Distributed Cube Materialization on Holistic Measures*

Arnab Nandi†, Cong Yu‡, Phil Bohannon♭, Raghu Ramakrishnan♭
†University of Michigan, Ann Arbor, MI ‡Google Research, New York, NY ♭Yahoo! Research, Santa Clara, CA
arnab@umich.edu, congyu@google.com, {plb, ramakris}@yahoo-inc.com

***Abstract*—Cube computation over massive datasets is critical for many important analyses done in the real world. Unlike commonly studied algebraic measures such as `SUM` that are amenable to parallel computation, efficient cube computation of holistic measures such as `TOP-K` is non-trivial and often impossible with current methods. In this paper we detail real-world challenges in cube materialization tasks on Web-scale datasets. Specifically, we identify an important subset of *holistic measures* and introduce MR-Cube, a MapReduce based framework for efficient cube computation on these measures. We provide extensive experimental analyses over both real and synthetic data. We demonstrate that, unlike existing techniques which cannot scale to the 100 million tuple mark for our datasets, MR-Cube successfully and efficiently computes cubes with holistic measures over billion-tuple datasets.**

I. Introduction

Data cube analysis [12] is a powerful tool for analyzing multidimensional data. For example, consider a data warehouse that maintains sales information as:

$\langle city, state, country, day, month, year, sales \rangle$

where $(city, state, country)$ are attributes of the location dimension and $(day, month, year)$ are attributes of the temporal dimension. Cube analysis provides the users with a convenient way to discover insights from the data by computing aggregate measures (e.g., total sales) over all possible groups defined by the two dimensions (e.g., overall sales for "New York, NY, USA" during "March 2010"). Many studies have been devoted to designing techniques for *efficiently computing the cube* [2], [3], [5], [10], [13], [14], [18], [22].

There are two main limitations in the existing techniques that have so far prevented cube analysis being extended to an even broader usage such as analyzing Web query logs. First, they are designed for a single machine or clusters with small number of nodes. Given the rate at which data are being accumulated (e.g., terabytes per day) at many companies, it is increasingly difficult to process data with a single (or a few) machine(s). Second, many of the established techniques take advantage of the measure being *algebraic* [12] and use this property to avoid processing groups with a large number of tuples. Intuitively, a measure is algebraic if the measure of a super-group can be easily computed from its sub-groups. (E.g. `SUM(a+b) = SUM(a) + SUM(b)`; Sec. II provides a formal definition.) This allows parallelized aggregation of data subsets whose results are then post-processed to derive the final result. Many important analyses over logs, however, involve computing ***holistic* (i.e., non-algebraic) measures** such as the distinct number of users or the top-k most frequent queries. An example of such a query is provided in Fig. 1 (we will revisit this example in detail in Sec. III.)

*Work done while AN & CY were at Yahoo! Research New York.

```
CUBE ON location(ip), topic(query)
FROM 'log.table' as (user, ip, query)
GENERATE reach(user), volume(user)
HAVING reach(user) > 5

reach(user) := COUNT(DISTINCT(user))
volume(user) := COUNT(user)
location(ip) maps ip to [Country, State, City]
topic(query) maps query to [Topic, Category, Subcategory]
```

Fig. 1. Typical cubing task on a web search query log, used to identify high-impact web search queries (for details, see Sec. III.) Data, dimensions and measures are given as input. Unlike *volume*, *reach* is holistic, and hence will typically fail or take an unreasonable amount of time to compute with existing methods due to data size and skew, further discussed in Sec. IV. MR-Cube efficiently computes this cube using MapReduce, as shown in Sec. VI.

Increasingly, such large scale data are being maintained in clusters with thousands of machines and analyzed using the MapReduce [9] programming paradigm. Extending existing cube computation techniques to this new paradigm, while also accommodating for holistic measures, however, is non-trivial.

The first issue is how to *effectively distribute the data* such that no single machine is overwhelmed with an amount of work that is beyond its capacity. With algebraic measures, this is relatively straight forward because the measure for a large super-group (e.g., number of queries in "US" during "2010") can be computed from the measures of a set of small sub-groups (e.g., number of queries in "NY, US" during "March 2010"). For holistic measures, however, the measure of a group can only be computed directly from the group itself. For example, to compute the distinct number of users who were from "USA" and issued a query during "March 2010," the list of unique users must be maintained. For groups with a large number of tuples, the memory requirement for maintaining such intermediate data can become overwhelming. We address this challenge by identifying an important subset of holistic measures, *partially algebraic measures*, and introducing a *value partition* mechanism such that the data load on each machine can be controlled. We design and implement sampling algorithms to efficiently detect groups where such value partition is required. The second issue is how to *effectively distribute the*

computation such that we strike a good balance between the amount of intermediate data being produced and the pruning of unnecessary data. We design and implement algorithms to partition the cube lattice into *batch areas* and effectively distribute the computation across available machines.

Main Contributions: To the best of our knowledge, our work is the first comprehensive study on cube materialization for holistic measures using the MapReduce paradigm. In addition to describing real world challenges associated with holistic cube computation using MapReduce for Web-scale datasets, we make the following main contributions:

- We formally introduce *partially algebraic measures*, an important subset of holistic measures that are MapReduce friendly.
- We propose two techniques, *value partitioning* and *batch area identification* that effectively leverage the MapReduce framework to distribute the data and computation workload.
- We propose a two-phase cube materialization algorithm **MR-Cube** that employs these techniques to successfully cube billion-tuple sized datasets.

Extensive experimental analyses over real data demonstrate that **MR-Cube** significantly outperforms existing techniques in terms of efficiency and scalability.

II. Preliminaries

We begin by introducing the basic formalisms used in the rest of the paper, most of which follow the original notions described in [12] for Data Cube and [9] for MapReduce. We explain each concept through a *running example*, which is based on a real log analysis task. The same example will also apply to the data set in our experiments.

As shown in Fig. 2, the raw data is maintained as a set of tuples and each tuple has a set of raw attributes, such as `ip` and `query`.

id	time	uid	ip	query
e1	091203	u1	64.97.55.3	iPhone
e2	091203	u2	34.2.54.21	iPhone
e3	091204	u1	64.97.4.23	Nikon D40
...	...	...	...	...

Fig. 2. Raw dataset, as maintained on disk.

For many analyses, it is more desirable to map some raw attributes into a fixed number of derived attributes through a *mapping* function. For example, `ip` can be mapped to `city`, `state`, and `country`. Similarly, `query` can be mapped to `sub-category`, `category`, and `topic`. We assume such mappings are accomplished by functions that are provided by the user. Fig. 3 illustrates the derived tuples after the raw attributes have been mapped.

Dimension attributes & Cube lattice

The term **dimension attributes** refers to the set of attributes that the user wants to analyze. Based on those attributes, a **cube lattice** can be formed representing all possible grouping(s) of the attributes. For example, Fig. 4 illustrates

			ip			query		
id	time	uid	c'ntry	state	city	topic	category	subcat
e1	091203	u1	USA	Michigan	Ann Arbor	Shopping	Phone	Smart
e2	091203	u2	USA	New York	NYC	Shopping	Phone	Smart
e3	091204	u1	USA	Michigan	Detroit	Shopping	Camera	SLR
...	...	...	...	...	...	...	...	...

Fig. 3. Derived dataset, by converting `ip` and `query` using classifiers.

a cube lattice (only a fraction of the lattice is displayed in detail) where the dimension attributes include the six attributes derived from `ip` and `query`.

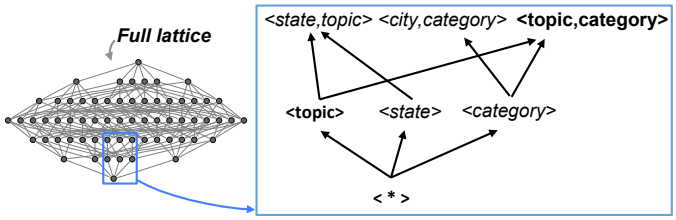

Fig. 4. Cube Lattice using a flat set of 6 dimensions, yielding 64 cube regions. 7 are shown in detail. Note that only 3 of the 7 depicted regions are valid: $< * >$, $< \texttt{topic}, \texttt{category} >$ and $< \texttt{topic} >$.

