

# Composite Subset Measures

Lei Chen<sup>1</sup>, Raghu Ramakrishnan<sup>1,2</sup>, Paul Barford<sup>1</sup>, Bee-Chung Chen<sup>1</sup>, Vinod Yegneswaran<sup>1</sup>

<sup>1</sup> Computer Sciences Department, University of Wisconsin, Madison, WI, USA

<sup>2</sup> Yahoo! Research, Santa Clara, CA, USA

{chenl, pb, beechung, vinod}@cs.wisc.edu

ramakris@yahoo-inc.com

## ABSTRACT

*Measures* are numeric summaries of a collection of data records produced by applying aggregation functions. Summarizing a collection of subsets of a large dataset, by computing a measure for each subset in the (typically, user-specified) collection is a fundamental problem. The multidimensional data model, which treats records as points in a space defined by *dimension* attributes, offers a natural space of data subsets to be considered as summarization candidates, and traditional SQL and OLAP constructs, such as GROUP BY and CUBE, allow us to compute measures for subsets drawn from this space. However, GROUP BY only allows us to summarize a limited collection of subsets, and CUBE summarizes *all* subsets in this space. Further, they restrict the measure used to summarize a data subset to be a one-step aggregation, using functions such as SUM, of field-values in the data records.

In this paper, we introduce *composite subset measures*, computed by aggregating not only data records but also the measures of other related subsets. We allow summarization of naturally related regions in the multidimensional space, offering more flexibility than either GROUP BY or CUBE in the choice of what data subsets to summarize. Thus, our framework allows more meaningful summaries to be computed for a targeted collection of data subsets.

We propose an algebra called *AW-RA* and an equivalent pictorial language called *aggregation workflows*. Aggregation workflows allow for intuitive expression of composite measure queries, and the underlying algebra is designed to facilitate efficient multi-scan execution. We describe an evaluation framework based on multiple passes of sorting and scanning over the original dataset. In each pass, several measures are evaluated simultaneously, and dependencies between these measures and containment relationships between the underlying subsets of data are orchestrated to reduce the memory footprint of the computation. We present a performance evaluation that demonstrates the benefits of our approach.

## 1. INTRODUCTION

In the multidimensional model of data, records in a central *fact table* are viewed as points in a multidimensional space. Attributes are divided into *dimension attributes*, which are the coordinates of the data point, and *measure attributes*, which are values associated with points. The domain of values for each dimension is organized

into a hierarchy, leading to a very natural notion of multidimensional regions. Each region represents a data subset. Summarizing the records that belong to a region by applying an aggregate operator, such as SUM, to a measure attribute (thereby computing a new measure that is associated with the entire region) is a fundamental operation in OLAP.

Often, however, more sophisticated analysis can be carried out based on the computed measures, e.g., identifying regions with abnormally high measures. For such analyses, it is necessary to compute the measure for a region by also considering other regions (which are, intuitively, “related” to the given region) and their measures. In this paper, we introduce *composite subset measures*, which differ from the traditional GROUP BY and CUBE approaches in three ways:

- 1) The measures for a region can be computed as the summaries of other “related” regions in a compositional manner. The relationships capture various types of proximity in multidimensional space.
- 2) In contrast to the CUBE construct, we do not offer a way to compute the summary of every region; this is typically overkill for the kinds of complex measures we seek to compute.
- 3) The language and algebra are carefully designed to enable highly scalable, parallelizable, and distributed evaluation strategies based on streaming the data in one or more passes, possibly with interleaved sorts.

This study is motivated by our ongoing work in two different application domains: environmental monitoring [18] and analysis of network traffic data [7]. Similar problems have been faced by researchers dealing with data at Internet companies such as Google and Yahoo!, leading them to also develop systems for scalable aggregation of tabular data [10,17,22]. In contrast to the proposal in [22], we explore a declarative, query-style approach in the spirit of OLAP constructs such as [4, 14, 19,20, 26]. Further, the focus of [10] is a highly parallelizable, distributed evaluation framework for simple one-step aggregation queries. This is an issue that we do not tackle in this paper, but such highly distributed computation has been a strong consideration in the design of our language, and we intend to address it in future work. In keeping with this goal, we have avoided implementation choices that require us to assign unique ids to records or to maintain indexes over (potentially widely distributed) tables, and focused on evaluation strategies that orchestrate aggregation steps across one or more scans of data (partitions).

Consider the following network traffic analysis example. Self-propagating worm outbreaks have the potential to wreak havoc in the Internet, and can take place on a variety of time scales. We can potentially identify new outbreaks based on the escalation of the traffic into a network from one time period to the next. This kind of escalation, which is defined on a per-time period and sub-

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.

VLDB '06, September 12-15, 2006, Seoul, Korea.

Copyright 2006 VLDB Endowment, ACM 1-59593-385-9/06/09

network basis, is a composite measure built on the traffic measures for two adjacent time periods.

When used to compute composite measures, existing tools, such as relation algebra or multidimensional query languages, frequently result in nested expressions that are hard for human analysts to understand and for the processing system to optimize. Further, their use requires us to import data into a DBMS, which can itself be a challenge for very large datasets [10]. Our goal is to develop a standalone, lightweight yet highly scalable analysis system that supports composite measure queries.

Our contributions are as follows:

- 1) We propose a pictorial language (called *aggregation workflows*) and algebra for expressing composite aggregation queries and representing streaming plans for such queries.
- 2) We show that any query in our language can be expressed in our algebra, and present a comprehensive framework for highly scalable, scan-based evaluation in one or more passes. We show how to exploit sorting between passes and orchestration of dependencies between different aggregation steps.
- 3) We present an evaluation that demonstrates the potential of our methods for significant performance gains over traditional relational approaches.

This work is a first step in our broader research agenda to develop efficient, streamlined tools for domain specialists to mine large, complex datasets. The complete, technical report version of this paper [8] discusses the optimization problem of finding good multi-pass streaming plans and describes a greedy optimizer. Beyond that, the approach offers potentially unlimited parallelism and ability to distribute computation, but our current implementation does not take advantage of these opportunities.

The rest of the paper is organized as follows. In Section 2, we describe the dataset for the running example, and define the main concepts underlying the multidimensional model, including domains, domain hierarchies, and regions. In Section 3, we introduce composite subset measures and the algebra, AW-RA. In Section 4, we describe the pictorial language of aggregation workflows and consider translation into AW-RA. We describe the evaluation framework in Section 5, and discuss query optimization briefly in Section 6. Experimental results based on synthetic as well as real datasets and workloads are presented in Section 7. We discuss related work in Section 8, and conclude in Section 9.

## 2. DEFINITIONS AND EXAMPLE DATA

Attacks and intrusions in the Internet are increasing in both volume and diversity, and represent a significant threat to government, industry, academia and home users writ large. Developing effective and efficient means for identifying a broad range of cyber threats is a focus for our work. We will use the network intrusion logs from Dshield.org [11] as a running example in this paper. This dataset is collected continuously from over 1,600 networks world wide, and contains normalized records of

**Table 1. The Schema for the Network Log Dataset**

Name	Description	Abbreviation
Timestamp	Packet arrival time	$t$
Source	The source IP address	$U$
Target	The target IP address	$T$
Targetport	The target port	$P$

attack packets identified in each network. Table 1 lists the attributes used in our examples.

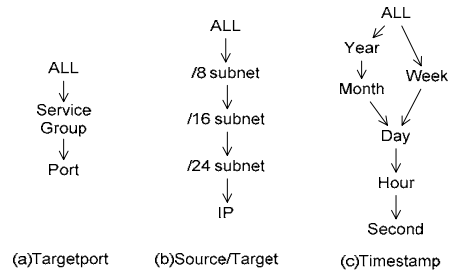
The schema of a multidimensional dataset  $\mathcal{D}$  with  $d$  dimension attributes has a *dimension vector*  $\mathbf{X} = (X_1, X_2, \dots, X_d)$ , and possibly additional *measure* attributes. While there are no explicit measure attributes in the Dshield dataset, they are typical in multidimensional datasets. Each record  $r$  in  $\mathcal{D}$  is denoted as a tuple of dimension values followed by measure values, if any:  $r = (x_1, x_2, \dots, x_d, m_1, \dots)$ , where  $x_i$  is the value for dimension attribute  $X_i$ . In the Dshield dataset,  $\mathbf{X} = (\text{Time}, \text{Source}, \text{Target}, \text{TargetPort})$ , which is abbreviated as  $\mathbf{X} = \{t, U, T, P\}$ .

### 2.1 Domains and Domain Hierarchies

A *base domain*  $D_{base}(X_i)$  is associated with each dimension attribute  $X_i$ . For example, the base domain for attributes Source and Target is 4-byte integers, and the base domain for Time is the number of seconds since the first second of 1970 (UNIX time).

