

Note that the SP-Sketch captures all requruired information for the algorithm to decide whether a c-group is skewed or not. This is implemented by maintaining a hash table in which items correspond to the skewed c-groups. The key for each item in the hash table is the (concatenated) value of the dimension attributes of the group. For each such key, the associated value is the c-group partial aggregated value. Then, to know whether a group is skewed or not, the SP-Cube algorithm checks whether the groups's key (concatenated value its the dimension attributes) appears in the table.

5.1 Algorithm Overview

The pseudo-code of our algorithm is depicted in Algorithm 3. We first explain what mappers do, then the reducers.

Map. The mappers process the tuples in the relation R as follows. For each tuple t assigned to the mapper, the tuple lattice $lattice(t)$ is built (line 4). Recall that each node in the lattice corresponds to one cuboid c-group to which the tuple belongs. The nodes in the lattice are traversed bottom up, in BFS (breadth first search) order (line 5). For each unmarked lattice node (initially all nodes are unmarked) we check, in the corresponding cuboid node in SP-Sketch node, whether the tuple's c-group is skewed or not. If it is skewed then we perform local aggregation, adding the tuple's measure value to the c-group (local) aggregated value, and mark the node as processed (lines 6-8). Otherwise, if the c-group is not skewed, we check (again, using the sketch) to which partition the c-group belongs. We send the tuple to the corresponding reducer, mark the node *and all its ancestors and their ancestors recursively* (lines 9-13) as processed, and continue with the next not marked node in the BFS traversal.

Note that if the given c-group is not skewed so are all its ancestors in the tuple lattice. Furthermore, following Observation 2.5, the reducer to which the tuple is sent will have all the needed information not only for computing the given c-group but also all its ancestors. This is why the ancestors are recursively marked as well and skipped (thereby reducing communication overhead).

Finally, once all tuples are processed, the mapper sends its partial aggregates of skewed c-groups to reducer in charge on skewed c-groups (lines 16-20).

EXAMPLE 5.1. *To illustrate the Map phase, let us consider the tuple $t = (laptop, Rome, 2012, 2000)$ and assume that the aggregate function is sum. The tuple lattice of t, depicted in figure 2, is constructed and traversed bottom up in BFS order, starting from the node $(*, *, *)$. As this c-group aggregates the values of all database tuples, it will be found in the SP-sketch as skewed, and thus the mapper performs local aggregation for the c-group and adds t's measure attribute (2000) to the so-far-computed sum. The node is marked and the mapper continues to $(laptop, *, *)$ - the next node in the BFS order.*

Assume this c-group is not skewed and that, according to the $(name, *, *)$ of the SP-Sketch, the tuple belongs to the second partition (as $(laptop, *, *)$ is lexicographically between $(keyboard, *, *)$ and $(printer, *, *)$). Thus it is sent to reducer 2 and $(laptop, *, *)$ as well as its ancestors in the lattice - $(laptop, Rome, *)$, $(laptop, *, 2012)$, and $(laptop, Rome, 2012)$ - are marked as processed.

Next the mapper continues to $(*, Rome, *)$. Assume this c-group is also not skewed. Then the tuple is sent to the reducer in charge of the corresponding partition, and both the c-group $(*, Rome, *)$ and its (non marked so far) ancestor $(*, Rome, 2012)$ are marked. Finally the mapper processes $(*, *, 2012)$. Assume this c-group is skewed (as it appears in the skewed c-groups list of $(*, *, year)$ in Figure 3), and thus the algorithm adds 2000 to the local partial sum maintained for $(*, *, 2012)$.

Once all tuples are processed, local partial sums computed e.g. for $(*, *, *)$, and $(*, *, 2012)$ are sent to reducer 0.

Reduce. We are now ready to describe what reducers do. For skewed c-groups , the responsible reducer aggregates the local aggregated values it received from the mappers (lines 24-27). For instance, to continue with the above example, it sums up, respectively, the partial sums obtained for each of the c-groups $(*, *, *)$ and $(*, *, 2012)$.

Note that for every skewed group, there are at most k values of partially aggregated tuples coming from the k mappers. The actual aggregation computation performed by the reducer depends on the aggregate function. If, for instance, as in the above example, the aggregate function is *sum*, then the reducer should simply compute the sum of the partial sums. For another example, if the aggregate function is *avg*, then the reducer should get from each mapper both the local *sum* and *count* and combine them to compute the global average, summing the partial sums and dividing the result by global sum of the local counts.

For non-skewed c-groups the reducer computes, given the set of tuples of a c-group g, not only the aggregate value for g but also of its ancestor c-groups, using a standard cube

Algorithm 3: Cube computation

```
1  // k mappers, SP-Sketch in main-memory
2  Map(t)
3      for t ∈ mapper-input do
4          L = lattice(t)
5          for g = NextUnmarkedBFS(L) do
6              if skewed(g,SP-Sketch) then
7                  partially-aggregate(g)
8                  mark g
9              else
10                 key = partition(g,SP-Sketch)
11                 emit(key,measure(t))
12                 mark g and its ancestors (recursively)
13             end
14         end
15     end
16     for g ∈ partially-aggregated-groups do
17         key = 0
18         value = Pair(g,aggregate-value(g))
19         emit(key,value)
20     end
21
22 // k+1 reducers , SP-Sketch in main memory
23 Reduce(key,values)
24     if skewed-group-reducer then
25         g = decode-group(values)
26         value = aggregate-func(values)
27         emit(g,value)
28     else
29         key = decode-group(key)
30         compute BUC over ancestors
31     end
32
```

algorithm (we use BUC [15] in our implementation). For instance, the reducer responsible for $(laptop, *, *)$ computes also the aggregated values for the c-groups $(laptop, Rome, *)$, $(laptop, *, 2012)$, and $(laptop, Rome, 2012)$.

Note however that some optimization is required to avoid redundant processing. Some c-groups have common ancestors, and so we should avoid computing these common ancestors multiple times. We thus assign the computation of each c-group to its smallest (in the BFS traversal order) non-skewed descendant. For instance, in our running example $(laptop, rome, *)$ is an ancestor of both $(laptop, *, *)$ and $(*, rome, *)$. $(laptop, *, *)$ preceded $(*, rome, *)$ in our lattice BFS traversal and thus $(laptop, rome, *)$ and is computed as part of the cube computation for $(laptop, *, *)$.