Cube group & cube region

Each node in the lattice represents one possible grouping/aggregation. For example, the node $\langle *, *, *, \texttt{topic}, \texttt{category}, * \rangle$ (displayed in short form as $\langle \texttt{topic}, \texttt{category} \rangle$) represents all groups formed by aggregating (i.e., groupby) on `topic` and `category`. Each group in turn contains a set of tuples satisfying the same aggregation value. In this paper, we use the term **cube region** to denote a node in the lattice and the term **cube group** to denote an actual group belonging to the cube region. For example, tuples with `id` *e1* and *e2* both belong to the **group** $\langle *, *, *, \texttt{Shopping}, \texttt{Phone}, * \rangle$, which belongs to the **region** $\langle *, *, *, \texttt{topic}, \texttt{category}, * \rangle$[1]. (A cube group is considered to *belong to* a cube region if the former is generated by the aggregation condition of the latter.) In another words, a cube region is defined by the grouping attributes while a cube group is defined by the values of those attributes.

Each edge in the lattice represents a *parent/child relationship* between two regions, where the grouping condition of the child region (i.e., the one pointed to by the arrow) contains exactly one more attribute than that of the parent region. A parent/child relationship can be similarly defined between cube groups. For example, the group representing the tuples of $\langle *, \texttt{Michigan}, \texttt{USA}, \texttt{Shopping}, \texttt{Phone}, * \rangle$ is a child of the group representing the tuples of $\langle *, *, \texttt{USA}, \texttt{Shopping}, \texttt{Phone}, * \rangle$.

An important observation is that not all regions represent valid aggregation conditions. For example, the region $\langle *, *, \texttt{city}, *, \texttt{category}, * \rangle$ groups tuples by the same city (and category), regardless of countries and states. This means, tuples from *Delhi, NCR, India* will be grouped together with tuples from *Delhi, Michigan, USA*—a scenario that is usually unintended by the user. This problem is caused by the inherent

[1]We overload the use of symbol "*" here: it denotes both *not grouping by this attribute* for cube region and *any value of this attribute* for cube group.

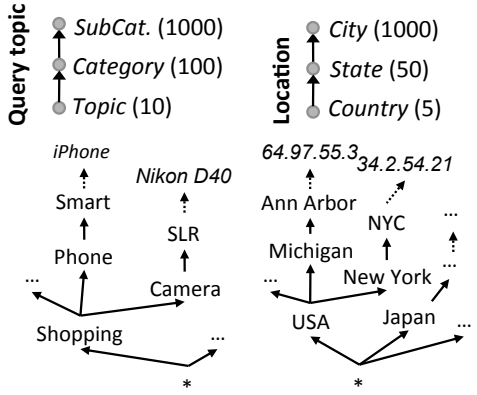

Fig. 5. 6 attributes forming 2 dimension hierarchies query topic and location, with cardinalities and example values for each.

hierarchical relationships among the attributes. As illustrated in Fig. 5, city, state, country form a **dimension hierarchy**, with city at the finest level and country at the broadest level. A region that groups on city needs to group on state and country as well to become valid. Indeed, while there are 8 possible aggregation conditions for the three attributes, only 4 are valid: $\langle \text{country}, \text{state}, \text{city} \rangle$, $\langle \text{country}, \text{state}, * \rangle$, $\langle \text{country}, *, * \rangle$, $\langle *, *, * \rangle$. By grouping attributes into hierarchies and eliminating invalid cube regions from the lattice in Fig. 4, we can obtain a more compact **hierarchical cube lattice** as shown in Fig. 6. Note that the idea of a hierarchical cube lattice is not novel; it is in fact similar in spirit to the descriptions in original data cube paper [12]. Here, the cube region $\langle \text{state}, \text{topic} \rangle$ corresponds to the original cube region $\langle *, \text{state}, \text{country}, *, *, \text{topic} \rangle$[2]. Hierarchical cubes are often significantly smaller than their counterpart flat cubes.

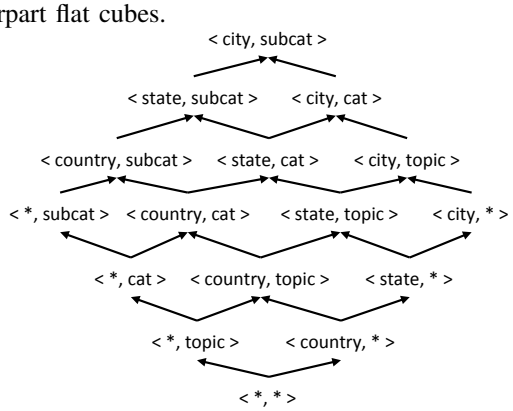

Fig. 6. Cube Lattice when considering 6 attributes in 2 dimension hierarchies. All cube regions are valid.

Cubing Task and Measures

Given the hierarchical cube, the task of cube computation is to *compute given measures for all valid cube groups*, where a measure is computed by an aggregation function based on all the tuples within the group. Example measures include SUM and TOP-K. Typically, measures are characterized by the following two properties.

Algebraic & Holistic: Given a group G and any mutually exclusive partition of G, $\{G_i \mid i = 1...k\}$ (i.e., $\bigcup_{i=1}^{k}(G_i) = G$ and $\forall ij, i \neq j, G_i \bigcap G_j = \emptyset$), a measure M is *algebraic* if $\exists$ F, G such that $M(G) = F(H(G_1), \ldots, H(G_k))$, where function H returns an n-tuple and n is a constant for all $|G_i|$. It is important to note that G can be partitioned in multiple ways and the same F and G apply to all possible partitions. A measure M is *holistic* if no such functions F and G exist for all possible partitions. Examples of algebraic functions are SUM and STD_DEV, and examples of holistic functions are TOP-K, MODE and MEDIAN.

Monotonic: A numerical measure is monotonic if for any pair of cube groups such that G_c is a child of G_p, $M(G_c) \geq M(G_p)$ (monotonically decreasing) or $M(G_c) \leq M(G_p)$ (monotonically increasing). Examples of monotonic measures are REACH and COUNT.

MapReduce

MapReduce is a shared-nothing parallel data processing paradigm that is designed for analyzing large amounts of data on commodity hardware. Hadoop is an open-source implementation of this framework. A typical MapReduce job involves three basic phases, which are illustrated in Fig. 7.

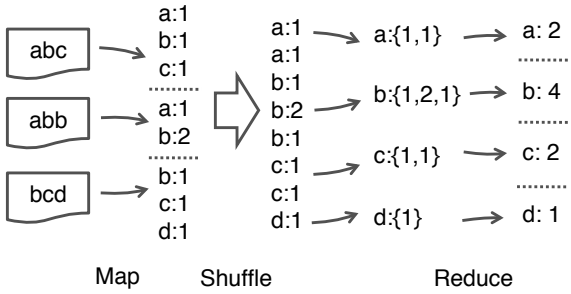

Fig. 7. Example MapReduce job for counting the occurrences of each character in the input strings.

During the Map phase, the input data is distributed across the *mapper* machines, where each machine then processes a subset of the data in parallel and produces one or more $\langle \text{key}, \text{value} \rangle$ pairs for each data record. Next, during the Shuffle phase, those $\langle \text{key}, \text{value} \rangle$ pairs are repartitioned (and sorted within each partition) so that values corresponding to the same key are grouped together into $\{v_1, v_2, ...\}$. Finally, during the Reduce phase, each *reducer* machine processes a subset of the $\langle \text{key}, \{v_1, v_2, ...\} \rangle$ pairs in parallel and writes the final results to the distributed file system. The map and reduce tasks are defined by the user while the shuffle is accomplished by the system. Fault-tolerance is inherent to a MapReduce system, which detects failed map or reduce tasks and reschedules the tasks to other nodes in the cluster.

III. Example Analysis Tasks

This paper aims to address challenges with the following core task: ***given a large amount of data, a holistic measure and dimension hierarchies, efficiently compute the measure of all cube groups that satisfy the pruning condition (if any), using the MapReduce infrastructure.*** In this section, we provide two real-world analysis tasks, for which cube computation is the core first step. They are representative

[2]I.e., we use $\langle \text{state} \rangle$ as a shorthand for $\langle *, \text{state}, \text{country} \rangle$.

of the typical tasks that analysts demand and will be used throughout this paper. They both involve holistic measures: `reach` for the former and `top-k` for the latter. The schema, data and lattice for both tasks are the same as our running example in Sec. II.

*Example 1 (***Coverage Analysis***): Given the location and topic hierarchies, compute* `volume` *and* `reach` *of all cube groups whose* `reach` *is at least* 5*. Highlight those cube groups whose* `reach` *is unusually high compared with their* `volume`*. Here, measure* `volume` *is defined as the number of search tuples in the group, while* `reach` *is defined as the number of unique users issuing those searches.*□

This analysis is inspired by the need to identify query topics that are issued relatively infrequently by the users, but cover a large number of users: these queries are often missed by traditional frequency based analyses because of their relative low volume, even though those query topics can in fact have an impact on a very large user population. A SQL-style specification corresponding to the cubing task for this analysis is shown in Fig. 1, while the dimension hierarchies and hierarchical cube lattice are shown in Fig. 5 and Fig. 6 respectively.