A base domain can be *generalized*. For example, we can generalize the base domain of Source IP into the /24 subnet domain (256 contiguous IP addresses). Each value in this domain is a 3-byte integer representing one /24 subset. Given two domains  $D_i$  and  $D_j$ ,  $D_i <_D D_j$  indicates that  $D_j$  is a *domain generalization* of  $D_i$ ; we also say that  $D_i$  is more specific than  $D_j$ .

All the domains associated with a given dimension attribute form a *domain generalization hierarchy*, which is a directed acyclic graph (DAG). Each node in this graph represents one domain. The relationship  $<_D$  defines a partial order in the graph.  $D_i <_D D_j$ , if there is an arc chain from  $D_j$  to  $D_i$ . A domain hierarchy is *linear* when the domain hierarchy graph forms a single path. For any dimension attribute, there is a special domain called  $D_{ALL}$  with a single value ALL, which is the generalization of all possible values for the given dimension. Figure 1 illustrates the domain hierarchies for the Dshield dataset. The domain hierarchies for TargetPort, Source and Target are linear, whereas the domain hierarchy for Time is non-linear since a week might span across two months. To simplify the discussion, in the rest of this paper, we will ignore the Week domain and treat Time as a linear attribute. In this paper, we will only use dimension with linear hierarchy domain hierarchy.



**Figure 1. Domain Hierarchies for the Network Log Data**

The collection of all domains associated with a dimension attribute  $X_i$  is denoted as  $\text{Hier}(X_i) = \{DX_i^{(0)}, DX_i^{(1)}, \dots, DX_i^{(l)}\}$ , where  $DX_i^{(0)} = D_{base}(X_i)$  and  $DX_i^{(l)} = D_{ALL}$ . For example,  $\text{Hier}(\text{Time}) = \{\text{Second}, \text{Hour}, \text{Day}, \text{Month}, \text{Year}, D_{ALL}\}$ . The *extended domain* for a given dimension attribute is the union of all its associated domains, which is denoted as  $EX_i = \bigcup_{1 \leq k \leq l} DX_i^{(k)}$ . For any value  $x \in EX_i$ , the function  $\delta(x)$  returns the domain to which  $x$  belongs. For example,  $\delta(2002) = \text{Year}$ ,  $\delta(2002-02-14) = \text{Day}$ . To avoid confusion, we assume that there is no overlap between two different domains associated with same attribute.

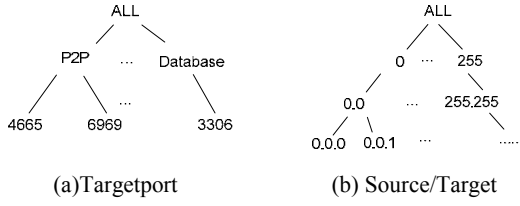


Figure 2. Value Hierarchies for Network Data

For  $D_i <_D D_j$ , we can define a many-to-one *value generalization function*  $\gamma_{D_j}$  that maps value  $x \in D_i$  into  $x' \in D_j$ . For example,  $\gamma_{/24 \text{ subnet}}(120.322.32.4) = 120.322.32$ . We require the value generalization functions to be consistent: given  $x_i \in D_i$ ,  $x_j \in D_j$ ,  $x_k \in D_k$  and  $D_i <_D D_j$ ,  $D_j <_D D_k$ , if  $\gamma_{D_j}(x_i) = x_j$ ,  $\gamma_{D_k}(x_j) = x_k$ , then  $\gamma_{D_k}(x_i) = x_k$ .

Using the value generalization function for a given dimension attribute, we can construct a value hierarchy graph in which each node represents a value in the extended domain of the dimension attribute, and arcs represent the value generalization mapping. Figure 2 illustrates the value hierarchies for the Dshield dataset.

The following observation is used later for query evaluation:

**Proposition 1:** For a given attribute, if the domain hierarchy is linear, which means the value hierarchy graph is a tree, then there is a total order  $<$  in the extended domain  $EX_i$  such that for  $u, v \in EX_i$ ,  $u < v \rightarrow \gamma_{D_j} u \leq \gamma_{D_j} v$ , for all  $D$  s.t.  $\delta(u) <_D D$  and  $\delta(v) <_D D$ . (see [8] for proof)

For certain dimension attributes, such as Timestamp, there is a natural order over domain values that satisfies the above property. If such an order does not naturally exist, we can encode the values in the extended domain so as to impose such an ordering over the encoded domain; when the number of distinct values in the extended domain is not too large, the encoding/decoding operations can be carried out very efficiently.

## 2.2 Cube Space, Regions and Region Sets

The *cube space* of a dataset  $\mathcal{D}$  denoted as  $\mathcal{C} = EX_1 \times EX_2 \times \dots \times EX_d$ , is a  $d$ -dimensional space. A *region*  $c = (v_1, v_2, \dots, v_d)$ ,  $v_i \in EX_i$ , is a hyper-rectangle which covers a data subset. The granularity of the region is specified by the granularity vector  $gran(c) = (X_1: \delta(v_1), X_2: \delta(v_2), \dots, X_d: \delta(v_d))$ , where  $\delta(v_i) \in \text{Hier}(X_i)$ . We use  $<_G$  to represent the relations between two granularity vectors. For two granularity vector  $G_1$  and  $G_2$ ,  $G_1 <_G G_2$  implies  $\forall i, G_1.D_i \leq_D G_2.D_i$ . To simplify the notation, we often omit attributes for which the domains are  $D_{\text{ALL}}$  in the granularity vector. So  $(U:IP)$  is an equivalent expression to  $(t:D_{\text{ALL}}, U:IP, T:D_{\text{ALL}}, P:D_{\text{ALL}})$ .

A region  $c$  covers a subset of the original data set, denoted as

$$\text{coverage}(c) = \{r \mid r \in \mathcal{D} \wedge \forall i, \gamma_{\delta(c,v_i)}(r.X_i) = c.v_i\}$$

A region  $c_1$  is the *parent* of another region  $c_2$ , denoted as  $c_2 <_C c_1$ , if  $\forall i, c_2.D_i <_D c_1.D_i \wedge \gamma(c_2.X_i) = c_1.X_i$ .

The *region set*  $[X_1:D_1, \dots, X_d:D_d]$  (we use square brackets to denote region sets and parentheses to denote granularity vectors) is the set of all regions in cube space with same granularity, i.e.,  $\{c \mid c \in \mathcal{C} \wedge gran(c) = (X_1:D_1, \dots, X_d:D_d)\}$ . For example, the region set  $[U:IP, T:IP]$  contains all the regions, with each region covers the data records sharing same source and target IP addresses.

## 3. COMPOSITE SUBSET MEASURES

In this section, we introduce the concept of composite subset measures and propose the algebra for computing them.

### 3.1 Examples of Composite Measures

Composite subset measures are defined on regions in cube space. For a typical analysis, we want to compute the measures for all regions with the same granularity (i.e., a region set) and identify regions with interesting measure values. We illustrate with five example queries, expressed as intuitive calculus formulas.

**Example 1 (Traffic counting):** For each hour, compute the outgoing packets from every source IP.

$$\forall c \in [t: \text{Hour}, U: \text{IP}], c.\text{Count} = |\text{coverage}(c)|$$

The measure is computed by aggregating the data records in the coverage of a region. It can be simply answered via a GROUP BY aggregation, in which the data table is first partitioned by the time and source attributes, and the measure for each group is computed by aggregating the measures for individual records in that group. Strictly speaking, this measure is not a composite measure.

**Example 2 (Busy source count):** For each hour, compute the number of sources which have at least five outgoing packets.

$$\forall c \in [t: \text{Hour}], c.s\text{Count} = |\{c' \in [t: \text{Hour}, U: \text{IP}], c.t = c'.t, c'.\text{Count} > 5\}|$$

The measure is constructed by composing the measures from the previous example. For each region  $c$ , we first identify all child regions ( $c'$ ) that satisfy the condition on the number of outgoing packets, and then count satisfactory child regions.

**Example 3 (Busy source traffic):** For each hour, compute the traffic generated by those sources with at least five outgoing packets.

$$\forall c \in [t: \text{Hour}], c.s\text{Traffic} = \text{sum} \{c'.\text{Count} \mid c' \in [t: \text{Hour}, U: \text{IP}], c.t = c'.t, c'.\text{Count} > 5\}$$

This example is the same as Example 2, except that a different aggregation function is used.

**Example 4 (Busy source count moving average):** For every six hour time window, compute the average of the hourly count for busy sources.

$$\forall c \in [t: \text{Hour}], c.\text{avgCount} = \text{average} \{c'.s\text{Count} \mid c' \in [t: \text{Hour}], c'.t \in [c.t, c.t+5]\}$$

In this example, the measure avgCount for a region is evaluated based on the measures of regions that share the same granularity and are “near” the original region in cube space.