We conclude the presentation of our algorithm by analyzing its efficiency in terms of the size of the intermediate data transferred between the mappers and the reducers.

5.2 Intermediate Data Transfer

By the definition of the cube, the size of the output may be exponential in the number of dimensions, and so is not surprising that in extreme cases the amount of data that is transferred may be exponential as well. However, we show that in common cases the transferred data size is polynomial in the number of dimensions thereby enabling efficient processing (recall that in the Naive algorithm presented in Section 3 the transferred data is exponential in the dimensions number). This will also be confirmed by our experiments on real-life as well as synthetic data. For space constraints, all proofs are deferred to the Appendix.

We start by considering skewed c-groups . The proof of the following proposition (as well as of the other results in this section) can be found in the Appendix.

PROPOSITION 5.2. *Assume $2^d \cdot k = O(m)$. Then, with probability of at least $1 - O(\frac{1}{k})$, the amount of data transferred for computing skewed c-groups is $O(dn)$.*

From now on, assume Proposition 5.2 holds. We next consider non-skewed c-groups. We first show that in the worse case, data transfer may be exponential in the number of dimensions.

THEOREM 5.3. *There exists a relation on which the SP-Cube algorithm generates network traffic size of $\theta(2^d \cdot n)$*

However, such extreme situations are unlikely to occur in practice. We prove that in common cases the transferred data size is polynomial in the number of dimensions.

We start by defining some useful notions. Consider a c-group g. By definition, if g is skewed, then every subset of g must be skewed as well. However, the opposite is not necessarily true. Namely, it may be the case that all subset of g are skewed, whereas g itself is not. We call relations where such situations do not occur **skewness-monotonic**.

DEFINITION 5.4. *A relation R is **skewness-monotonic** if for every c-group g, g is skewed if and only if all of its sub-groups are skewed. For databases without skews, the definition of skewness-monotonic is vacuously true.*

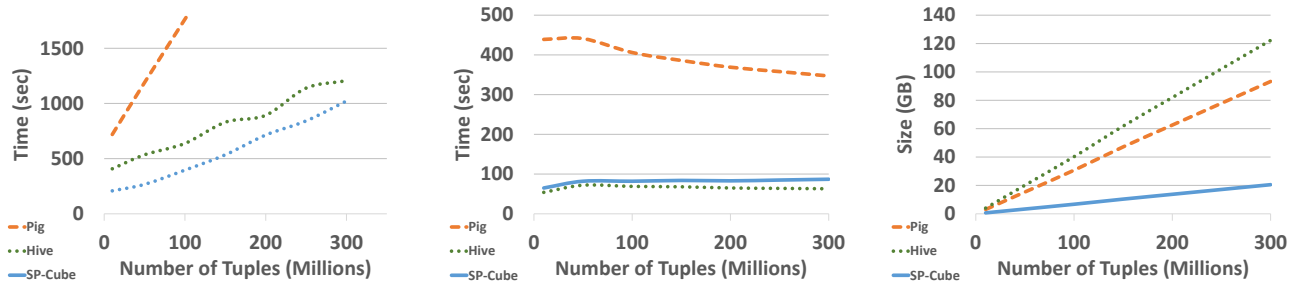
We show the following.

PROPOSITION 5.5. *For every **skewness-monotonic** relation R, the amount of network traffic generated by SP-Cube when applied to R is $O(d^2 \cdot n)$*

Even for databases that are not skewness-monotonic, traffic is still often bounded. Consider a database in which attributes values are taken independently from some skewed distribution. Namely, for every attribute, it has some given probability of having a skewed value. Then, there is a smaller probability for having skews in higher levels of the cube lattice. Therefore, in such databases, there are instances of non-skewed c-groups that all of their sub-groups are skewed.

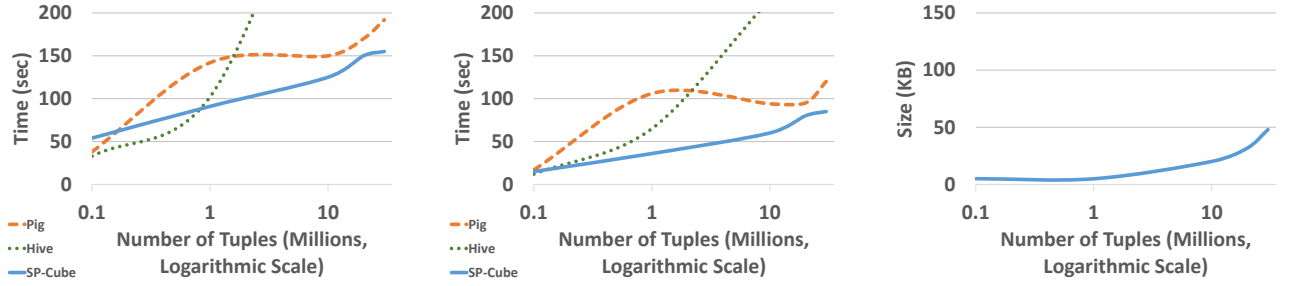
PROPOSITION 5.6. *Let R be a relation and consider its cube. Assume all attributes in R are independently distributed. Let c be a cuboid of l attributes. Let $t \in R$ be a tuple. If the probability that t contributes to a skewed group in c is at most $\frac{l+1\sqrt{d}}{d}$, then the total network traffic of the SP-Cube algorithm is bounded by $O(d^3 \cdot n)$.*

In accordance with the above results, our experiments on both real-life and synthetic data always exhibit moderate data transfer. They show that the intermediate data size was always significantly smaller than that of the competitor algorithms. We leave for future work the full characterization of relations over which traffic is guaranteed to be polynomial.