*Example 2 (***Top-k Query Analysis***): Given the location and topic hierarchies, compute* `top-5` *frequent queries for all cube groups. Highlight those cube groups that share less than* 3 *top queries with their parent groups.*□

This analysis aims to discover location bias for various query topics. For example, the top political queries in Austin can be very different from those in Texas in general, which in turn can be different from those in the entire USA.

We note again that cube computation is only part of both analysis tasks, albeit an essential part. Highlighting the desired cube groups occurs above the cube computation layer and is beyond the scope of this work. We provide some discussion on how to efficiently identify interesting groups in Sec. VIII.

IV. Challenges

Typically, the `CUBE` operation can be expressed in high-level MapReduce languages (e.g., Pig [19]) as a disjunction of `groupby` queries. A query optimizer would then (in the ideal case) combine all the queries into a single MapReduce job. Algo. 1 represents this combined cube computation. Intuitively, this naive algorithm divides the full cubing task into a collection of aggregation tasks, one for each cube group, and distributes them for computation using the MapReduce framework. In particular, the function $R(e)$ extracts, from the tuple e, the values of the groupby attributes specified by the cube region R.

For example, given the cube lattice in Fig. 6, the search tuple $e1 = \langle 091203, u1, 64.97.55.3, iPhone\rangle$ is mapped to 16 groups (one per region in the cube), including the smallest group $\langle Ann\_Arbor, Smart\rangle$ and the broadest group $\langle *, *\rangle$. Each reducer then computes the measures for its assigned groups by applying the measure function. Measure-specific optimization can be incorporated into the naive algorithm to reduce amount of intermediate data. As an example, for coverage analysis (Example 1), to compute the measure `reach`, we only need the attribute `uid`. Therefore the mapper can emit just $e.uid$ instead of the full tuple.

Algorithm 1 Naive Algorithm

```
MAP(e)
1  # e is a tuple in the data
2  let C be the Cube Lattice
3  for each Region R in C
4  do k = R(e);
5     EMIT k ⇒ e

REDUCE/COMBINE(k, {e1, e2, ...})
1  let M be the measure function
2  EMIT k ⇒ M({e1, e2, ...})
```

It should be noted that despite its simplicity, the Naive algorithm fares quite well for small datasets. As we will see in Sec. VI, it outperforms its competitors for such datasets due to the extremely low overhead costs. However, as the scale of data increases, we encounter two key challenges that cause this algorithm to perform poorly and eventually fail: *size of intermediate data* and *size of large groups*. We briefly describe these challenges next.

A. Size of Intermediate Data

The first challenge arises from the large size of intermediate data being generated from the map phase, which measures at $|C| \times |D|$, where $|C|$ is the number of regions in the cube lattice and $|D|$ is the size of the input data. Since $|C|$ increases exponentially with both the number and depth of dimensions to be explored, the naive approach can quickly lead to the system running out of disk space during the map phase or struggling through the shuffle (i.e., sort) phase.

B. Size of Large Groups

The second challenge arises from cube groups belonging to cube regions at the bottom part of the cube lattice, such as $\langle USA, Shopping\rangle$ or even $\langle *, *\rangle$ (i.e., the cube group containing all tuples). The reducer that is assigned the latter group essentially has to compute the measure for the entire dataset, which is usually large enough to cause the reducer to take significantly longer time to finish than others or even fail. As the size of the data increases, the number of such groups also increases. We call such groups **reducer-unfriendly**. A cube region with a significant percentage of reducer-unfriendly groups is called *reducer-unfriendly region*.

For algebraic measures, this challenge can addressed by not processing those groups directly: we can first compute measures only for those smaller, reducer-friendly, groups, then combine those measures to produce the measure for the larger, reducer-unfriendly, groups. Such measures are also amenable to mapper-side aggregation which further decreases the load on the shuffle and reduce phases. For holistic measures, however, measures for larger groups can not be assembled from their smaller child groups, and mapper-side aggregation is also not possible. Hence, we need a different approach.

V. The MR-Cube Approach

We propose the **MR-Cube approach** that addresses the challenges of large scale cube computation with holistic measures. The complexity of the cubing task depends on two aspects: *data size*, which impacts both intermediate data size and the size of large groups, and *cube lattice size*, which impacts intermediate data size and is controlled by the number/depth of dimensions. We deal with those complexities in a two-pronged attack: *data partitioning* and *cube lattice partitioning*. Specifically, our goal is to divide the computation into pieces such that no reducer has to deal with extremely large data groups, and the overall intermediate data size is controlled. A pictorial representation of the overall MR-Cube process is shown in Fig. 8, for easy reference to the details in this section.

A. Partially Algebraic Measures

We begin by identifying a subset of holistic measures that are *easy* to compute in parallel than an arbitrary holistic measure. We call them *partially algebraic measures*. This notion is inspired by common ad-hoc practices for computing a single holistic measure from an extremely large number of data tuples. For example, to compute the measure `reach` (i.e., unique number of users) of billions of search tuples, a known practical approach is to first group the tuples by the user id (uid) and then count the number of such groups produced. It is as if the holistic measure has become *algebraic* for the attribute uid. Formally, we have:

*Definition 1 (***Partially Algebraic Measure***):* Given a cube group G, an attribute a, and any mutually exclusive partitions of G, $\{G_i | i = 1...k\}$ (i.e., $\bigcup_i^k(G_i) = G$ and $\forall ij, i \neq j$, $G_i \bigcap G_j = \emptyset$), such that $G_i.a \bigcap G_j.a = \emptyset$. An aggregate measure M is *partially algebraic on* a if $\exists$ F, H, s.t. $M(G) = F(H(G_1), \ldots, H(G_k))$ where H returns an n-tuple and n is constant regardless of all $|G_i|$. We call a the *algebraic attribute* of M.

Unlike traditional algebraic measures (see Sec. II), which can be computed from any mutually exclusive sub-groups ($G_i \bigcap G_j = \emptyset$), partially algebraic measures can be computed from only those sub-groups that are not only mutually exclusive on the full tuple, but also mutually exclusive after projecting on the algebraic attribute ($G_i.a \bigcap G_j.a = \emptyset$). For example, consider `reach`, which is a holistic measure that is partially algebraic on attribute uid, and the large group $\langle USA, * \rangle$, which contains all searches initiated within USA. If the partitioning is done arbitrarily, the measure of the whole group can not be computed from the smaller sub-groups, since uids can be shared across different sub-groups. However, if we split the whole group into sub-groups based on the uid (i.e., each sub-group can be considered as $\langle USA, *, hash(uid) \rangle$ where $hash(uid)$ hashes the uid), we can then compute the `reach` of $\langle USA, * \rangle$ by summing up the `reach` for those sub-groups.

We note that Definition 1 can be extended to measures computed based on multiple attributes. Again, consider a cube analysis task with location and topic hierarchies. To compute top-k (query, user) pairs per cube group, the sub-groups needs to be partitioned on both uid and query, i.e., a *Composite* Partially Algebraic Measure.

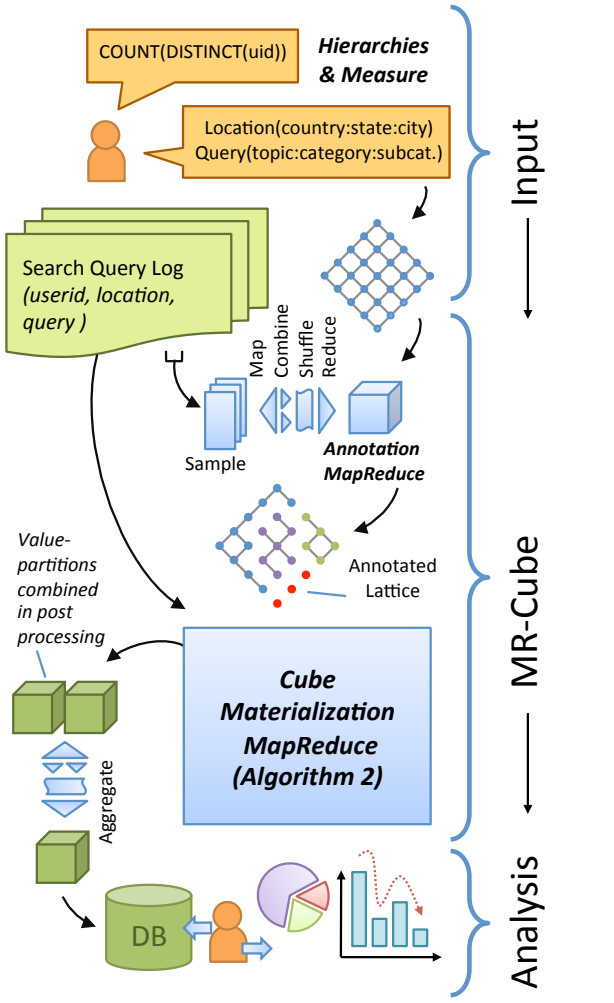

Fig. 8. Overview of the MR-Cube system: A user specifies dimension hierarchies and a measure. An annotated cube lattice is constructed using a data sample to estimate region cardinalities: *reducer-unfriendly regions* (Sec. IV) are *value partitioned* (Sec. V-B). Regions are then combined into *batch areas* (Sec. V-C). In Cube Materialization (Alg. 2), tuples are mapped to each batch area. Reducers evaluate the measure for each batch area. Partitioned measures are merged in a post-processing step. The cube is loaded into a DB for future exploration. ■