**Example 5 (Ratio):** For every hour, compute the ratio between the average of six hour busy source count and the average traffic carried by a busy source in that hour.

$$\forall c \in [t: \text{Hour}], c.\text{ratio} = \frac{c.\text{avgCount}}{c.s\text{Traffic} / c.s\text{Count}}$$

This example differs from the previous example in that the new measure does not directly depend on the measures of other regions. Instead, it is computed by combining multiple measures for the same region.

It is not hard to see that the above examples can be answered by combining relation algebra operators (e.g., selection and join) and the aggregation operator. However, using relational algebra to express composite subset measures has several disadvantages: Relational algebra, when applied to composite subset measures, typically results in cumbersome expressions, since almost every processing step requires at least two operations (join and aggrega-

tion). Once the algebra is translated into SQL, the resulting query often contains multiply nested sub-queries, making it both hard to understand and difficult to optimize.

On the other hand, multidimensional data query languages, such as MDX [20], make it easy to write queries that aggregate records in regions of cube space, but for complex compositional measure the result expression is still very complicated.

### 3.2 AW-RA Algebra

We introduce a new algebra called *Aggregation Workflow Relational Algebra (AW-RA)* that is designed for expressing composite subset measure queries. While an AW-RA expression can be rewritten in relational algebra with a GROUP BY aggregation extension, it allows us to write composite measure queries succinctly, and exposes opportunities for efficient scan-based evaluation plans.

Recall that  $X$  is the dimension vector for the given cube space.  $G = (X_1:D_1, X_2:D_2, \dots, X_d:D_d)$  is the vector that is used to indicate the granularity of a region. The measures for regions having the same granularity can be stored in a table  $T$ . The schema of  $T$  is  $T:\langle G, M \rangle$ , where  $G$  is the granularity vector and  $M$  is the measure attribute.  $T.D_i$  is the domain associated with dimension attribute  $X_i$  and  $T.X_i$  is the value of dimension attribute.  $T.X_i \in T.D_i$ .  $T.X$  is the abbreviation of  $\langle T.X_1, \dots, T.X_d \rangle$ .

AW-RA expressions produce tables representing composite measures. Each table represents one measure for regions that share the same granularity. Let  $\mathcal{A}$  denote the collection of expressions that can be generated via AW-RA, then  $\mathcal{A}$  can be constructed based on the following rules:

1. **Fact Table.** The fact table  $D$  is an atomic expression;  $D \in \mathcal{A}$ . The schema for the fact table is  $D:\langle G_0, M_0 \rangle$ , with granularity  $G_0 = \langle X_1:DX_1^0, X_2:DX_2^0, \dots, X_d:DX_d^0 \rangle$ , where all dimension attributes are defined over their base domains.  $M_0$  is the measure defined over the raw data records.

2. **Aggregation operator.** If an expression  $T \in \mathcal{A}$  and there is a granularity vector  $G = (X_1:D_1, X_2:D_2, \dots, X_d:D_d)$  such that  $T$ 's granularity is  $\langle_G G$ , we can "roll up" table  $T$  and produce a higher-level aggregation. We use the notation  $g_{G,agg}(T)$  to represent such an operation. This is equivalent to the SQL query shown in Table 2, where  $\gamma_G T.X$  is the abbreviation of  $\langle \gamma_{D_1} T.X_1, \dots, \gamma_{D_d} T.X_d \rangle$

**Table 2 Equivalent SQL Query for the  $g$  operator**

<b>SELECT</b>	$\gamma_G T.X, agg(T)$
<b>FROM</b>	$T$
<b>GROUP BY</b>	$\gamma_G T.X$

In Example 1, the measure Count is computed by directly aggregating the fact table; it can be expressed as follows:

$$S_C = g_{(t:hour,U:IP),count(*)} D \quad (3.2.1)$$

In defining the aggregation operator, we assume that the value mapping function  $\gamma_{D_i}(T.X_i)$  to be an atomic function. In real world, dimension hierarchy information might be stored in dimension tables. In order to perform a value mapping between two different domains, we then need to lookup the dimension table.

However, since a dimension table is typically much smaller than the fact table and can usually be stored in memory for efficient lookup, it is reasonable to treat them as inexpensive functions.

3. **Selection operator.** If an expression  $T \in \mathcal{A}$ , then a selection  $\sigma_{cond}(T) \in \mathcal{A}$ . Examples 2 and 3 can be expressed as follows:

$$S_S = g_{(t:hour),count(*)}(\sigma_{M>5} S_C) \quad (3.2.2)$$

$$S_T = g_{(t:hour),count(S_C.M)}(\sigma_{M>5} S_C) \quad (3.2.3)$$

Notice that the expression  $S_C$  is defined in (3.2.1), which is used here as the source of the composition.

4. **Match join** A composite measure is typically generated based on the measures of other, related, regions. If the related regions are descendants of the given region, we can use the aggregation operator. Otherwise, we need a new operator, *match join*.

We can join two algebra expressions  $S, T$  and then group the results by the dimension attributes of  $S$ . We use the notation  $S \bowtie_{cond,agg} T$  to indicate the match join operation, which is equivalent to the SQL query in Table 3.

**Table 3 Equivalent SQL Query for the  $\bowtie$  operator**

<b>SELECT</b>	$S.X, agg(T)$
<b>FROM</b>	$S$
<b>LEFT OUTER JOIN</b>	$T$ ON $cond(S, T) = true$
<b>GROUP BY</b>	$S.X$

LEFT OUTER JOIN is part of SQL-92 standard; when there is no matching tuple in the outer relation for a tuple in the inner, the join will produce a filler tuple with all outer attributes equal to NULL.

Using match join, we can express Example 4 as follows:

$$S_{avg} = S_S \bowtie_{(S_S.t=S'_S.t),avg(S'_S.M)} S'_S \quad (3.2.4)$$

We use  $S'_S$  as an alias of  $S_S$  to avoid confusion in the expression.

The join condition  $cond(S, T)$  matches rows in  $S$  with rows in  $T$ . While the join condition can be arbitrary, in practice some types of join conditions are common, as we discuss later.

5. **Combine Join.** If a collection of expressions  $S, T_1, T_2, \dots, T_n$  satisfies the following conditions:

- (1)  $S, T_1, T_2, \dots, T_n \in \mathcal{A}$
- (2)  $S.G = T_1.G = T_2.G = \dots = T_n.G$
- (3) None of these expressions is  $D$  or  $\sigma(D)$

We can combine their measures and construct a new composite measure. This is done by equi-joining these tables on the dimension attributes. Since none of the  $T_i$  is either  $D$  or a selection over  $D$ , each unique value combination of the dimension attributes will appear at most once in the table. We denote this operation as  $S \bowtie_{fc}(T_1, T_2, \dots, T_n)$ , where  $fc$  is the function used to combined different measures. The equivalent SQL is shown in Table 4.

**Table 4 Equivalent SQL query for the  $\bowtie_{fc}$  operator**

<b>SELECT</b>	$S.X, fc(S, T_1, T_2, \dots, T_n)$
<b>FROM</b>	$S$
<b>LEFT OUTER JOIN</b>	$T_1$ ON $S.X = T_1.X$
.....	
<b>LEFT OUTER JOIN</b>	$T_n$ ON $S.X = T_n.X$

Using combine join, we can express Example 5 as follows:

$$S_{avg} \bowtie_{\text{self}} \text{Savg.M} / (\text{S}_T \text{M} / \text{S}_S \text{M}) (S_T, S_S) \quad (3.2.5)$$

Table 5 shows the summary of the atomic operators in AW-RA. Notice that we deliberately exclude projection  $\pi$  from the algebra, since a table will not represent region measures if we drop either the dimension attributes or the measure attribute.