(a) running times comparison (b) reduce time comparison (c) map output comparison

Figure 4: The Wikipedia Statistics dataset



(a) running times comparison (b) map time comparison (c) SP-Sketch size

Figure 5: The USAGOV dataset

6. EXPERIMENTS

We tested the performance of the SP-Cube algorithm in an extensive series of experiments. We ran the experiments on Amazon's cloud environment, AWS [2]. We rented a cloud of 20 virtual machines of type m3.xlarge, each machine having 4 cores, a memory of size 15GB, and a 80GB of SSD. In the experiments we used two real-life datasets and two synthetic datasets that we have generated. We wrote our algorithm in Java and ran it on top of Hadoop [3], version 2.4.0. We ran our algorithm against Apache Pig Cube algorithm, version 0.12.0 and Apache Hive Cube algorithm, version 0.13.1. To assure that we compete against a good, optimized implementation of the competitors, we chose to run against the code developed by the actual authors of the competing algorithms. The [26] authors directed us to use Pig cube operator. Their algorithm is shipped as a feature in Pig since version 0.11.0. In [26], the algorithm's performance is demonstrated and shown to outperform previous cube algorithms implemented on top of MapReduce and we therefore do not compare SP-Cube to them. Regarding Hive cube algorithm, as any algorithm in Hive, it is compiled into a query plan that is computed according to some Hive policies, for instance having heuristics for map-side aggregations and choosing which aggregates should be computed before others. We refer below to the two cube algorithms as Pig ad Hive respectively.

The measures that we use to present our results are the total running time, the average running time of a mapper and a reducer in a single job, and the intermediate data size, which is the size of traffic in the cluster that is delivered between mappers and reducers. Note that in every experiment we computed all measures however we omit some of the graphs. In all cases where omitted a graph, this is because it showed similar trends to previous graphs, and thus for space constraints we just point to a previous graph to see the trend.

6.1 Real World Datasets

We use two real datasets for checking the performance of the algorithms on real-world data distributions. The use of two distinct datasets from different sources allows to examine different real-life data distributions. One dataset that we use is large while the other one is smaller. This allows us to examine what effect the data size has on the relative performance of the algorithms. In addition, to have a closer look at the affect of data size within each data distribution, we also considered for each of the two datasets subsets of varying sizes, selected by random sampling.

The first dataset that we use is the Wikipedia Traffic Statistics Dataset [7] containing statistics about user browser requests to Wikipedia pages. The dataset is 150GB in size, and contains about 12 billion records. We run our experiments on a random sample of 300 million records of it. The dataset has 4 dimension attributes and we calculate the cube over them. In retrospect we have found approximately 180 million c-groups in the data, and around 50 of them were skewed, of cardinality 5%-30% of the original tuples number.

The second dataset is the USAGOV dataset [1], containing log files of all clicks of users entering USA government websites between the years 2011-2013. This dataset is of size 22GB and we run our experiments on random sample of 30 million records out of it. We have found that this dataset contains around 30 of groups were skewed with cardinality of 2-8 million (6%-25% of the original tuples number). The total number of c-groups in this dataset was around 20

million. The dataset has 15 dimension attributes. As we wanted to compare the results to what was obtained for the Wikipedia dataset we built our cubes over 4 of them with similar settings to the Wikipedia traffic dataset. We also experimented with building cubes for more than 4 attributes, and the algorithms showed similar trends.

Wikipedia Traffic dataset. To examine the effect of the data size on the algorithm behavior, we took random samples of varying sizes out of the 300 million tuples, and analyzed the performance of the algorithm as the data grows. We further examined additional parameters that may explain the behavior. The results are depicted in Figure 4. Regarding running time, SP-Cube was 20% faster than Hive and 300% faster than Pig. This is shown in Figure 4a. (The Pig curve is shown here only partially as it ran out of scale for larger datasets).

Trying to explain the results, we examined closely the map and reduce times in the different algorithms. The average reduce times are shown in Figure 4b. We can see that Pig's average reduce time is both longer than those of SP-Cube and Hive. Hive's reduce time is close to that SP-Cube (and even slightly shorter). However, its map time was much larger than that of SP-Cube. The average map time shows a trend similar to what seen in Figure 4a and thus omitted.

We additionally examined the size of the intermediate data size. As can be seen in Figure 4c, much less data is transferred between the machines in SP-Cube compared to Pig and Hive. The growth rate of intermediate data size of SP-Cube is the slowest, and for 300 million tuples it is 5 times smaller than Pig and 6 times smaller than Hive. As we previously explained, in MapReduce, the amount of network traffic highly influences the running time of the algorithms. Regarding the SP-Sketch size, it was 6 orders of magnitudes smaller than the original dataset size, and we omit its graph due to space constraints.

USAGOV dataset. Here again we examine the algorithm behavior for growing data sizes. The results are shown in Figure 5. Figure 5a shows that, again, the SP-Cube algorithm performs significantly better than its competitors. In this experiment, Hive performs worse and we see a 300% relative speedup for CP-Cube even on small subsamples of the order of ten million tuples. We also see that SP-Cube has a speedup of 30% compared to the Pig algorithm. (The Hive curve is partial as it ran out of scale for larger datasets).

We tracked some measurements in the experiment and found that the exhibited running time is highly affected by the average map time. This is shown in Figure 5b. In this graph, we see that the Pig and Hive algorithms have a longer average map time as data size grows. The Pig algorithm map time is worse than that of our SP-Cube by almost 30%, whereas the Hive map time is much longer, which explains the long time for running Hive on this dataset. As it turns out, the SP-Cube optimized computation not only reduces data transfer but also allows the mappers to save work (and time) by not scanning redundant cube groups. Regarding the average reduce time, there was no significant difference between all three algorithms.

Looking closer at the Wikipedia and USGOV datasets, and in particular at the algorithms running time for subsets of similar sizes (10M-30M), we can note that unlike Pig and Hive that behave very differently on the two data sets, SP-Cube demonstrates similar performance (around 150 seconds) for two data distributions.