Many holistic measures turn out to be partially algebraic, including the above mentioned `reach` and `top-k` frequent queries. To detect if a holistic measure M is partially algebraic, we adopt the following **detection principle**: if there exists an aggregation A based on attribute a and an algebraic measure M', such that $M(D) = M'(A(D))$, where D is the original data, then M is partially algebraic on attribute a. For the measure `reach`, the aggregation is '`groupby uid`' and the algebraic measure is `count`.

In this work, we assume the algebraic attribute is either provided by the analyst or detected by the system for a few frequently used measures. Automatically deciding whether a holistic measure is also partially algebraic and then detecting its algebraic measure is by itself an interesting and hard

problem. We leave this as part of our future work.

We call this technique of partitioning large groups based on the algebraic attribute **value partitioning**, and the ratio by which a group is partitioned the *partition factor*. In the next section, we describe how value partitioning leverages the algebraic attribute of a partially algebraic measure to efficiently compute the cube over a large amount of data. We note that our work is the first to focus on this practically important subset of holistic measures.

B. Value Partitioning

An easy way to accomplish value partitioning is to run the naive algorithm, but further partition each cube group based on the algebraic attribute. However, such an approach is problematic. The number of map keys being produced is now the product of the number of groups and the partition factor (instead of just the former in Algo. 1). This can put a significant burden on the shuffle phase. Further, many of the original groups contain a manageable number of tuples and partitioning those groups is entirely unnecessary. Even for large, reducer-unfriendly, groups, some will require partitioning into many sub-groups (i.e., large partition factor), while others will only need to be partitioned into a few sub-groups.

Therefore, we want to perform value partitioning only on groups that are likely to be reducer-unfriendly and dynamically adjust the partition factor. One approach is to detect reducer-unfriendly groups on the fly and perform partitioning once they are detected. This is, however, undesirable because it requires us to maintain information about groups visited so far at each mapper, which can be overwhelming for the mapper. Another approach is to scan the data and compile a list of potentially reducer-unfriendly groups, for which the mapper will perform partitioning. This is again undesirable because checking against the potentially large list slows down the mapper. We observe that different regions in the cube lattice have different reducer-unfriendliness depending on their size. Intuitively, regions at the bottom of the cube lattice, that contain few groups (e.g., $\langle *, * \rangle$ or $\langle \texttt{country}, * \rangle$), are most likely to contain groups that are reducer-unfriendly, while regions containing many groups have mostly reducer-friendly groups.

As a result, we adopt a **sampling approach** where we estimate the reducer-unfriendliness of each cube region based on the number of groups it is estimated to have, and perform partitioning for all groups within the list of cube regions (a small list) that are estimated to be reducer unfriendly.

This sampling is accomplished by performing cube computation using the naive algorithm on a small random subset of data, with `count` as the measure. For each discovered group, this gives us the number of tuples in the sample it contains. Based on Proposition 1, we declare a group G to be reducer-unfriendly if we observe more than $0.75rN$ tuples of G in the sample, where N is the sample size and $r = \frac{c}{|D|}$ denotes the maximum number of tuples a single reducer can handle (c) as a percentage of the overall data size ($|D|$). (See Proposition 1.) We declare a region to be reducer-unfriendly if it contains at least one reducer-unfriendly group. In addition, let the sample count of the largest reducer-unfriendly group in the region be s, we annotate the region with the appropriate *partition factor*, an integer that is closest to $\frac{s}{rN}$[3]. Fig. 9 illustrates the cube lattice of Fig. 6 after it is annotated by the sampling process: the lattice is divided into reducer-friendly and reducer-unfriendly parts and for each reducer-unfriendly region, a partition factor is assigned.

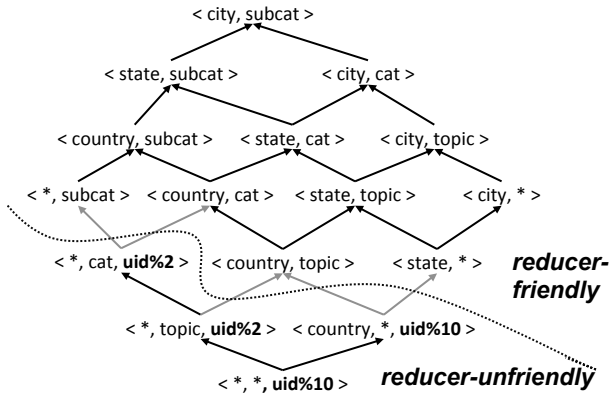

Fig. 9. Value partitioned lattice. The cube lattice is divided into reducer-friendly and reducer-unfriendly areas. Each reducer-unfriendly region is then *value partitioned* using a partitioning factor estimated from sampling.

Proposition 1: Let $|D|$ denote the total number of tuples in the data, c denote the reducer limit (i.e., the maximum number of tuples a reducer can handle), and $r = \frac{c}{|D|}$. Let N denote the sample size. If a cube group G contains less than $0.75rN$ tuples in the sample, then the probability of G being a reducer-unfriendly group (i.e., containing more than c tuples) $\texttt{ProbUF}(G) \leq 5\%$ if $N > \frac{100}{r}$.

Proof: We can derive the probabilities based on Chernoff Bound [6]. Consider cube group G which contains c tuples $(t_1, t_2, ...t_n)$, where c is the reducer limit and $r = \frac{c}{|D|}$. The random sampling process can be viewed as a series of Bernoulli trials on each tuple in G with $\mathbf{Pr}[t_i = 1] = \frac{N}{|D|}$, where N is the sample size. The expected number of tuples in G appearing in the sample is $\mathbf{E}[G] = \frac{cN}{|D|} = rN$. Let X denote the number of tuples in G appearing in the sample. According to the general case Chernoff Bound [6], for any $\delta > 0$, we have:

$$\mathbf{Pr}[X < (1-\delta)rN] < e^{-rN\frac{\delta^2}{2}}$$

Let $\delta = 0.25$, we have: $\mathbf{Pr}[X < 0.75rN] < e^{-0.031rN}$. If $N > \frac{100}{r}$, then $\mathbf{Pr}[X < 0.75rN] < 5\%$. This means if G contains at least c tuples, the probability of observing less than $0.75rN$ tuples in G in the sample is less than 5% if the sample size is large enough (i.e., $> \frac{100}{r}$). Conversely, if we observe less $0.75rN$ tuples in G, then the probability of G containing more than c tuples ($\texttt{ProbUF}(G)$) is less than 5% if the sample size is large enough. □

A reducer in our setting can easily handle 1M tuples. Therefore, for data with 1B tuples, the sample size N can be as small as 100K and easily manageable by the naive algorithm. In practice, we increase N to 2M, which allows us to have a

[3]Intuitively, $\frac{1}{r}$ is the partition factor required for groups containing all the tuples, and $\frac{s}{N}$ is the relaxation factor for groups with a subset of the tuples.

much more accurate estimation or handle up to 20B tuples.

Finally, we note that extreme data skew can occur in some datasets, which will cause value partitioning to be applied to most of the cube regions. Addressing this issue is part of our future work and we discuss some initial thoughts in Sec. VIII.

C. Batch Areas

Given the annotated cube lattice, we can again directly apply the naive algorithm, process each cube group independently with the added safeguard that partitions the groups that belong to a reducer-unfriendly region. This partially alleviates the problem of large intermediate data size. However, each tuple is still duplicated at least $|C|$ times. Furthermore, another significant drawback of the naive approach is its incompatibility with pruning for monotonic measures, i.e., each cube group is processed independent of its parent group, we can no longer prune a group's children based on the `HAVING` conditions such as those specified in Fig. 1.