**Table 5. Summary of operators in AW-RA**

Notation	Prerequisite	Name
$D$		Raw fact table
$\sigma_{cond}(T)$	$T \in \mathcal{A}$	Selection
$g_{G,agg}(T)$	$T \in \mathcal{A}, T.G <_G G$	Aggregation
$S \bowtie_{cond,agg} T$	$S, T \in \mathcal{A}, S \neq D \text{ or } \sigma(D)$	Match join
$S \bowtie_{fc}(T_1, T_2, \dots, T_n)$	$S, T_1, \dots, T_n \in \mathcal{A}$ $S.G = T_1.G = \dots = T_n.G$ $S, T_i \neq D \text{ or } \sigma(D)$	Combine Join

**Theorem 1.** AW-RA has the following properties (proof in [8]):

**Property 1:**  $g_{G_1,agg}(g_{G_2,agg}(T)) = g_{G_1,agg}(T)$  if the aggregation function  $agg$  is distributive.

**Property 2:**  $\sigma_{cond_1}(g_{G,agg}(T)) = g_{G,agg}(\sigma_{cond_2}(T))$ , if the value of  $cond_1$  only depends on the value of dimension attributes, and  $cond_2(X_1, \dots, X_d) = cond_1(\gamma_D X_1, \dots, \gamma_D X_d)$

**Property 3:**  $(S \bowtie T) \bowtie U \neq S \bowtie (T \bowtie U)$

**Property 4:**

$$S \bowtie_{fc}(T_1, \dots, T_k, T_{k+1}, \dots, T_n) \equiv S \bowtie_{fc'}(T_1, \dots, T_{k+1}, T_k, \dots, T_n)$$

where  $fc'$  is derived from  $fc$  and  $fc(v_1, \dots, v_k, v_{k+1}, \dots, v_n) = fc(v_1, \dots, v_{k+1}, v_k, \dots, v_n)$

**Property 5:**

$S \bowtie_{fc}(T_1, T_2, \dots, T_n) = (S \bowtie_{fc_1}(T_1, \dots, T_k)) \bowtie_{fc_2}(T_{k+1}, \dots, T_n)$ , if we can find decomposition functions  $fc_1$  and  $fc_2$  such that:

$$fc(v, v_1, v_2, \dots, v_n) \equiv fc_2(fc_1(v, v_1, \dots, v_k), v_{k+1}, \dots, v_n).$$

With match join  $S \bowtie_{cond,agg} T$ , we can match related regions with different granularities and use the aggregation of their measures as a measure for the given region. In practice, four commonly used join conditions are:

**Self Match:**  $cond_{self} : S.\bar{X} = T.\bar{X}$  The source region is the same as the target region. In such cases, the match join operator is equivalent to a combine join operator.

**Parent/Child:**  $cond_{pc} : \gamma(S.\bar{X}) = T.\bar{X}$  This condition is used when the target region set has finer granularity than the source region set. For each target region, we include the measure of its ancestor region in the input for the aggregation function; i.e., the ancestor is the (only) matching region.

**Child/Parent:**  $cond_{cp} : \gamma(T.\bar{X}) = S.\bar{X}$  This condition is used when  $S$  has coarser granularity than  $T$ . For each target region, we include the measures of its descendants in the source region set as input for aggregation, i.e., all such descendants are matching re-

gions. A match join with  $cond_{cp}$  is essentially equal to an aggregation operator.

**Sibling:**  $cond_{sb} : T.\bar{X} \in NEIGHBOR(S.\bar{X})$  The set  $NEIGHBOR(S.\bar{X})$  denotes a collection of regions that are adjacent to  $S.\bar{X}$  in cube space. The regions in the neighbor set have the same granularity as  $S.G$ . For each target region, we include the measures of “neighbor” regions as input to the aggregation. A common example of sibling matches is the moving windows condition:  $T.X_i \in [S.X_i - low_i, S.X_i + high_i]$ , where  $low_i$  and  $high_i$  give the boundary of attribute  $X_i$  in the neighbor set.

## 4. AGGREGATION WORKFLOWS

While AW-RA is a rigorous way to define region-centric composite subset measures, and more natural than relational algebra, it is still not intuitive for complex queries. In this section, we present a pictorial interface to address this issue.

An **aggregation workflow** is a graph that describes one or more composite subset measures. It contains three types of objects: rectangles that represent region sets; ovals representing measures and the arcs representing the value dependencies.

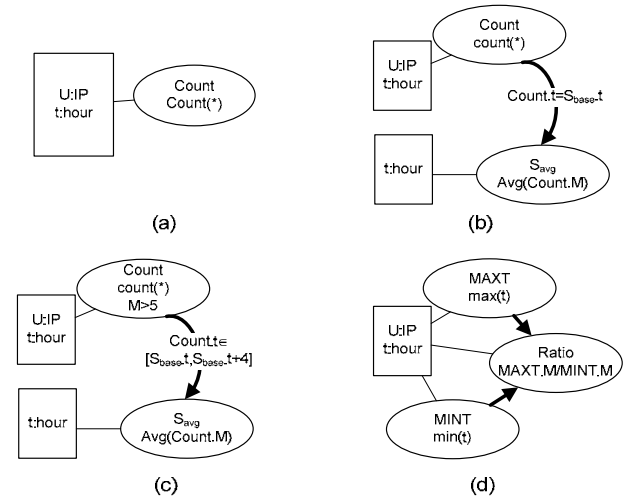
Each *rectangle* in an aggregation workflow represents a region set in cube space. It is associated with one or more ovals, each of which stands for one measure table in AW-RA. The name of the measure is shown inside the oval (with bold type), together with the formula for aggregation and the optional selection condition.

If a measure is computed by applying the aggregation operator  $g$  to fact table  $D$ , there is no computing arc entering that oval; such a measure is called a *basic measure*. For example, the aggregation workflow shown in Figure 3(a) computes a measure Count based on the following algebra expression:

$$\text{Count} = g(\text{U:IP, t:hour}, \text{count}(*))D \quad (4.1)$$

When a measure is computed using values of other measures, the value dependencies are reflected in the aggregation workflow as *computational arcs*. For a given measure oval, if there is one computational arc leading into it, that measure is then computed compositely from other measures via aggregation or match join.

For example, the aggregation workflow in Figure 3(b) is equivalent to:



**Figure 3. Examples of Aggregation Workflows**

lent to the following AW-RA expression:

$$\begin{aligned} \text{Count} &= g(U:IP, t:hour, count(*))D, \quad S_{base} = g(t:Hour, 0)D \\ S_{avg} &= S_{base} \bowtie_{(Count.t=S_{base}.t), avg(Count.M)} Count \end{aligned} \quad (4.2)$$

Since the match condition is a child/parent condition, we can also use the simplified expression:

$$S_{avg} = g(t:hour, avg(Count.M))Count$$

The aggregation workflow in Figure 3(c) is equivalent to the following AW-RA expression:

$$\begin{aligned} \text{Count} &= g(U:IP, t:hour, count(*))D, \quad S_{base} = g(t:Hour, 0)D \\ S_{avg} &= S_{base} \bowtie_{(Count.t \in [S_{base}.t, S_{base}.t+5]), avg(Count.M)} [\sigma_{M>5}(Count)] \end{aligned} \quad (4.3)$$

In the above two examples,  $S_{base}$  is an important auxiliary table. It provides the cells for which the measures need to be computed and appears in every the join condition associated with the arc.

If there are multiple computational arcs leading into the same oval, it means that the corresponding measure is computed via combine join. In this case, all the arcs should come from ovals of the same rectangle, which represent measures for the same region set as that for the target measure. For example, the aggregation workflow in Figure 3(d) is equivalent to the following algebra expression:

$$\begin{aligned} MAXT &= g(U:IP, \max(t))D, \quad MINT = g(U:IP, \min(t))D \\ Ratio &= (g(U:IP, 0)D) \bowtie_{(MAXT.M - MINT.M)} (MINT, MAXT) \end{aligned} \quad (4.4)$$

Generalizing from these examples, we have the following result:

**Theorem 2:** *Each measure in an aggregation workflow can be expressed as an AW-RA expression. (Proof in [8].)*

An aggregation workflow has the following benefits when used to express composite subset measures:

1. It can show the complex internal value dependencies in a composite subset measure query. Just like an ER-diagram summarizes a relational schema, a pictorial diagram captures many complex dependencies in a more intuitive manner.
2. It allows the analyst to include multiple measures within a single workflow diagram.
3. Specific measures are easier to identify since all the measures that are defined on the same region set are attached in the same rectangle. Thus, if aggregation workflows were implemented as a graphical user interface, the user could easily perform various actions such as show/hide all the measures that are associated with the same region set, show/hide measures associated with region sets above/below a certain granularity, and show/hide all the measures that are involved in the computation of a given measure.

## 5. EVALUATION FRAMEWORK

Each composite measure query will typically correspond to multiple correlated aggregation queries. There is considerable opportunity to optimize execution by taking advantage of the connections between them. In this section, we present a systematic evaluation framework that exploits such connections.

### 5.1 The Single-Scan Algorithm

We begin with a single-scan algorithm, following [19]. This method can evaluate all measures, including composite ones, by scanning the raw dataset only once, but it might require massive amounts of memory for large datasets.

The basic idea is to build one hash table for each measure, where each entry in the table corresponds to one region. For a basic measure, the entry contains the value of the measure. For a composite measure as a result from match join or combine join, the entry contains the values of source measures. We scan the raw dataset once and evaluate the values for all the basic measures simultaneously. If the aggregation functions are either distributive or algebraic, we can accomplish this by probing the hash table to update the corresponding entry. After the scan, we have the values for all basic measures, and we can use these values to compute dependent composite measures by topologically ordering the dependent measures so that each is evaluated after all the measures it depends on are finished. Since we don't allow recursive measure definitions, such an order always exists and can be obtained by a topological sort of the aggregation workflow.