Finally, we have also examined the size of the SP-Sketch computed for the two datasets. As the trend is similar we show here only the results for USAGOV. In our implementation, the sketch is implemented as a java class, and we use java serialization engine to create a serialized stream of bytes for representing the sketch object. This file is delivered to all machines using the distributed file system. Each machine de-serializes the file and thus getting the sketch object. Figure 5c shows the sketch size as a function of the number of tuples. We see that the size grows linearly with a small gradient. It is worth mentioning that the growth is due to the fact that more skewed c-groups are identified. However, in all of the experiments, the sketch size is negligible compared to the original dataset size. Where the data is in the order of tens of Gigabytes, the sketch size is in the order of tens of Kilobytes, thus smaller by 6 orders of magnitude.

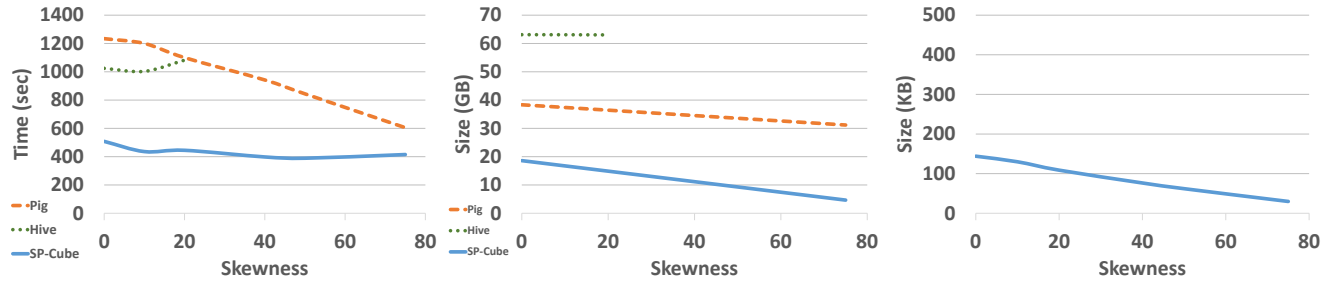
Note that the running time of SP-Cube includes the construction of the SP-Sketch and the actual cube computation. While the construction of the SP-Sketch is negligible compared to the cube computation time for large data sets, it is more noticeable for small data and this yields the slightly slower performance of SP-Cube. Note however that such small size of data is anyhow not a practical candidate for MapReduce computation, as the cube could be computed by a single machine using standard algorithms, and we show these results here only for completeness. In general, we conclude from our experiments that the SP-Cube consistently outperforms both Pig and Hive, with the performance gain increasing significantly for large datasets.

6.2 Synthetic Datasets

The next set of experiments involves two synthetic datasets that allow to isolate the effect of different properties of the data, and in particular examine the sensitivity of the algorithms to the data distribution. We name the the datasets *gen-binomial* and *gen-zipf*. We describe below the generation process of each dataset and then explain some interesting insights obtained from running the three algorithms on it. We ran the experiments for datasets with varying number of dimension attributes. As the trends were similar we show here a sample of the results for cubes with 4 dimension attributes. This allows to relate them to the results obtained for the real-life datasets.

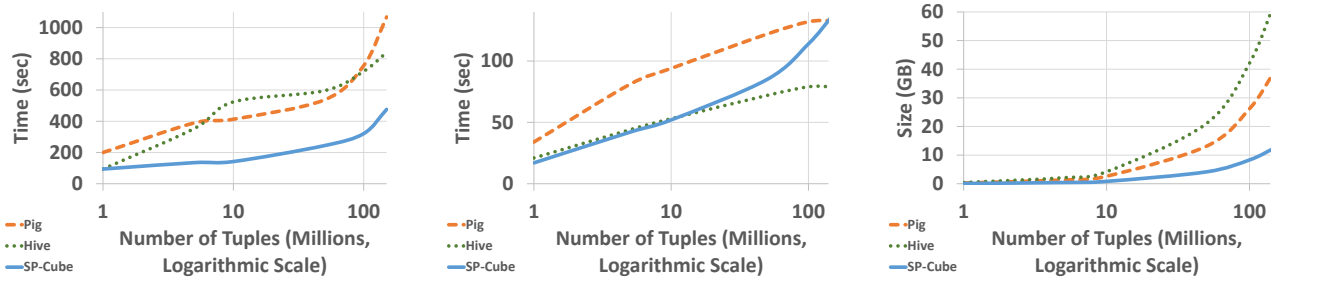
gen-binomial dataset. The generation process for this dataset is as follows. Tuples were generated independently with the following probabilities. With probability p, We uniformly pick a number $i \in 1, \ldots, 20$, and create a tuple having i in all of its attributes (namely the tuples $(1, 1, \ldots, 1), (2, 2, \ldots, 2)$, and so on). With probability $1 - p$, we draw each attribute uniformly as a 32-bit integer. Intuitively, in the generated dataset, a fraction p of the tuples contribute to skews in each cuboid. The other $1 - p$ of the tuples are likely not to form skews. We ran two sets experiments using these settings. In the first set of experiments, we fixed the size of the database and measured the algorithms performance for varying values of the probability p. We describe below a representative sample of the results.

Figure 6 shows the results for a dataset containing 300 million tuples with four dimension attributes and varying p values. Running time results are shown in Figure 6a. We



(a) running times comparison (b) map output size comparison (c) SP-Sketch size

Figure 6: gen-binomial: Varying skewness



(a) running times comparison (b) average reduce time comparison (c) map output size comparison

Figure 7: gen-zipf: Zipfian Distribution

can see that SP-Cube outperforms Pig and Hive and shows stable running time. Regarding Hive, it did not manage to handle heavy skews in the data: For $p \geq 0.4$ it got stuck as some reducers got out of memory. For Pig we found that it is highly affected by the value of p. Its performance decreased by a factor of 2 as p grows form 0 to 0.75.

These results can be explained by examining the size of the intermediate data, shown in Figure 6b. We can see that the map output in Pig is larger for non skewed data and is in general much larger than that of SP-Cube. The size of intermediate data decreases as p grows in both Pig and SP-Cube because the total number of c-groups is getting smaller. Hive's intermediate data is the largest and it appears that when p is large there are reducers that get too much information, and therefore get stuck. We also examined the average map and reduce times; their trend is similar to what we see for the size of the intermediate data and we therefore omit the graphs.