To address those problems, we propose to combine regions into **batch areas**. Each batch area represents a collection of regions that share a common ancestor region. Mappers can now emit one key-value pair per batch for each data tuple; thus drastically reducing the amount of intermediate data. Reducers, on the other hand, instead of simply applying the measure function, execute a traditional cube computation algorithm over the set of tuples using the batch area as the local cube lattice. A batch area typically contains multiple regions with parent/child relationships, groups can therefore be pruned based on monotonic measures, assuming a pruning condition is specified, and the cube computation algorithm adopted can take advantage of that. One such algorithm is the Bottom Up Cubing Algorithm (BUC) [3], which we adopt[4]. Since the core cubing algorithm being executed on a single reducer is self-contained, BUC can be replaced with an algorithm of choice if needed. We also note here that if a group has been value partitioned, then monotonicity-based pruning can no longer apply since the measure for the entire group may satisfy the `HAVING` conditions even though the measure for each individual partition may not.

Forming batch areas for reducer-unfriendly regions is straightforward: we simply combine regions based on their partition factors. Forming batch areas for reducer-friendly regions however, requires some thought.

A key determinant of intermediate data size is the overall number of derived attributes to be retained for the reduce phase. As an example, for batch area b5 in Fig. 10, three derived attributes $(\texttt{city}, \texttt{state}, \texttt{topic})$ need to be maintained for each tuple. The lower the number of total derived attributes need to maintained, the smaller the size of the intermediate data. Furthermore, we would like to encourage that batch areas have uniform completion times, since any skew can impact the full utilization of reducers. Based on these observations, we formulate the batch areas identification problem as the following:

[4]While there are other more recent algorithms such as Star-Cubing [28], they require the measure to be algebraic and are therefore not applicable for our analysis tasks.

*Definition 2 (***Batch Areas Identification***):* Given the set of reducer-friendly regions C' in the cube lattice and let $(R_i \prec R_j)$ indicating the parent-child relationship (R_j being the parent) between two regions in the whole cube lattice, assign each $R \in C'$ into one of the mutually exclusive set of batch areas $(B_1, B_2, ..., B_k)$ such that the following constraints are satisfied:

- $\forall R \in C'$ with at least one parent region in C', $R \in B_i \Rightarrow \exists R'', R \prec R'', R'' \in B_i$;
- $\forall R_1, R_2 \in C', R_1 \prec R_1', R_2 \prec R_2', R_1', R_2' \notin C', R_1 \in B_i \Rightarrow R_2 \in B_j, i \neq j$;
- $\forall ij, i \neq j, |(|B_i| - |B_j|)| \leq 2$

Intuitively, the three constraints state that: i) a region with at least one parent that is also reducer-friendly must belong to a batch area that contains at least one of its parents; ii) no two regions whose parents are reducer-unfriendly can belong to the same batch area; iii) the difference in the number of regions of two batch areas can not be more than 2, a heuristic used to balance the workload of each batch area.

Since each batch area will effectively require an independent projection of the dataset, they directly impact the size of intermediate data, and hence overall performance, as discussed in Sec. IV-A. Thus, it is important to construct batch areas that minimize the amount of intermediate data generated.

A viable set of batch areas can be derived greedily by considering the lowermost reducer-friendly regions as initial batch areas, and then walking up the lattice adding each region to the least populated batch area, given the constraints. For typical lattices, it is feasible pick the solution with the lowest total cost, i.e., $\texttt{min}(\sum_i \texttt{cost}(B_i))$ by exhaustive enumeration. The $\texttt{cost}$ function reflects the amount of intermediate data per batch area, and is defined as the count of set of attributes required by that batch area. For larger lattices, the search space for an ideal set of batch areas is exponential to the size of the lattice; simulated annealing can be used to generate an acceptably good solution.

The combined process of identifying and value-partitioning unfriendly regions followed by the partitioning of friendly regions is referred to as ANNOTATE in Algo. 2. The lowest cost annotated lattice is presented in Fig. 10.

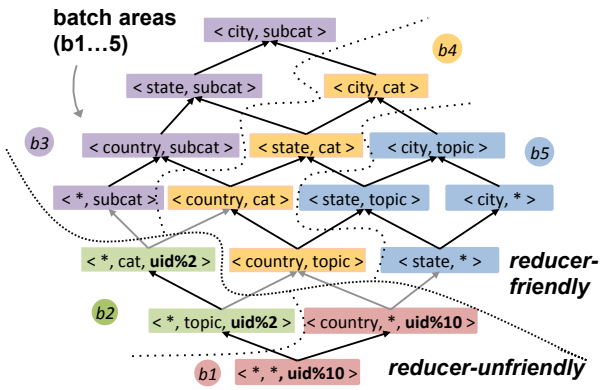

Fig. 10. Annotated Cube Lattice. Each color in the lattice indicates a batch area, b1...b5. The reducer-friendly blocks are grouped into three batch areas to exploit pruning and reduce intermediate data size. Two of the reducer-unfriendly regions are value partitioned on `uid` into 10 partitions.

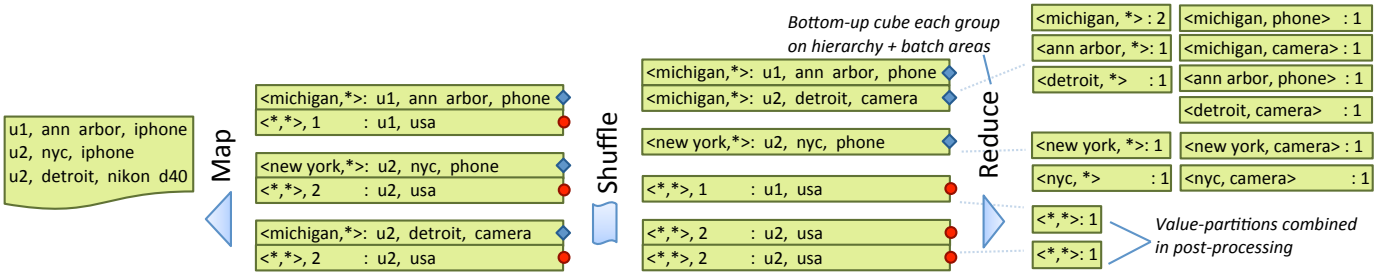

Fig. 11. Walkthrough for **MR-Cube** with `reach` as measure, as described in Sec. V-D. Both the Map and Reduce stages take the Annotated Cube Lattice(Fig. 10) as a parameter. Walkthrough is shown only for batch areas b1 (denoted by red ∘ on tuples) & b5 (denoted by blue ⋄).

D. Cube Materialization & Walkthrough

As shown in Algo. 2, an annotated lattice is generated and then used to perform the main MR-CUBE-MAP-REDUCE.

Algorithm 2 MR-Cube Algorithm

ESTIMATE-MAP(e)
1 *# e is a tuple in the data*
2 let C be the Cube Lattice;
3 **for each** c_i **in** C
4 **do** EMIT $(c_i, c_i(e) \Rightarrow 1)$ *# the group is the secondary key*

ESTIMATE-REDUCE/COMBINE($\langle$r,g$\rangle$, $\{e_1, e_2, ...\}$)
1 *#* $\langle r, g \rangle$ *are the primary/secondary keys*
2 MaxSize S $\leftarrow \{\}$
3 **for each** r, g
4 **do** S[r] $\leftarrow$ MAX(S[r], |g|)
5 *# |g| is the number of tuples* $\{e_i, ..., e_j\} \in g$
6 **return** S

MR-CUBE-MAP(e)
1 *# e is a tuple in the data*
2 let C_a be the Annotated Cube Lattice
3 **for each** b_i **in** C_a.batch_areas
4 **do** s $\leftarrow$ b_i[0].partition_factor
5 EMIT (e.SLICE(b_i[0]) + e.id%s $\Rightarrow$ e)
6 *# value partitioning: 'e.id % s' is appended to primary key*

MR-CUBE-REDUCE(k, V)
1 let C_a be the Annotated Cube Lattice
2 let M be the measure function
3 cube $\leftarrow$ BUC(DIMENSIONS(C_a, k), V, M)
4 EMIT-ALL (k, cube)
5 **if** (MEMORY-EXCEPTION)
6 **then** D′ $\leftarrow$ D′ $\cup$ (k, V)

MR-CUBE(Cube Lattice C, Dataset D, Measure M)
1 D_{sample} = SAMPLE(D)
2 RegionSizes R = ESTIMATE-MAPREDUCE(D_{sample}, C)
3 C_a = ANNOTATE(R, C) *# value part. & batching*
4 **while** (D)
5 **do** R $\leftarrow$ R $\cup$ MR-CUBE-MAPREDUCE(C_a, M, D)
6 D $\leftarrow$ D′ *# retry failed groups D' from MR-Cube-Reduce*
7 $C_a \leftarrow$ INCREASE-PARTITIONING(C_a)
8 Result $\leftarrow$ MERGE(R) *# post-aggregate value partitions*
9 **return** Result

In Fig. 11 we present a walkthrough of **MR-Cube** over our running example. Based on the sampling results, cube regions $\langle *, * \rangle$ and $\langle \texttt{country}, * \rangle$ have been deemed reducer-unfriendly and require partitioning into 10 parts. We depict materialization for 2 of the 5 batch areas, b1 and b5. For each tuple in the dataset, the MR-CUBE-MAP emits key:value pairs for each batch area, denoted by red ∘ (b1) and blue ⋄ (b5). In required, keys are appended with a hash based on value partitioning, e.g. the 2 in $\langle *, * \rangle, 2 : \texttt{u2}, \texttt{usa}$. The shuffle phase then sorts them according to key, yielding 4 reducer tasks. Algorithm BUC is run on each reducer, and the cube aggregates are generated. The value partitioned groups representing $\langle *, *, 1 \rangle$ are merged during post-processing to produce the final result for that group, $\langle *, *, 2 \rangle$.