### 5.2 Streaming Aggregation Plans

The single-scan algorithm is effective only when the size of memory is big enough to hold all hash tables, which is unlikely to be the case for large datasets, where both the number of hash tables and the number of entries in each hash table can be very large. The idea that we can reduce the memory requirement if we can detect when a hash table entry will no longer be modified is referred to in [19] as "out-of-core processing" and in [2] as "base-tuple completion." But neither paper exploited this idea in any detail. In this subsection, we study the evaluation graph induced from an aggregation workflow and summarize a general evaluation strategy.

Given a collection of AW-RA expressions, we can draw an evaluation graph. Each node in the evaluation graph represents one AW-RA algebra operator. Each arc in the evaluation graph represents a data flow, taking data produced by one operator and feeding it into another operator as input.

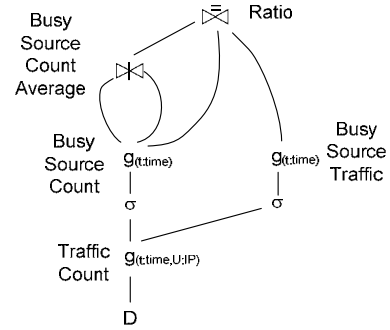


Figure 4. Evaluation Graph for Example 5

The evaluation graph is a directed acyclic graph with a single source. The unique source node represents the fact table  $D$ . All the other nodes are reached from this unique node. Given the large number of operators in the evaluation graph, if we can perform certain operations on  $D$  which will benefit evaluation of the remaining operators, the query performance can be greatly improved.

On the other hand, it is well known that in RA, a selection, join or aggregation operator can be evaluated more efficiently if the input data is sorted appropriately, because the result can be produced before the end of the operation. Furthermore, when the input is ordered, the operator can also produce output possibly ordered by a different key, and the output order can benefit evaluation of subsequent operators. The operators in AW-RA also have these useful

properties. So if we can scan the fact table  $D$  based on a specific order, this order will propagate through the whole evaluation graph and benefit the execution of all the operators.

When there is a data arc from one operator to another, the output order of the first might lead to a large memory footprint for the second. We face the choice of either (1) sticking with the original output order and paying the cost of a large memory footprint, or (2) caching the output of the first operator and re-sorting it. Further, when multiple output streams are combined via match join or combine join, there is the possibility that different input streams might not be synchronized. The asynchrony is mainly caused by the match join operator: if we use a sibling condition in match join, then the resulting output stream might lag behind the input stream. For example, when computing a six hour moving average, the output result will have a six hour delay compared with the input. In the next subsection, we discuss how to handle such asynchrony by recording the “slack” of the stream.

A *streaming aggregation plan* sorts the fact table by a given ordering vector, and then propagates the ordered data into subsequent operators, following the arcs in the evaluation graph. For each arc, its order vector and the synchronization information are computed before the query is evaluation. Based on this information, the total memory footprint can be estimated before a plan is executed.

For a given operator, the input and output orders are related, as explained next. We use order vector, or key,  $\overline{K} = \langle K_1:D_1, \dots, K_m:D_m \rangle$ ,  $K_i \in \{X_1, \dots, X_d\}$  and  $D_i \in \text{Hier}(K_i)$ , to indicate how a data stream is sorted. For example,  $P = \langle t:\text{hour}, D:\text{IP} \rangle$  is an order vector which indicates data records is ordered first by the hour and then by the source IP. An key vector  $\overline{K}_1$  is said to be more general than another key vector  $\overline{K}_2$  if (1)  $\overline{K}_1$  has fewer elements than  $\overline{K}_2$ , (2)  $K_{i1} = K_{i2}$ , and  $D_{i2} <_D D_{i1}$ .

If the input order keys for an operator are  $\overline{K}_1, \overline{K}_2, \dots, \overline{K}_n$ , then  $f_{\text{memory}}(\overline{K}_1, \overline{K}_2, \dots, \overline{K}_n)$  indicates the minimum memory required for computing the measure results. The function  $\overline{K}_{\text{out}}(\overline{K}_1, \overline{K}_2, \dots, \overline{K}_n)$  is the most specific ordering vector the result could have, if the memory usage is bounded by  $f_{\text{memory}}$ . In [8], we show that  $\overline{K}_{\text{out}}$  will always be more general than any of the input vectors  $\overline{K}_i$ . Every aggregation workflow has a streaming evaluation plan:

**Theorem 3:** Given an aggregation workflow and the ordering vector for the raw fact table, there is a corresponding streaming aggregation plan (see [8] for proof).

While the above claim gives us the big picture about how an aggregation workflow can be converted into an executable streaming aggregation plan, here we list several issues to be addressed and the solutions that we provide in the rest of the paper.

1. How do we capture the asynchronization information for each individual data stream? (Section 5.3)
2. How do we compute the output order of an operator, given the order and synchronization of its inputs? (Section 5.3)
3. Given the input stream, what's the evaluation strategy for each individual operator? (Section 5.3)
4. What happens if the memory budget is not big enough even for the optimal sorting order? (Section 5.4)

5. How do we identify the sorting order which will result in minimal memory footprint? (Section 6)

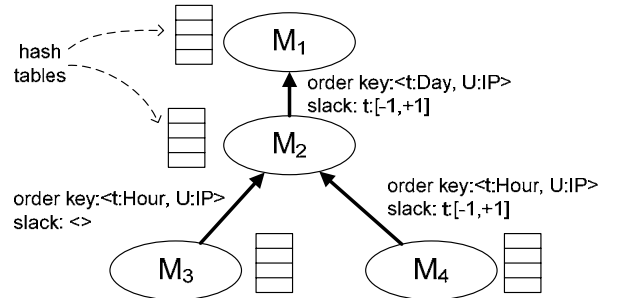
### 5.3 The One-Pass Sort/Scan Algorithm

Based on the streaming aggregation plan, there is a one-pass algorithm that is capable of handling all composite subset measure queries, given enough memory. The evaluation consists of first sorting and scanning the dataset. (The question of how to choose a good sort order will be deferred until Section 6) Processing a record requires us to update hash table entries for all basic measures, as in the single-scan algorithm. In addition, we check for finalized entries, which are those hash entries that will no longer be updated in the rest of the scan, and propagate them to compute dependent measures. The key issues to be addressed are:

1. Determine when a hash table entry is finalized based on the order of the dataset.
2. Propagating such entries to compute dependent measures.
3. Updating the hash-table entry for a dependent measure upon receipt of a finalized entry; we must keep the intermediate information until all finalized entries have been received.

#### 5.3.1 Computation Graph

We begin by constructing an evaluation network based on the streaming aggregation plan. Each node is associated with a hash table that stores the intermediate aggregation result. (If the aggregation function is distributive or algebraic, we only need to keep a constant number of registers to track the intermediate aggregation state.) Each edge is a computational arc indicating the data flow between two operators. Each edge also has two labels, order and slack, which we discuss below. Consider the example in Figure 5.



**Figure 5 Evaluation Network**

Conceptually, each node behaves like a stream data processor. Its input streams are either the scanned data or the output streams of underlying nodes, and the hash table at the node is updated as we process records in the input streams. The node produces a result stream consisting of finalized entries.

Each arc in the computation graph is associated with a stream. For each stream, we have to determine two important properties—the *order* and the *slack*. We label each edge in the evaluation network with its order and slack, and identify finalized entries for each node using these properties for the set of incoming streams.

**Order:** The order property describes how records are arranged in a stream. Based on this information, we can make inferences about the remaining records in the stream. For example, if we know that a stream is ordered by year and we see a record with year equal to 2000, then all the future records will have time value not less than year 2000. In general, the dataset can be sorted by the order vector. For example, if the key is  $\langle t:\text{Day}, T:\text{IP}, U:\text{IP} \rangle$ , the dataset is sorted