Finally, the SP-Sketch sizes for this experiment are depicted in Figure 6c. We can see that the sketch size is always very small and always less than 200KB (approximately 6 orders of magnitude smaller than the input data size). It gets smaller for large values of p as most of the data contributes to the given set of predefined skews and thus fewer additional ones are generated.

In the second set of experiments we fixed p, and changed the database size. We got similar trends. Due to space constraints, these results are shown in the Appendix.

gen-zipf dataset. In the gen-zipf experiment we tested the SP-Cube performance against a data with attributes drawn from a Zipfian distribution. We generated 150 million tuples independently. Inside each tuple all attributes were also drawn independently. Two of the attributes were generated using a zipf data generator, with 1000 elements and an exponent factor of 1.1. The other two attributes were drawn from a uniform distribution having 1000 elements. In this experiment we had groups of various sizes. Namely, for some cuboids, there were c-groups with a cardinality of 30 million (20% of the tuples) together with c-groups that created from a small number of tuples (dozens of tuples). We checked the algorithm behavior on subsamples of this dataset. Some of the results are shown in Figure 7. In 7a we show an improvement of 100% over Hive and 150% over Pig. We explain the behavior on smaller sets as in previous experiments - the preprocessing time of computing the sketch becomes more dominant. In this experiment, the map average time had a similar trend to the running time graph and thus omitted. The results of the average reduce time are shown in Figure 7b. Note that in this measure Hive performs the best, and SP-Cube and Pig perform quite similarly. However, we have found that the most dominant measure here was the map output size, that is presented in Figure 7c, in which SP-Cube has an improvement of 400% over Pig and 600% over Hive, and therefore had the best running times. Regarding SP-Sketch, its size was always bounded by 200KB, a 6 orders of magnitudes smaller than the original data. This graph is omitted as it has similar trends to previously presented sketch graphs.

We conclude by mentioning the performance of SP-Cube in terms of parallelism. In all of our experiments, SP-Cube achieved a good balancing between reducers, with the reducers' output data files being of similar sizes.

7. RELATED WORK

Sequential Cube Algorithms. The Cube Operator was originally presented in [23]. Many efficient sequential cube al-

gorithms have been developed over the years, e.g. [12, 22, 31, 24, 30, 34]. The cube lattice is often used in these algorithms. Some employ bottom up traversal over the lattice, as BUC [15], whereas others prefer top-down traversal [12]. We adopt the bottom up approach as it allowed us to achieve a two phases MapReduce algorithm, compared to previous top down MapReduce algorithm [25] that computes the cube using multiple rounds (to be further explained below). As the size of the cube may be exponential in the size of the input, some algorithms deal with full materialization of the cube, whereas others deal with partial materialization of the data and on-demand computation of cuboids [22].

Parallel Cube Algorithms and MapReduce. The fact that the data cube is expensive to compute, in particular when the input relation is large, has motivated the study of parallel algorithms for cube computation. [27] presented some parallel algorithms for cube computation that work for small clusters. More recently, dedicated platforms such as MapReduce are employed for enabling parallel computation over huge amounts of data, with much work dedicated to their implementation and efficiency [20, 28, 21]. The implementation of database operators over MapReduce has received much attention, suggesting efficient algorithms for *Join* [11, 35, 10, 16] and *cartesian product* [29]. Skewed data has been shown to be challenging for the computation of such operators as well [14]. Our work follows this line of works and proposes an efficient MapReduce algorithm for cube computation, that is resilient to data distribution. A complementary line of work considers MapReduce as a parallelized computational model and defines measurements for an efficient MapReduce algorithm [32, 9]. Among them, balanced workload and small communication overhead are the measurements that our work adopts.

MapReduce is implemented as an open source framework in Hadoop [3]; a very popular implementation that is highly used for massive computation. The Pig [5] and Hive [4] projects were developed on top of Hadoop, and give an abstraction of a database and an SQL-like query engine on top of it. As previously mentioned, the cube algorithm from [26] is shipped as a feature for implementing the cube in Pig. Both [26] and SP-Cube follow a bottom up approach and they both use sampling. However, as explained in the introduction a key disadvantage is that it makes decisions about the existence of skews in the granularity of a full cuboid and tails recursion and aborts that yield inferior performance. Note that the Pig framework adds to the original algorithm the use of combiners [19] but, as we show the result is still sensitive to the data distribution.

Another MapReduce cube algorithm is [25] that takes top-down traversal approach parallelizing the Pipesort algorithm [12]. This algorithm finds top-down computation paths in the lattice. This yieds a series of MapReduce rounds. Note that the more MapReduce rounds, the more are the ram-to-disk transactions and thus performance is inferior to previously mentioned algorithms. Furthermore, this algorithm suffers from the skews problem mentioned in Section 3. In case of a skewed c-group , the assigned reducer will be heavily loaded and parallelism will not be utilized. Thus, we did not include it in the experiments section.

Finally, we mention [33] which describes a LinkedIn system that contains implementation of cube on top of MapReduce. The algorithm is embedded in an internal system and exploits the system's data types and thus unlike SP-Cube cannot be run on unprocessed data in a generic platform.

Types of Aggregate Functions. Aggregation functions are traditionally divided into three classes [23]: distributive functions, such as count and sum, where partial aggregation can be merged to the full one; algebraic functions, like average, where multiple partial aggregates can be combined to obtain the result (e.g. partial sums and counts can be used for computing the full average using division); and holistic functions (like top-k most frequent) that in general cannot be computed from partial aggregates. [26] defines a subset of holistic measures called partially algebraic measures, which are functions whose computation can be partitioned according to one of their attributes' value. By its behavior, SP-Cube supports all distributive and algebraic aggregate functions, and all partially algebraic functions in which the generated partitions are not skewed. The support of efficient computation of arbitrary holistic aggregate functions for skewed c-groups is a challenging future work.

Sketching. Sampling-based sketches over large data sets are used for scalable computation of data properties such as top-k, distance measures, distinct counting, and various aggregates [13, 18, 17, 32, 26]. As previously mentioned, the sampling approach we use for building our SP-Sketch is inspired by [32, 26] and we adapt some of their proofs to show it is both small and accurate. The sampling technique adapted from [32] is a standard technique for sketching a data using uniform sampling. However, our use of the sample is novel. We use the sample to detect skewed groups and partition elements of each cuboid (for building the SP-Sketch), whereas [32] uses it to find bucketing elements for sorting.