Note that if a reducer fails due to wrong estimation of group size, all the data for that group is written back to the disk and follow-up MapReduce jobs are then run with more aggressive value partitioning, until the cube is completed. It should be noted that in practice, a follow-up job is rarely, if at all, needed.

VI. EXPERIMENTAL EVALUATION

We perform the experimental evaluation on a production Yahoo! Hadoop 0.20 cluster as described in [24]. Each node has 2×Quad Core 2.5GHz Intel Xeon CPU, 8GB RAM, 4×1TB SATA HDD, and runs RHEL/AS 4 Linux. The heap size for each mapper or reducer task is set to 4GB. All tasks are implemented in Python and executed via Hadoop Streaming. Similar to [1], we collect results only from successful runs, where all nodes are available, operating correctly, and there is no delay in allocating tasks. We report the average number from three successful runs for each task.

A. Experimental Setup

1) Datasets: We adopt two datasets. The **Real** dataset contains real life click streams obtained from query logs of Yahoo! Search. We examined clicks to Amazon, Wikipedia and IMDB on the search results. For each click tuple, we retain the following information: `uid`, `ip`, `query`, `url`. We establish three dimensions containing a total of six levels. The *location* dimension contains three levels (from lowest to highest, with cardinalities): `city`(74214) → `state`(2669) → `country`(235) and is derived from `ip`[5]. The *time* dimension

[5]Mapping is done using the free MaxMind GeoLite City database: `http://www.maxmind.com/app/geolitecity`.

contains two levels: month(6) → day(42). The *gender* dimension contains only one level, gender(3), and is derived from the user's profile information. This dataset in full contains 516M click tuples for a size of 55GB. The number of unique users and queries are in the range of tens of millions.

We also generate the synthetic **Example** dataset, which has been our running example and contains 1B tuples. Attributes and dimension hierarchies of this dataset are shown in Fig. 5. The probability distributions used to spread the attribute values across the tuples are: Normal distribution for , Zipf distribution for query, Gaussian distribution for city and Uniform distribution for time. The parameters for the distributions are chosen based on prior studies [26], [27]. The full dataset is called **Example-1B** and amounts to 55GB on disk.

2) Cube Materialization Tasks: We focus on two cube computation tasks shown in Sec. III: computing **user reach** for the coverage analysis and computing **top-k queries** for the top-k analysis. The first measure computes the number of distinct users within the set of tuples for each cube group. It is monotonic and holistic, but partially algebraic on uid. We output a cube group only if its user reach is greater than 20. The second measure, top-k queries, computes the top-5 most popular queries. It is again holistic, but partially algebraic on query. Since it is not a numerical measure, monotonicity does not apply.

3) Baseline Algorithms: We compare our **MR-Cube** algorithm against three baseline algorithms: the naive MapReduce algorithm (**Naive**) described in Algo. 1 and adaptations of two parallel algorithms proposed in Ng et al. [18], **BPP** and **PT**. The latter two algorithms are designed for cube materialization over flat dimension hierarchies using small PC clusters. To ensure a reasonable comparison, we adjust these algorithms to fit our MapReduce infrastructure.

MR-BPP: Adapted from BPP (Breadth-first Partitioned Parallel Cube), a parallel algorithm designed for cube materialization over flat dimension hierarchies. It divides the cube lattice into subtrees rooted at each dimension attribute. For each subtree, it partitions the dataset according to the attribute value and computes the measure (or intermediate results) for each data partition in parallel. The overall measures are then combined from those of individual partitions.

MR-PT: Adapted from PT (Partitioned Tree), a parallel algorithm designed for cube materialization using PC clusters with a small number of machines. It repeatedly partitions the entire cube lattice in binary fashion until the number of partitions matches the number of available machines. It then computes the measures for all cube groups in each lattice partition on a single machine. PT can be directly adapted to our MapReduce setting: when the number of machines exceeds the number of cube regions, each machine then only needs to process a single cube region.

Since both original algorithms are designed for flat dimensions, we convert our dataset into an acceptable format by flattening our multi-level hierarchies. The task allocations step (i.e., assigning cube computation tasks to machines) in both algorithms are conducted during the map phase. During the reduce phase, the BUC algorithm is then run on the tuples just like in the original algorithms. We note here that, incidentally, MR-BPP is similar to the algorithm proposed by You et al. [29] and MR-PT is similar to the algorithm proposed by Sergey et al. [23].

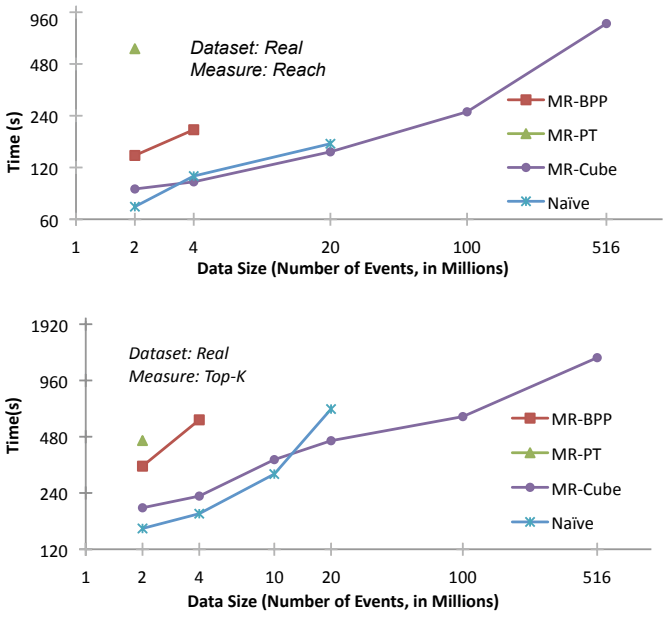

Fig. 12. Running time over Real dataset, for measures reach and top-k queries.

There are two more algorithms described in Ng et al [18], namely RP (Replicated Parallel BUC) and ASL (Affinity Skiplist). Algorithm RP is dominated by PT and is therefore ignored here. Algorithm ASL processes each cube region in parallel and uses a Skiplist to maintain intermediate results for each cube group during the process. In this study, the functionality of the Skiplist is provided by the MapReduce framework itself, turning ASL into our naive algorithm; and is thus also ignored.

B. Experimental Results

We focus on three main parameters that impact the performance of the algorithms: **data scale** (number of tuples in the dataset), **parallelism** (number of reducers) and **hierarchies** (number and depth of the dimension hierarchies, which affect the cube lattice size). During the map phase, the system automatically partitions the input data into roughly 1M tuples per mapper. As a result, the number of mappers is the same for all algorithms with the same input data and we do not examine its impact. We emphasize that, for MR-Cube, we measure the total time including the sampling process and batch areas generation, each of which take just under a minute to complete for all runs. In addition to the experimental results described here, we provide additional analysis and anecdotal results over a *real life cubing task* in Sec. VI-C.