**Table 6 Compute the Order and Slack**

<p><b>Input</b>  The region set [ <math>X_1 : D_{r1}, \dots, X_d : D_{rd}</math> ]  The order vectors of input streams  <math>\langle X_1 : D_1^1, X_2 : D_2^1, \dots, X_n : D_n^1 \rangle \dots \langle X_1 : D_1^m, X_2 : D_2^m, \dots, X_n : D_n^m \rangle</math>  The slacks of incoming update streams  <math>\langle (L_1^1, H_1^1), \dots, (L_n^1, H_n^1) \rangle \dots \langle (L_1^m, H_1^m), \dots, (L_n^m, H_n^m) \rangle</math></p> <p><b>Output</b>  The orders of finalized entries <math>\langle X_1 : D_1, \dots, X_d : D_d \rangle</math>  The slack of the finalized entries <math>\langle (L_1, H_1), \dots, (L_d, H_d) \rangle</math></p> <p><b>Algorithm ComputeOrderSlack</b>  for <math>i = 1</math> to <math>n</math>  if <math>(\exists u, v, D_i^u \neq D_i^v)</math>  <math>\forall j \geq i, L_i = H_i = 0</math>, RETURN  else  <math>L_i = \min(L_i^k), H_i = \max(H_i^k)</math>  if <math>(D_i^1 &lt;_D D_{ri})</math>  K.append( <math>X_i : D_{ri}</math> )  <math>L_i = L_i / \text{card}(D_i^1, D_{ri}) - 1</math>  <math>H_i = H_i / \text{card}(D_i^1, D_{ri})</math>  <math>\forall j &gt; i, L_i = H_i = 0</math>, RETURN  else  K.append( <math>X_i : D_i^1</math> )  if <math>(L_i \neq H_i)</math>  <math>\forall j &gt; i, L_i = H_i = 0</math>, RETURN</p>
--

in ascending order of day, with source IP as the first tie breaker and target IP as the second tie breaker. Consider this example:

$$S = g(t:\text{day}, T:\text{IP}, U:\text{IP}, \text{count}(*))D$$

This query computes the number of packets from a source IP to a target IP in a given day. Suppose that the underlying dataset is sorted by key  $\langle t:\text{Month}, T:\text{IP}, U:\text{IP} \rangle$ . We know that all entries in the hash table are finalized whenever the target IP for the current scanned data record changes, since for the current source/target combination, all records scanned subsequently will have a higher value for month (and thus, also for day). The maximum memory footprint is 31, which is the maximum number of days in one month. Further, the finalized hash entries are sorted by the key  $\langle t:\text{Month}, T:\text{IP}, U:\text{IP} \rangle$ .

On the other hand, if the underlying dataset is sorted by key  $\langle t:\text{Hour}, T:\text{IP}, U:\text{IP} \rangle$ , then entries in the hash table are only finalized when the value of the time attribute for the current record switches from one day to the next. The maximum memory required is the number of source IP/target IP combinations for any one day, and the finalized hash entries are sorted by  $\langle t:\text{Day} \rangle$ .

We make an observation about orders that simplifies notation.

**Proposition 2:** All update streams are ordered by an order vector in which the attribute vector is identical to the sort key for the dataset being scanned. Different update streams differ solely in the granularities at which different attributes appear in their sort keys. This allows us to specify every stream order in the form of  $\langle X_1 : D_1, X_2 : D_2, \dots, X_n : D_d \rangle$ , where trailing attributes that don't influence the sort are simply padded out with the domain  $D_{\text{ALL}}$ . (See [8] for proof)

**Slack:** The slack property implies how a specific data stream lags behind or advances over the actual progress of the scan over fact

table. For example, if the stream is sorted by  $\langle t:\text{Year} \rangle$  and the slack is between -1 to 1 year, then we might see records with time value equal to 2000 or 2002 when the scan has progressed to records with time value 2001. The reason why the progress of the data stream might go ahead of the scan over fact table is the existence of sibling match join.

The slack information is important for identifying finalized entries and determining the memory footprint for measures that take multiple inputs. For each attribute, we need to keep track of the highest and lowest slack values. For example, in the following query, suppose we sort the data by key  $\langle t:\text{Day} \rangle$ :

$$S_1 = g(t:\text{Month}, \text{count}(*))D, S_2 = g(t:\text{day}, \text{count}(*))D$$

$$S_{\text{ratio}} = S_2 \bowtie_{\text{cond}_{pc, S_2, M} / S_1, M} S_1$$

The lowest value of slack is -31 and the highest is 0 for the input stream  $S_{\text{ratio}}$ . This is because for a given region in  $[t:\text{Day}]$ , its value depends on the aggregation of the corresponding month, which will only be available at the end of the month. In general, the slack of the stream is caused by computational arcs with parent/child or sibling match conditions.

For a given data stream, its slack can be expressed as the vector of value pairs  $\langle (l_1, h_1), \dots, (l_d, h_d) \rangle$  where  $l_i / h_i$  are the lower/upper bounds of the slack for attribute  $X_i$ .

### 5.3.2 Determining Order and Slack for an Data Stream

We now consider how to identify the order and slack of an update stream for a given composite measure query, given the sort order of the dataset, and the order and slack for all the incoming finalized entry streams.

The problem can be decomposed into two sub-problems. First, for a given measure, if we know the orders and the slacks of all update streams from its source measures, how can we compute the order and the slack of the finalized entries for this measure? Second, knowing the order and the slack of the finalized entry stream for this measure, how can we determine the order and the slack for the corresponding update stream produced by applying the match condition? The latter problem requires separate analyses for different match conditions and the details are provided in [8].

For the former problem, in general, the order of the finalized entries is the common prefix of the sort orders for all the incoming update streams. The slack is computed using the bounding box of slacks for the first attribute that share the same domains but different slacks in the orders of all incoming update streams. Table 6 shows the algorithm to identify the order and the slack of a given composite measure. In this algorithm, the function  $\text{card}(D_1, D_2)$  returns the number of values in domain  $D_1$  that can be mapped into the same value in domain  $D_2$ . For most datasets, this number is not fixed. But the precision of this function will only affect the size estimation, and will not impact the correctness of the evaluation algorithm.

### 5.3.3 The One-Pass Algorithm

We now introduce an algorithm that aims at reducing the memory footprint of the query evaluation. We sort the data before scanning it, and reduce the memory needed to evaluate a measure by flushing hash table entries as soon as we know that they are finalized. The algorithm is as follows (see Table 7):

1. (line 1) Identify all the measures involved in a composite subset measure query. As we described before, each oval in the aggregation workflow represents one measure. Compute the order and the slack of their finalized entries.
2. For each measure, create a hash table. The key is the combination of dimension attributes that appear in the region set of the measure, and the value is the measure itself (or the intermediate values, for composite measures).
3. (line 2) Sort the dataset on the given sort key.
4. (line 3,4,5) Scan the dataset in sort order, and for each data record, repeat Steps 4 to 7:
5. (line 6,7) Use the record to probe the hash tables for the basic measures and update the value of the hash entries using the corresponding aggregation functions.
6. (line 8) Identify “finalized” entries in the hash table.
7. For each computational arc from the current measure, first prepare the update collection from the set of finalized entries by applying the match condition (line 11), then propagate the “update set” to the dependent measures (recursively). Repeat this for all outgoing arcs (line 12). After the update is propagated, the finalized entries are flushed to disk (line 13) and removed from the hash table (line 14).

Most of the steps in Table 7 are quite straightforward. For example, consider the following query:

$$S_1 = g_{(t:Day,U:IP),count(*)}D, S_2 = g_{(t:Day,T:IP),count(*)}D$$

$$S_{max1} = g_{(t:Day),MAX(S_1,M)}S_1, S_{max2} = g_{(t:Day),MAX(S_2,M)}S_2$$

$$S_{max} = S_{max1} \bowtie_{MAX(S_{max1},M,S_{max2},M)} S_{max2}$$

Suppose the dataset is sorted by key  $\langle t:Day, T:IP \rangle$ . Then for measure  $S_{max2}$ , the hash entries are finalized whenever the value of

**Table 7. One-Pass Sort/Scan Algorithm**

	<b>Input</b>
	Dataset $\mathcal{D}$ , sort key $K$
	Measure set $\mathcal{M} = \{M_1, \dots, M_k\}$ ,
	Base measure set $\mathcal{M}_B \subseteq \mathcal{M}$
	<b>Algorithm OnePassEvaluation</b>
(1)	Compute the orders and slacks based on $K$
(2)	Sort $\mathcal{D}$ based the sort key $K$
(3)	Scan dataset $\mathcal{D}$
(4)	For $r \in \mathcal{D}$ and $M \in \mathcal{M}_B$
(5)	EvalMeasure( $\{r\}, M$ )
	Flush the hash tables of all measures
	<b>Algorithm EvalMeasure(updateSet, M)</b>
(6)	Scan updateSet, for each $r \in \text{updateSet}$
(7)	Use $r$ to update Hashtable( $M$ )
(8)	FinalSet $\leftarrow$ finalized entries in Hashtable( $M$ )
(9)	For each computational arc $A$ from measure $M$
(10)	$M_i$ is the target measure of $A$
(11)	Generate UpdateSet from FinalSet
(12)	EvalMeasure(UpdateSet, $M_i$ )
(13)	Flush the FinalSet to disk
(14)	Remove FinalSet from Hashtable( $M$ )

target IP switches. These finalized entries will be propagated to measure  $S_{max}$ . However, measure  $S_{max}$  also depends on the value of  $S_{max1}$ , which is only available when the day value switches. So the updates from  $S_{max2}$  will be cached in the hash table for measure  $S_{max}$ . When the day values switches, the entries in the hash table for  $S_{max1}$  will also be finalized. When the finalized entries are propagated to measure  $S_{max}$ , they will trigger the finalization of entries in the hash table for measure  $S_{max}$ .

There are two critical steps that require further discussion. One is how to convert finalized entries into updates for the target measure. The other is how to determine whether a hash entry is finalized.