8. CONCLUSIONS

In this paper, we present an efficient algorithm, SP-Cube, for cube computation over large data sets in MapReduce environments. Our algorithm is resilient to varied data distribution and consistently performs better than the previous solutions. At the core of our solution is a compact data structure, called SP-Sketch, that records all needed information for detecting skews and effectively partitioning the workload between the machines. Our algorithm uses the sketch to speed up computation and minimize communication overhead. Our theoretical analysis and experimental study demonstrate the feasibility and efficiency of our solution compared to the state of the art alternatives.

We focus here on aggregate functions that are distributive or algebraic. The support of arbitrary holistic aggregate functions is a challenging future work. Another challenge is exact theoretical characterization of data sets for which network traffic is guaranteed to be polynomial in the number of dimension attributes. Skews are problematic also for other common database operators such as joins. It is interesting to see whether some of the ideas presented here can be used to reduce network traffic and speed up computation for such operators. Finally, optimizing cube computation in other big data processing frameworks, such as Spark [6], is an intriguing future work.

Acknowledgements We are grateful to Yael Amsterdamer and the anonymous reviewers for their insightful comments. This work has been partially funded by the European Research Council under the FP7, ERC grant MoDaS, agreement 291071.

9. REFERENCES

[1] 1.usa.gov data. http://www.usa.gov/About/developer-resources/1usagov.shtml.

[2] Amazon web services. http://aws.amazon.com.

[3] Apache hadoop. http://hadoop.apache.org.

[4] Apache hive. http://www.hive.com.

[5] Apache pig. https://pig.apache.org/.

[6] Apache spark. https://spark.apache.org.

[7] Wikipedia page traffic statistic v3. http://aws.amazon.com/datasets/6025882142118545.

[8] A. Abelló, J. Ferrarons, and O. Romero. Building cubes with mapreduce. In *DOLAP*, pages 17–24, 2011.

[9] F. N. Afrati, A. D. Sarma, S. Salihoglu, and J. D. Ullman. Upper and lower bounds on the cost of a map-reduce computation. *PVLDB*, 6(4):277–288, 2013.

[10] F. N. Afrati and J. D. Ullman. Optimizing joins in a map-reduce environment. In *EDBT*, pages 99–110, 2010.

[11] F. N. Afrati and J. D. Ullman. Optimizing multiway joins in a map-reduce environment. *IEEE*, 23(9):1282–1298, 2011.

[12] S. Agarwal, R. Agrawal, P. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. In *VLDB*, pages 506–521, 1996.

[13] Z. Bar-Yossef, T. S. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan. Counting distinct elements in a data stream. In *RANDOM*, pages 1–10, 2002.

[14] P. Beame, P. Koutris, and D. Suciu. Skew in parallel query processing. In *PODS*, pages 212–223, 2014.

[15] K. S. Beyer and R. Ramakrishnan. Bottom-up computation of sparse and iceberg cubes. In *SIGMOD*, pages 359–370, 1999.

[16] S. Chu, M. Balazinska, and D. Suciu. From theory to practice: Efficient join query evaluation in a parallel database system. In *SIGMOD*, pages 63–78, 2015.

[17] E. Cohen. All-distances sketches, revisited: HIP estimators for massive graphs analysis. In *PODS*, pages 88–99, 2014.

[18] E. Cohen and H. Kaplan. Leveraging discarded samples for tighter estimation of multiple-set aggregates. In *SIGMETRICS/Performance*, pages 251–262, 2009.

[19] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[20] I. Elghandour and A. Aboulnaga. Restore: Reusing results of mapreduce jobs. *PVLDB*, 5(6):586–597, 2012.

[21] M. Y. Eltabakh, Y. Tian, F. Özcan, R. Gemulla, A. Krettek, and J. McPherson. Cohadoop: Flexible data placement and exploitation in hadoop. *PVLDB*, 4(9):575–585, 2011.

[22] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. D. Ullman. Computing iceberg queries efficiently. In *VLDB*, pages 299–310, 1998.

[23] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. In *Proceedings of the Twelfth International Conference on Data Engineering, February 26 - March 1, 1996, New Orleans, Louisiana*, pages 152–159, 1996.

[24] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *SIGMOD*, pages 205–216, 1996.

[25] S. Lee, J. Kim, Y. Moon, and W. Lee. Efficient distributed parallel top-down computation of ROLAP data cube using mapreduce. In *Data Warehousing and Knowledge Discovery - 14th International Conference, DaWaK 2012, Vienna, Austria, September 3-6, 2012. Proceedings*, pages 168–179, 2012.

[26] A. Nandi, C. Yu, P. Bohannon, and R. Ramakrishnan. Data cube materialization and mining over mapreduce. *IEEE Trans. Knowl. Data Eng.*, 24(10):1747–1759, 2012.

[27] R. T. Ng, A. S. Wagner, and Y. Yin. Iceberg-cube computation with PC clusters. In *SIGMOD*, pages 25–36, 2001.

[28] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas. Mrshare: Sharing across multiple queries in mapreduce. *PVLDB*, 3(1):494–505, 2010.

[29] A. Okcan and M. Riedewald. Processing theta-joins using mapreduce. In *SIGMOD*, pages 949–960, 2011.

[30] K. A. Ross and D. Srivastava. Fast computation of sparse datacubes. In *VLDB*, pages 116–125, 1997.

[31] S. Sarawagi, R. Agrawal, and N. Megiddo. Discovery-driven exploration of OLAP data cubes. In *EDBT*, pages 168–182, 1998.

[32] Y. Tao, W. Lin, and X. Xiao. Minimal mapreduce algorithms. In *SIGMOD*, pages 529–540, 2013.

[33] S. Vemuri, M. Varshney, K. Puttaswamy, and R. Liu. Execution primitives for scalable joins and aggregations in map reduce. *PVLDB*, 7(13):1462–1473, 2014.