1) Impact of Data Scale: We first analyze the cubing time over the **Real** datasets of different data scales for computing user reach and top-k queries (where k is 5). The dimension hierarchies adopted are *location*, *domain*, and *gender*, as

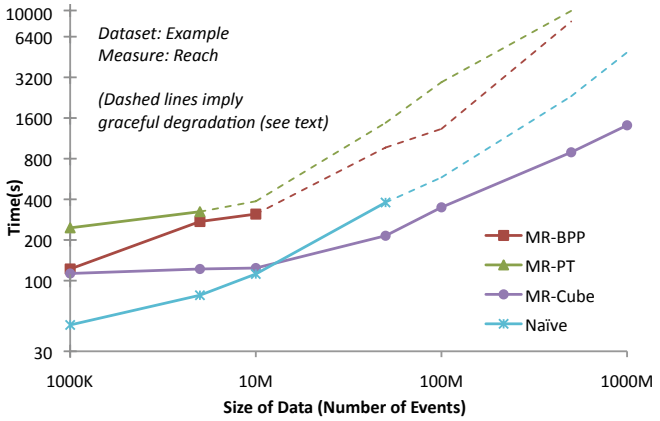

Fig. 13. Running time over Example dataset. Dashed lines indicate use of graceful degradation.

described in Sec. VI-A1. This results in a cube lattice with 24 regions. The full dataset contains 516M tuples and datasets of smaller scales are random subsets of the full dataset. The parallelism is set to 128. Except the number of tuples, this general setting is similar to those adopted in earlier studies [3], [18] in terms of the product of the attribute cardinalities.

As shown in Fig. 12, MR-Cube scales much better than the baseline algorithms. MR-BPP and MR-PT both fail to cube after 4M tuples because they are not able to leverage the full parallel power provided by MapReduce. Naive performs better at small data scales because of its low overhead. It, however, fails after the 20M tuples mark. MR-Cube is the only algorithm that can successfully cube the full dataset.

Fig. 13 illustrates the running times for computing reach for the **Example** datasets. The hierarchies being adopted are *location* and *query topic*, as shown in Fig 5, for a cube lattice of 16 regions. The full dataset contains 1B tuples and we again randomly form the smaller datasets. For this particular experiment, we incorporate *graceful degradation*: i.e., when a cube group is too large to be computed, we allow the algorithm to proceed by ignoring the offending group. This prevents the algorithm from failing, but leads to the cube being only partially materialized. As expected, Naive, MR-BPP, and MR-PT cannot scale up to the full dataset, failing at 100M, 10M and 50M tuples, respectively, if graceful degradation is not used (solid lines). MR-Cube, however, scales well and fully computes the cube within 20 minutes for the full dataset. Even when graceful degradation is used (dashed lines), MR-Cube still performs significantly better than the baseline algorithms.

Insights: MR-Cube performs worse than Naive at small data scale, especially when computing the top-k queries. This is because monotonic property can not apply here and hence the BUC algorithm employed by MR-Cube cannot prune efficiently. Also, in Figs. 12 and 13, using MR-Cube becomes viable when the speedup accrued due to better value partition and lattice partitioning outweighs the overhead involved in the sampling process. Further, for MR-PT, its approach of effectively copying the dataset to each reducer is clearly not practical under MapReduce. It also does not take advantage of any parallelism beyond the number of regions.

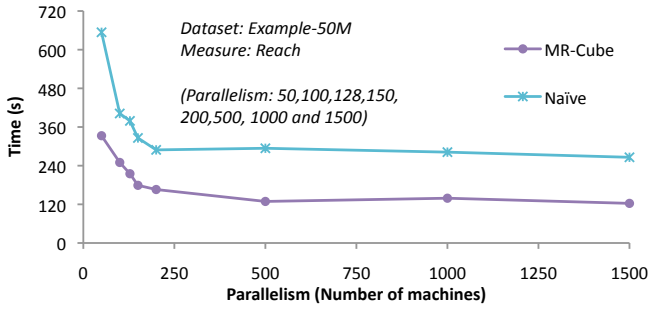

Fig. 14. Running time over Example dataset with varying parallelism.

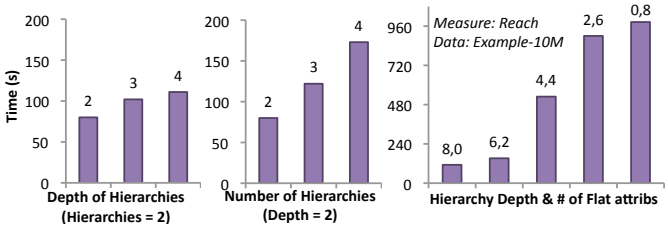

Fig. 15. Running time over Example-10M dataset with different hierarchy configurations.

2) Effect of Parallelism: Next, we analyze the impact of increasing parallelism on the cube computation time in Fig. 14 using the **Example** dataset with 50M tuples. We report numbers for Naive and MR-Cube only: due to the overhead of task setup for large jobs, the effect of parallelism can only be appreciated with large datasets, at which both MR-PT and MR-BPP fail. We observe that MR-Cube scales linearly up to 200 nodes, after which the setup and runtime overhead of the MapReduce infrastructure factors in and cancels out the benefit from increasing parallelism.

3) Effect of Hierarchies: We further analyze the effect of hierarchies on cube computation using the **Example-10M** dataset. In the first two experiments in Fig. 15, we either fix the depth and vary the number of hierarchies (middle panel) or fix the number and vary the depth of the hierarchies (left panel). Not surprisingly, increasing the depth has a more gradual increase in runtime than the number of hierarchies. In the third experiment (right panel), we fix the total number of levels across all dimensions and vary the configuration: going from 8 levels in a single hierarchy $(8,0)$ to 8 flat dimensions $(0,8)$. The result indicates that having 8 flat dimensions is a lot more costly for cube computation than having a single 8-level dimension hierarchy, which is not surprising since the former produces a much larger cube lattice than the latter.

In summary, MR-Cube clearly outperforms currently available distributed algorithms for cubing large data on holistic measures. From an implementation standpoint, it is advisable to provide both Naive and MR-Cube algorithms to the user. In the case when the data is small, or the measure is algebraic, the Naive algorithm is recommended. As the data and lattice size increase, and when the measure is holistic, the user can switch to MR-Cube.

Hier. Regions	192	Annotation M/R	1m 14s
Naive Regions	512	Cube Materialize	20m 3s
Head Regions	28	Post-process	4m 12s
Avg Map time	3m 21s	Map Out Tuples	14B
Avg Shuffle Time	8m 29s	Map Out Bytes	1.1TB
Avg Reducer Time	4m 12s	Cube Size	216GB

Fig. 16. Performance statistics of the anecdotal analysis.

NY	RI	ME	WI
Monday	*Sunday*	*Sunday*	*Monday*
Tuesday	*Friday*	*Monday*	*Sunday*
Friday	*Monday*	*Tuesday*	*Tuesday*
Thursday	*Thursday*	*Friday*	*Friday*

(a) Top 4 frequent days for female users clicking on IMDB URLs, by U.S. state.

Jan	Feb
Joanna Pacitti	*Joanna Pacitti*
Lisa Bonet	*Tonya Harding Today*
Martin Luther King Jr	*Eliza Dushku*
Kim Kardashian	*Rihanna*
Malia Obama	*Chris Brown*
March	**April**
Crystal Mckellar	*Swine Flu*
Natasha Richardson	*Twitter*
Watchmen	*Keshia Night Pullam*
XBox 360 Ring	*Sabrina Lebeauf*
Marcus Jordan	*Lady Gaga*
May	**June**
Montauk Monster	*Hyalinobatrachium pellucidum*
Kris Allen	*David Carradine*
Adam Lambert	*Dream Interpretations*
Kate Gosselin	*Frank Lloyd Wright Houses*
Derecho Storms	*Air France Flight 447*

(b) Queries with highest reach on Wikipedia URLs, by Month

Fig. 17. Anecdotal Results from the search log analysis

C. Anecdotal Results

In this section, we present some anecdotal results from an actual cube analysis task over a Yahoo! Search log sample with 500M items. The analysis involves 6 dimensions containing a total of 9 individual attributes (including the attribute `query` for individual queries) and computes the measures *reach* and *top-5 most frequent days*. It is performed on a Hadoop 0.20 cluster with 2048 mappers and reducers. Fig. 16 illustrates relevant performance statistics on the analysis. Some example results are shown in Fig. 17.

We gained some further insights while performing this cubing task. First, MR-Cube properly partitioned the cube lattice and distributed the computational work load evenly across all nodes. As a result, no single long-running reducer held up the progress of the task. Second, due to the large number of cube groups (resulting from the fact that one of the dimension attributes, `query`, has millions of unique values), the *shuffle* phase took longer than the map or reduce phases. Third, as expected from our discussion in Sec. V-B, our conservative estimation of partition factors avoided any skew-based reducer failures. Finally, we noticed that skew in the `query` attribute (e.g., queries like "amazon" or "imdb" were issued by millions of unique users) leads to many regions being value partitioned, but the average groups within those regions were very small. Since partitioned groups are not pruned, this further led to a large number of groups which had to be pruned in the post-processing step. While we do not address this issue of extreme data skew in this paper, we provide some initial discussion of it in Sec. VIII.