First, we consider how to apply match conditions to a stream of finalized entries to convert it into an update stream. For a self-mapping match condition, there is no need for such a conversion since each finalized entry can by itself be used as the input for the target measure. For a child/parent match condition, we need to find the parent regions for the finalized entries and update the corresponding measure. The parent/child and the sibling match conditions require more sophisticated mechanisms to ensure that the update stream is appropriately ordered; we omit the details due to space constraints.

Next, we consider how to identify the finalized entries. We need to maintain an array of key values, called the *watermark* array. Each entry in the watermark array represents the progress of one input stream. When an update comes from a certain source measure, we convert that update record into a key value, based on the order key schema computed in Table 6. Since the output order of a composite measure is always the prefix of the output orders for its source measures, it is guaranteed that the key value computed from the new updates is always bigger than the value stored in the entry corresponding to that source measure. So we use the new key value to replace the values in the watermark array. If the key value to be replaced was originally the smallest one in the watermark array, then the global watermark for that composite measure has risen. We then check the hash table and identify those entries that are below the global watermark.

*Sketch of the correctness proof.* As the definition of the AW-RA and the aggregation workflow, we proved that a composite subset measure query can be converted into relational algebra with group by extensions. That proof shows us a simple way to evaluate composite subset measure queries by first topologically sorting the

**Table 8. Identify Finalized Entries**

	<b>Input</b>
	The precomputed order key $K = \langle X_1 : D_{r1}, \dots, X_d : D_{rd} \rangle$
	The source measure $M_s$ and the target measure $M$
	The orders of incoming update streams
	$\langle X_1 : D_1^1, X_2 : D_2^1, \dots, X_n : D_n^1 \rangle \dots \langle X_1 : D_1^m, X_2 : D_2^m, \dots, X_n : D_n^m \rangle$
	The watermark array $WM$
	<b>Output</b>
	The collection of finalized entries
	<b>Algorithm FindFinalized(update record r)</b>
	$WM(M_s) = \text{mapKey}(r, K)$
	$W_{min} = \min\{WM\}$
	RETURN $\{v   v \in \text{Hashtable}(M) \wedge \text{mapKey}(v) < W_{min}\}$
	<b>Function mapKey(record r, key schema K)</b>
	RETURN $v, v.x_i = \gamma_{D_i}(r.x_i)$

measures based on their dependencies, and evaluating them in this order. Our algorithm interleaves the evaluation of different measures. The key for the correctness proof is that once an entry is labeled as “finalized,” it will not be updated in future. The correctness proof builds on the following proposition, in addition to Proposition 2, stated earlier:

**Proposition 3:** The algorithm shown in Table 6 guarantees that the all the update streams are sorted in the identified order. Hence that order can be used as the order key for result streams. (see [8] for proof)

Based on Proposition 3, the algorithm in Table 8 will collect the data entries such that the key value computed from those entries is smaller than that for any update stream. This means that these entries will not receive any future updates from any of the input update streams; hence they are finalized.

### 5.3 Multi-Pass Sort/Scan

When the dataset is very large with many dimensions attribute and the query is complex including measures for different region sets, it may not be possible to simultaneously fit all the intermediate results into memory, even if we dynamically flush out finalized hash entries in an optimal manner. In such case, we have to scan in multiple passes. Different sorting order can be used for each pass to maximize the number of measures that can be evaluated. We call each pass a *Sort/Scan (SS) iteration*. It is costly to sort and scan a large dataset, therefore the number of such passes should be minimized.

Since each pass will only produce a subset of measures, then some composite measures might depend on measures that are produced in different passes. In order to compute these composite measures, we need to first materialize each individual dependent measures during the SS iteration and resort to traditional join strategies to

combine them together.

## 6. OPTIMIZATION

The evaluation cost for a composite measure plan is affected by four factors:

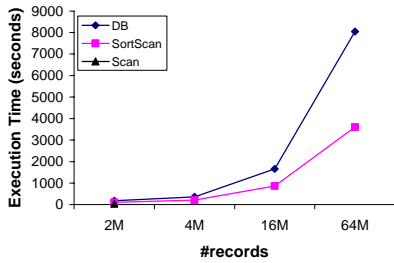
1. The costs of sorting/scanning the raw dataset,  $C_{\text{sort}}$  and  $C_{\text{scan}}$ . In general, we can assume that both costs do not depend on the sort key.
2. The cost of updating in-memory data structures within a pass,  $C_{\text{update}}(\bar{K}, \mathcal{M})$ .  $\bar{K}$  is a sort order of the dataset and  $\mathcal{M}$  is the measure to be evaluated.
3. The cost of writing the values of a measure,  $C_{\text{write}}(\mathcal{M})$ .
4. The cost of evaluating a measure using traditional query evaluation techniques, assuming that all measures it depends on have already been evaluated and stored on disk.  $C_{\text{rel}}(m)$ .

In [8], we present an algorithm to estimate the memory footprint result from an initial sorting order  $\bar{K}$ . Based on that, we can generalize the optimization problem as a general assignment problem [24]. Typically, such problem is NP-hard. But in the case when number of dimensions is small, a brute-force search which tries all possible sorting order combination still can yield result with reasonable cost.

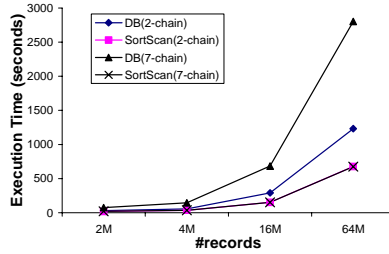
## 7. EXPERIMENTAL RESULTS

In this section, we present a preliminary experimental result of the proposed one pass sort/scan algorithm. For optimization, we used brute force to search all possible sort orders and identify the one with the smallest (estimated) minimal memory footprint.

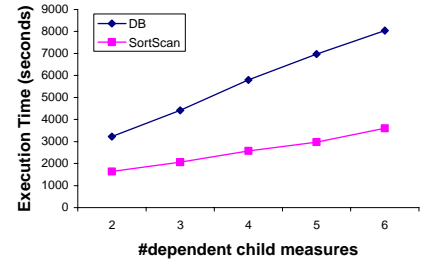
The system is implemented using standard C++ with the STL library. We run all the experiments on a machine with Windows Server 2003, Intel dual-3GHz CPU and 1GB physical memory.



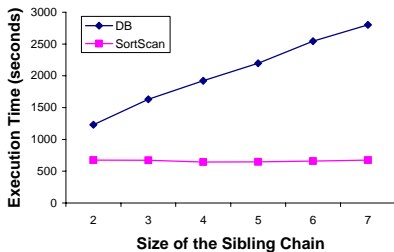
(a) Q1: Result for Child/Parent Match



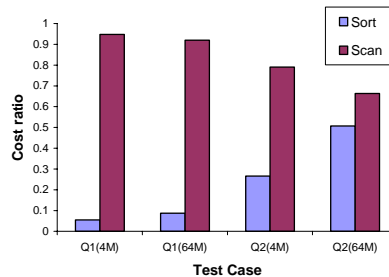
(b) Q2: Result for Sibling Match



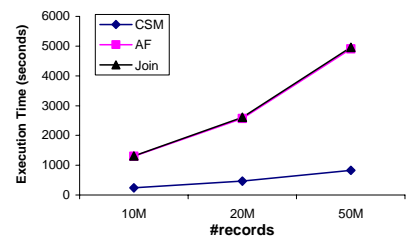
(c) Increasing Number of Measures for Child Region ( $|D|=64M$ )



(d) Increasing Size of Sibling Chains ( $|D|=64M$ )



(e) Cost Breakdown



(f) Performance for Network Data

Figure 6 Experiment Results

## 7.1 Synthetic Dataset

We first used a synthetic dataset to evaluate performance. It contains four dimension attributes that share the same domain hierarchy. For each attribute, there are four domains in the domain hierarchy ( $D_1 <_D D_2 <_D D_3 <_D D_3=D_{ALL}$ ). Any value in any domain will cover 10 distinct values of its sub-domains.

Four datasets were generated, containing 2M (million), 4M, 16M, and 64M records respectively. The values of each attribute were generated independently based on uniform distribution. The datasets were stored in flat files as the input for our algorithm. They were also imported into a commercial relational database system as relational tables for comparison. We configured the RDBMS to reduce logging activity to a minimum. We ran two distinct types of queries and compared the execution time between the sort/scan algorithm, the relational database system, and the single-scan algorithm, which does not sort the dataset before scanning.

The first query contains a measure which is computed by combining seven aggregations for its child regions. The goal is to illustrate that the sort-scan algorithm can exploit the relation between child regions with different granularity and use small size of memory to hold all the intermediate results. For the relational approach, we use the aggregation function `COUNT(DISTINCT(...))` to generate the aggregation for child regions. No explicit join is used. Figure 6(a) shows the case when seven child measures are used. As expected, the single-scan algorithm performs well for small dataset due to the low maintenance cost. However, its performance slows down significantly due to insufficient memory, so we only show the number for 2M dataset. For larger dataset, the sort-scan approach achieves better performance than the relational approach.