[34] D. Xin, J. Han, X. Li, and B. W. Wah. Star-cubing: Computing iceberg cubes by top-down and bottom-up integration. In *VLDB*, pages 476–487, 2003.

[35] X. Zhang, L. Chen, and M. Wang. Efficient multi-way theta-join processing using mapreduce. *PVLDB*, 5(11):1184–1195, 2012.

[36] Y. Zhao, P. Deshpande, and J. F. Naughton. An array-based algorithm for simultaneous multidimensional aggregates. In *SIGMOD*, pages 159–170, 1997.

APPENDIX

Proofs: We provide below the proofs for Sections 4 and 5.

PROOF PROPOSITION 4.5. As in [26] we start by bounding the probability that our algorithm does not detect a single skewed group. Let g be a skewed c-group , and assume its size is s. The expected size of g in the sample is $\Omega(ln(nk)) = C*ln(nk)$ for some constant C. Using Chernoff Bound-

$$\Pr(s < (1-\delta) * C * ln(nk)) < exp^{-C*ln(nk)\frac{\delta^2}{2}} = O(\frac{1}{n}) \quad (1)$$

The last transition is due to the fact that C, δ are small constants and k is very small compared to n. Now, we need to consider all c-groups . Each cuboid contains at most $O(k)$ skewed groups, as the size of a skewed groups is at least m and the database contains n tuples. Therefore, the total number of skewed groups in D is bounded by $2^d \cdot k = O(m)$, summing over all cuboids. Using union bound, with probability of at least $1 - O(\frac{1}{k})$, the algorithm detects all skewed groups and keeps them in the sketch. □

PROOF PROPOSITION 4.6. In [32], a bucketing argument shows that sorting a set of n items, then with probability of at least $1-\frac{1}{n}$, the partitioning elements divide the input into partitions of size $O(m)$ each. We apply the same argument in our case. Then, for a single cuboid, with probability of at least $1 - \frac{1}{n}$, its partitioning elements kept in the SP-Sketch divide the cuboid (its non-skewed groups) into partitions of size $O(m)$. Thus, using union bound, the event happens for all cuboids with probability of at least $1 - \frac{2^d}{n}$. □

PROOF PROPOSITION 4.7. As stated in Proposition 4.5, with probability of at least $1 - \frac{1}{k}$ the algorithm detects all skewed c-groups. As we have stated earlier, there are at most k skewed c-groups at each node in the lattice, and $k-1$ partition elements. Putting it all together, we get that data size at each node in the SP-Sketch is $O(k)$. Since there are 2^d nodes, then the lattice size is bounded by $O(2^d \cdot k) = O(m)$. □

PROOF PROPOSITION 5.2. We use here Proposition 4.5 which shows that with probability of at least $1 - O(\frac{1}{k})$ the SP-Cube algorithm detects all skewed c-groups. The size of the key-value pair generated for every such skewed c-group is $O(d)$. (Recall that the size of attribute values and computed aggregates are taken as a constant). Each mapper locally aggregates at most $O(m)$ skewed c-groups. As there are k mappers, we obtain that they all create an intermediate data for skewed c-groups of size at most $O(dmk) = O(dn)$ □

PROOF THEOREM 5.3. We describe how to build such a relation R. R has d dimension attributes, and we describe how to build them (the value of the measure attribute is not important). Mark $w = m+1$, and mark by $S_{\frac{d}{2}}$ the set of all sets of $\frac{d}{2}$ numbers between 1 and d. For every $s \in S_{\frac{d}{2}}$, we add w identical tuples to R, each of the tuples having the value 1 in all attributes that their index is in s, and 0 in the other attributes. By definition of w, each cuboid in level $\frac{d}{2}$ contains a skewed group, which is the group of ones in the cuboid attribute indexes. In addition, no cuboid of size $\frac{d}{2}+1$ contains a skewed group as there are no $s_1, s_2 \in S$ that share the same values in any subset of $\frac{d}{2}+1$ attributes. Therefore, for every tuple, the algorithm consideres each of the c-groups of $\frac{d}{2}+1$ attributes as unmarked and non-skewed. For each such c-group, an intermediate data is generated. Summing over all tuples, the network traffic size for R is $\theta(2^d * n)$. □

PROOF PROPOSITION 5.5. Intuitively, such databases are handled very efficiently by our algorithm: In case there are no skews other than the most general $(*, *, \ldots, *)$ c-group, all c-groups are generated using the reducers assigned to the single attribute c-groups, so each tuple is sent at most d times. For instance, all c-groups of the tuple $t = (laptop, Paris, 2013, 700)$ are handled using the reducers of c-groups $(laptop, *, *)$, $(*, Paris, *)$, and $(*, *, 2013)$. When skews do exist, they are all "catched" using the sketch, and then the remaining c-groups can still be efficiently computed as above. We make this argument more formal below.

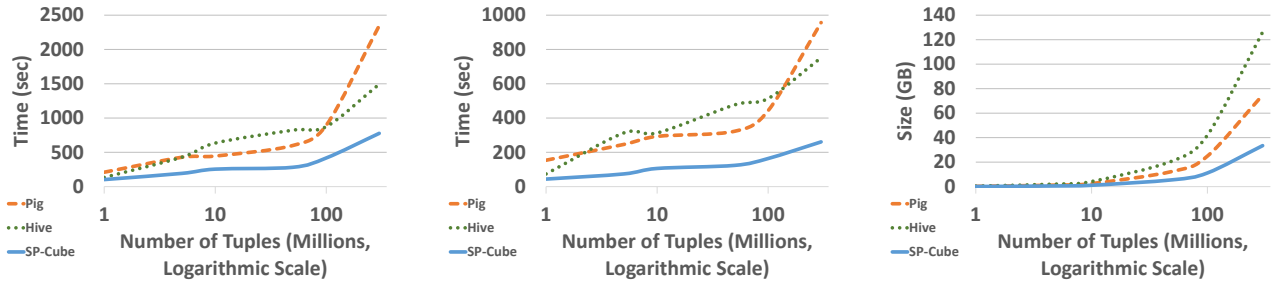
We have seen that all skews are computed using $O(dn)$ intermediate data. Let $t \in D$. *lattice*(t) contains 2^d nodes. We can ignore all of the nodes that their groups are skewed. Because D is skewness-monotonic, all attributes that are skewed, must be skewed together. All other attributes which are non-skewed, cover the rest of the lattice. As the number of non-skewed attributes is $O(d)$, each tuple generates at most $O(d)$ key-value pairs, each of size $O(d)$. Summing over all tuples, we have $O(d^2 \cdot n)$ intermediate data. Additionally to the data sent for skewed groups, we get a total size of $O(d^2 \cdot n)$. □