VII. Related Work

Since the introduction of data cube by Gray et al [12], many techniques [2], [3], [10], [13], [14], [22], [28] have been proposed for efficient cube computation. Leveraging the algebraic or monotonic properties of the measures has been at the center of those techniques. In particular, the BUC algorithm [3] leverages monotonic measures like `COUNT` to efficiently compute the iceberg cube. All these studies focus on non-parallel algorithms and are therefore not scalable to the billions of tuples that we aim to analyze. Further, many of these approaches assume the cube measures to be algebraic and are therefore not applicable to the analysis tasks that we are interested in.

Ng et al [18] first introduced a series of parallel algorithms for cube computation over small PC clusters. Two of our baseline algorithms (MR-BPP and MR-PT) are adopted from this study. However, as we demonstrate in our experiments, those algorithms are designed for small PC clusters and therefore can not take advantage of the MapReduce infrastructure. Other recent algorithms for parallel cubing either require a special parallel architecture that is different from MapReduce [15] or require the measures to be algebraic [5], [23], [29]. Chen et al [4] recently proposed an algorithm for parallel evaluation of composite aggregate queries. However, it focuses on computing composite measures for specific cube regions and does not handle the reducer-unfriendliness that we study here.

Our work is also complementary to recent studies on MapReduce languages like Pig [19] and Sawzall [21], which provide a user-friendly layer over MapReduce for ad-hoc aggregate analyses. The cube computation task can be incorporated as an operator into those languages to provide users with a friendly way to explore their data without issuing many ad-hoc aggregate queries.

Implementations of large-scale distributed aggregation have been documented in proprietary systems. The Google Dremel [17] system uses a hierarchical architecture to compute aggregates which cannot be directly applied to holistic measures. Aster Data's SQL/MapReduce [11] uses the MapReduce model in combination with a mature database system; we expect cube-style queries to be executed similar to our Naive approach in their environment.

We note that computing certain holistic measures approximately with memory bounds or in the presence of heavy skew is the subject of many previous studies [8], [25]. Our work differ from those by providing a generic framework that works for a large number of holistic measures without the need for ad hoc specialization. Furthermore, we are able to compute the measures exactly instead of approximately.

VIII. Future Work

There are several interesting issues beyond the scope of the current paper that we would like to address as part of future work. The first issue is the challenge of *extreme data skew*, which occurs if a few cube groups are unusually large even when they belong to a cube region at the top of the lattice (i.e., those with fine granularities such as $\langle \texttt{query}, \texttt{city} \rangle$). This causes value partitioning to be applied to the entire cube and therefore reduces the efficiency of our algorithm. The second issue is identifying *interesting cube groups* on top of the cube computation layer. Indeed, computing measures for each cube groups is just the initial, albeit essential, step for the full analysis task, and a natural follow up step is to automatically discover unusual cube groups. We briefly describe here some preliminary studies we have done on both.

Group-level Partitioning using Sketches: We currently perform value partitioning on a region-by-region basis: if a cube region is estimated to contain a reducer-unfriendly group, all groups within the region are value partitioned, many of which may not be necessary. This approach works well until there is extreme data skew which can lead to most cube regions being value partitioned. We are actively investigating an alternative approach of marking reducer-unfriendly *groups* instead of regions. Since the number of groups can be very large, it may not be feasible to compute quickly or maintain some statistics in the mapper's memory, as can be easily done for regions. We are looking into using compressed counting data structures such as CM-Sketch [8] as a solution.

Identifying Interesting Groups: Computing the cube (i.e., computing measures for all cube groups satisfying the pruning conditions) is only the first step, identification of interesting cube groups often needs to follow. Such tasks are trivial when the size of the full cube is tenable and when the *interestingness* can be defined as a simple value predicate. However, analysts often require more complex measures of interestingness. For example, the example task in Example 2 requires the system to compare the top-k queries of parent and child groups. This relative nature of interestingness requires non-trivial computation and breaks the ability to distribute computation to multiple nodes. We are investigating *duplicate and co-locate* approaches where certain groups are duplicated and co-located to their parent/child groups, such that uninteresting groups can be pruned without being computed.

IX. Conclusion

In this paper, we study cube computation of holistic measures over extremely large data such as search logs using the MapReduce framework. We identify a subset of holistic measures that are *partially algebraic* and propose the technique of *value partitioning* to make them easy to compute in parallel. We design algorithms that partition the cube lattice into *batch areas* to effectively exploit both the parallel processing power of MapReduce and the pruning power of cube materialization algorithms. Experiments over real and synthetic data show that our **MR-Cube** algorithm efficiently distributes the computation workload across the machines and is able to complete cubing tasks at a scale where prior algorithms fail.

References

[1] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, et al. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. *VLDB*, 2009.

[2] S. Agarwal, R. Agrawal, P. M. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan, and S. Sarawagi. On the Computation of Multidimensional Aggregates. *VLDB*, 1996.

[3] K. Beyer and R. Ramakrishnan. Bottom-Up computation of sparse and iceberg CUBEs. *SIGMOD*, 1999.

[4] L. Chen, C. Olston, and R. Ramakrishnan. Parallel evaluation of composite aggregate queries. In *ICDE*, 2008.

[5] Y. Chen, F. K. H. A. Dehne, T. Eavis, and A. Rau-Chaplin. PnP: sequential, external memory, and parallel iceberg cube computation. *Distributed and Parallel Databases*, 2008.

[6] H. Chernoff. A Measure of Asymptotic Efficiency for Tests of a Hypothesis Based on the Sum of Observation. *Math. Statistics*, 1952.

[7] J. Cohen, B. Dolan, M. Dunlap, J. Hellerstein, and C. Welton. MAD skills: New Analysis Practices for Big Data. *VLDB*, 2009.

[8] G. Cormode and S. Muthukrishnan. An Improved Data Stream Summary: The Count-Min Sketch and its Applications. *Journal of Algorithms*, 2005.

[9] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *OSDI*, 2004.

[10] M. Fang, N. Shivakumar, H. Garcia-molina, R. Motwani, and J. D. Ullman. Computing Iceberg Queries Efficiently. *VLDB*, 1998.

[11] E. Friedman, P. Pawlowski, and J. Cieslewicz. SQL/MapReduce: A Practical Approach to Self-Describing, Polymorphic, and Parallelizable User-Defined Functions. *VLDB*, 2009.

[12] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data Cube: A Relational Operator Generalizing Group-By, Cross-Tab and Sub-Totals. *ICDE*, 1996.

[13] J. Hah, J. Pei, G. Dong, and K. Wang. Efficient Computation of Iceberg Cubes with Complex Measures. *SIGMOD*, 2001.

[14] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing Data Cubes Efficiently. *SIGMOD*, 1996.

[15] R. Jin, K. Vaidyanathan, et al. Communication & Memory Optimal Parallel Datacube Construction. *Parallel Distrib. Syst.*, 2005.

[16] X. Li, J. Han, and H. Gonzalez. High-dimensional OLAP: A Minimal Cubing Approach. In *VLDB*, 2004.

[17] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive Analysis of Web-Scale Datasets. *VLDB*, 2009.

[18] R. T. Ng, A. S. Wagner, and Y. Yin. Iceberg-cube computation with PC clusters. *SIGMOD*, 2001.

[19] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A not-so-foreign language for data processing. In *SIGMOD*, 2008.

[20] A. Pavlo, E. Paulson, A. Rasin, D. Abadi, D. DeWitt, S. Madden, and M. Stonebraker. A Comparison of Approaches to Large-Scale Data Analysis. *SIGMOD*, 2009.

[21] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming*, 2005.

[22] K. A. Ross and D. Srivastava. Fast Computation of Sparse Datacubes. *VLDB*, 1997.

[23] K. Sergey and K. Yury. Applying Map-Reduce Paradigm for Parallel Closed Cube Computation. *DBKDA*, 2009.

[24] K. V. Shvachko and A. C. Murthy. Scaling Hadoop to 4000 nodes at Yahoo! *Yahoo! Developer Network Blog*, 2008.

[25] D. Talbot. Succinct Approx. Counting of Skewed Data. *IJCAI*, 2009.

[26] J. Walker. Mathematics: Zipf's Law and the AOL Query Database. *Fourmilog: None dare call it reason.*, 2006.

[27] Y. Xie and D. O Hallaron. Locality in search engine queries and its implications for caching. *INFOCOM*, 2002.

[28] D. Xin, J. Han, X. Li, and B. W. Wah. Star-Cubing: Computing Iceberg Cubes by Top-Down And Bottom-Up Integration. *VLDB*, 2003.

[29] J. You, J. Xi, P. Zhang, and H. Chen. A Parallel Algorithm for Closed Cube Computation. *ICIS*, 2008.

[30] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. *OSDI*, 2008.