PROOF PROPOSITION 5.6. We have seen that handling of skewed groups requires intermediate data of size $O(dn)$. We compute the amount of traffic needed for the rest of the groups. Let t be a tuple. Denote by $C(t)$ the number of non-skewed c-groups in *lattice(t)* for which the algorithm generates intermediate data. These are the groups that the mapper sends to reducers. Other non-skewed c-groups are treated in the reduce phase using BUC, and need not a special traffic. As t is an arbitrary tuple, $C(t)$ is a random variable. It depends on which projections of t are skewed and which are not. We compute $\mathrm{E}(C(t))$. Denote by $C_i(t)$ the indicator random variable of the i'th node in *lattice(t)*. Then -

$$C_i(t) = \begin{cases} 1 & \text{c-group in node i not skewed, sent to a reducer} \\ 0 & \text{elsewhere} \end{cases}$$

Note that for $C_i(t) = 1$ for non-skewed c-groups that have a single attribute, or that all of their descendants in *lattice(t)* are skewed. We express $C(t)$ as a sum of the 2^d indicator random variables, of the nodes in *lattice(t)*, and get by linearity of expectation -

$$\mathrm{E}(C(t)) = \mathrm{E}(\sum_{\mathrm{i}=1}^{2^{\mathrm{d}}} C_i(t)) = \sum_{i=1}^{2^d} \mathrm{E}(C_i(t))$$



(a) running times comparison (b) average map time comparison (c) map output size comparison

Figure 8: gen-binomial: Varying data size

We compute the expectation of a single indicator variable. Assume a specific indicator of a node $v \in lattice(t)$, and assume v has $l \geq 2$ attributes (we later handle $l = 1$). We mark by g_t the c-group of t in v, and denote by A the event that all of g_t descendants in *lattice(t)* are skewed c-groups. Then-

$$\begin{aligned} \Pr(C_v(t) = 1) &= \Pr(C_v(t) = 1 \mid A) * \Pr(A) \\ &\leq \Pr(A) \end{aligned} \tag{2}$$

We use the law of total probability, and the fact that if g_t has a non-skewed descendant in *lattice(t)*, then g_t is not sent to a reducer. As v's descendants are all cuboids of $l-1$ attributes, and the attributes in R are independently distributed, we get-

$$\begin{aligned} \Pr(A) &\leq \cap_{p \in descendants(g_t)} \Pr(p \text{ is skewed}) \\ &\leq (\frac{d^{\frac{1}{l}}}{d})^l \\ &= \frac{d}{d^l} \\ &= \frac{1}{d^{l-1}} \end{aligned} \tag{3}$$

Therefore, $\mathrm{E}(C_v(t)) \leq \frac{1}{d^{l-1}}$. Recall that the number of nodes having l attributes is $\binom{d}{l} = O(d^l)$. Therefore, for every $l \geq 2$, the expected number of non-skewed c-groups with l attributes that are sent to reducers is bounded by $O(d)$. Summing over the lattice levels from 2 to d, we get that the expected number for the entire lattice (except from level 1) is bounded by $O(d^2)$. For the case of $l = 1$, the number of c-groups in *lattice(t)* with one attribute is d, and therefore the number of non-skewed c-groups that are sent to reducers is $O(d)$. We thus get that for (an arbitrary tuple) t, the expected number of non-skewed c-groups that are sent to a reducer is at most $O(d^2)$. Namely, $\mathrm{E}(C(t)) = O(d^2)$. As the information sent for each non skewed c-group is of size $O(d)$, we get that the expected size of network traffic that the algorithm is sending for (an arbitrary tuple) t is $O(d^3)$. Therefore, the entire network for all of the tuples is at most $O(d^3 \cdot n)$. □

Experiments: We now provide our results of the second set of experiments on the synthetic *gen-binomial*.

In the second set of experiments we fixed p, and checked the algorithms performance for varying database size. Figure 8 illustrates the results for $p = 0.1$. Regarding running time, we can again see in Figure 8a that SP-Cube outperforms Pig and Hive. We can also see compatibility with the results obtained for the real-life datasets. On samples of size 50 million, for instance, Hive is worse than Pig similarly to the USAGOV dataset. For samples of size 300 million, Pig is worse than Hive, similarly to the Wikipedia dataset.

However, the gaps are dramatically larger in this experiment. For 300 million tuples, SP-Cube achieves a speedup of about 200% compared to Hive and 300% compared to Pig. These gaps follow from similar gaps in the average map time and map output sizes, shown in the graphs in Figure 8b and 8c, respectively. In this experiment the reduce average times behaved similarly to the total running time, and the sketch size was bounded by hundreds of Kilobytes, similarly to previous experiments.

In the second set of experiments we fixed p, and checked the algorithms performance for varying database size. Figure 8 illustrates the results for $p = 0.1$. Regarding running time, we can again see in Figure 8a that SP-Cube outperforms Pig and Hive. We can also see compatibility with the results obtained for the real-life datasets. On samples of size 50 million, for instance, Hive is worse than Pig similarly to the USAGOV dataset. For samples of size 300 million, Pig is worse than Hive, similarly to the Wikipedia dataset.

However, the gaps are dramatically larger in this experiment. For 300 million tuples, SP-Cube achieves a speedup of about 200% compared to Hive and 300% compared to Pig. These gaps follow from similar gaps in the average map time and map output sizes, shown in the graphs in Figure 8b and 8c, respectively. In this experiment the reduce average times behaved similarly to the total running time, and the sketch size was bounded by hundreds of Kilobytes, similarly to previous experiments.