The second query contains a measure which is computed through multiple levels (up to seven) of nested sliding windows. In the database system, this is implemented as nested queries with analytical functions. Figure 6(b) shows the results comparison between two approaches; the sort-scan approach performs better than the relational approach. 2-Chain is the case when two level of nesting is used and 7-Chain is the case when seven levels of nesting are used. The sort-scan algorithm outperforms the relational approach for all the cases. More importantly, as we increase the level of nesting, the cost for sort-scan approach almost does not increase since the result is pipeline through the measures without writing any additional intermediate result into disk.

In the next experiment, we fixed the size of the dataset, but increased the number of measures to be simultaneously maintained, to emphasize the benefits of coordination across dependent measures. Figure 6(c) shows the case when we increase the number of child measures from two to six. Figure 6(d) shows the case when we increase the number of sibling chains from two to seven. In both cases, we see that the cost of sort-scan increases at a much slower rate than the relational approach.

Figure 6(e) shows the cost breakdown for two queries in both the small and the dataset. As the graph shows, although the scan step one pass over the raw data table (compared with two for the sort step), it is actually much more expensive than the sort phrase. That effect is more considerable for the first query since that query will use larger memory. That means the in memory operation account for significant part in the evaluation cost and requires further work for optimization.

## 7.2 Honeynet Data Set

To further demonstrate the capabilities of our techniques, we experimented with two analysis queries developed to identify specific types of malicious activity and run over the Lawrence Berkeley Laboratory (LBL) HoneyNet dataset. The target dataset was an 8GB log that was collected at LBL using the monitor described in [21], and transformed into a format described in Section 2.

The first query (network escalation detection) was developed to identify instances where attack packet volume grows significantly from one time period to the next, and contains a measure with several sibling match joins. The intermediate result for this query is quite small. In Figure 7(a), we can see that the sort-scan algorithm does not perform particularly well compared with other methods. The reason is that since the size of intermediate result is so small, the cost of sorting the raw fact table dominates the overall cost. Thus, the simple scan algorithm actually performs the best. This situation can be addressed by switching to simple scan when the required memory is smaller than the memory budget.

The second query (multi-recon detection) was developed to identify instances where attack packets from multiple unique source IP addresses target a specific destination network over a specific period of time. This query contains three measures, each of which based on child/parent match joins. Figure 7(b) shows that the sort-scan algorithm performs significantly faster than the alternative database approach. This effect becomes more pronounced when we combine the two analysis tasks into one query. The result is shown in Figure 6(f). Since the aggregation workflow is capable of expressing multiple measures and evaluating them together, the sort-scan approach, in this case, results in an order of magnitude performance improvement over the relational database query.

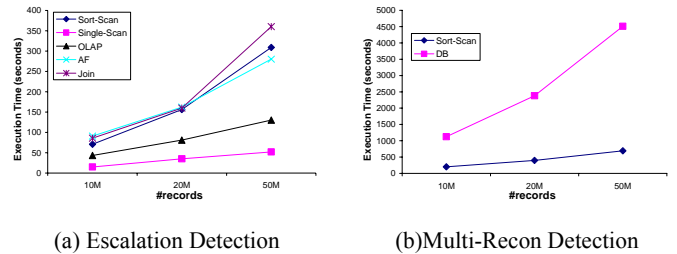


Figure 7. Performance for two queries for real data analysis

## 8. RELATED WORK

Various languages have been proposed for constructing composite measures. The SQL analytical function extension provides partial support via the `PARTITION` and `ORDER` keywords. Our query language provides more flexibility in terms of the form of the measures and can define composite measures with more than two levels. In [26], a new clause is introduced as a SQL extension, which treats the relation as a multidimensional array and defines the composite measures via value assignment. It has been integrated with the Oracle DBMS as the `MODEL` clause. However, in order to combine measures from different region sets, the proposed language still requires nested subqueries. MDX [20] is a query language used in Microsoft Analytical Service. It allows the user to define abstract measures that can be applied to arbitrary regions and to define a composite measure; it also requires nested expression for composite measures.

In [5], a SQL extension is proposed that enables the user to combine multiple aggregations within one GROUP BY query. This language extension was formalized as a relational operator called MD-join in [4]. In [23], this idea is generalized to generate cubes with complex measures.

In [19], the authors expand the idea in [5] and allow the group variables to be matched with the group variables from nested aggregations. This is similar to our idea of using computational arcs to express the computational dependencies between measures. So measures for one group can be used as the input for other related groups. The single scan algorithm is also proposed in this paper. The major difference between our work and the work in [19] is that we impose several carefully chosen restrictions over the computational arcs in a composite measure query, which enables us to use sort/scan evaluation and flush records early. While the work in [19] and the follow-up work in [2, 4] all point out the benefit of flushing finalized entries, they do not provide a detailed mechanism to realize this critical optimization.

The most common evaluation strategies for GROUP BY queries use sorting and hashing [13]. Several papers [6, 15, 27] have studied optimization of aggregation queries in relational databases; they do not consider the multidimensional setting. The CUBE construct was introduced in [14] and several subsequent papers investigated efficient evaluation strategies. Two algorithms proposed in [1] first sort the dataset, then scan the sorted result and evaluate correlated aggregates simultaneously. [16, 25] consider how to choose a cube subspace to materialize, such that the cost of subsequent queries over the cube is minimized. The work in [9] considers optimization of multiple aggregation queries. All these approaches use the implicit dependencies between grouping sets (because of hierarchical region containment) to share common computation. In contrast, the dependencies we seek to exploit are made explicit as computational arcs in the query, and capture relationships other than simple hierarchical containment.

## 9. CONCLUSION

In this paper, we proposed a novel query interface to construct complex measures over a multidimensional dataset. The measure for a given subset is computed by aggregating the data records in that subset, as well as the measures for related subsets in cube space. We proposed an algebra called AW-RA and developed an intuitive pictorial query language.

We also presented an evaluation framework in which multiple related measures are computed in a coordinated fashion. As shown in the empirical study, this strategy can substantially improve query efficiency compared to a traditional relational or OLAP execution engine. We studied optimization in the new framework and suggested some heuristics for finding good evaluation plans.

## 10. ACKNOWLEDGEMENT

This work is supported in part of NSF grant numbers ITR IIS-0326328, IIS-0524671, CNS-0347252, ANI-0335234, and CCR-0325653. Any opinions, findings conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF. We thank Johannes Ullrich and Vern Paxson for permission to use Dshield and LBL datasets respectively.

## 11. REFERENCES

- [1] S. Agarwal, R. Agrawal, et. al., On the Computation of Multi-dimensional Aggregates, in *VLDB '96*, 1996, 506-521.
- [2] M.O. Akinde and M.H. Böhlen, Efficient Computation of Subqueries in Complex OLAP. in *ICDE*, 2003, 163.
- [3] D. Chatziantoniou, Evaluation of Ad Hoc OLAP: In-Place Computation. in *SSDBM*, 1999, 34-43.
- [4] D. Chatziantoniou, M.O. Akinde, et. al. The MD-join: An Operator for Complex OLAP, in *ICDE'01*, 2001, 524-533.
- [5] D. Chatziantoniou and K.A. Ross, Querying Multiple Features of Groups in Relational Databases, in *VLDB '96*, 1996, 295-306.
- [6] S. Chaudhuri and K. Shim, Optimizing Queries with Aggregate Views, in *EDBT*, 1996, 167-182.
- [7] B. Chen, V. Yegneswaran, P. Barford and R. Ramakrishnan: Toward a Query Language for Network Attack Data. *ICDE NetDB Workshops 2006*: 28
- [8] L. Chen, R. Ramakrishnan et. al. Composite Subset Measures, Technical Report 1557, University of Wisconsin - Madison. <http://www.cs.wisc.edu/techreports/>
- [9] Z. Chen and V. Narasayya, Efficient computation of multiple group by queries, in *SIGMOD '05*, 2005, 263-274.
- [10] J. Dean and S. Ghemawat, MapReduce: Simplified Data Processing on Large Clusters, In *OSDI'04*, 2004
- [11] DShield Project, <http://www.dshield.org>
- [12] S. Ghemawat, H. Gobioff and S. T. Leung, The Google File System, in *SOSP'03*, 2003
- [13] G. Graefe, Query evaluation techniques for large databases, *ACM Comput. Surv.*, vol. 25, pp. 73-169, 1993.
- [14] J. Gray, S. Chaudhuri, et. al., Data Cube: A Relational Aggregation Operator Generalizing Group-by, Cross-Tab, and Sub Totals. *Data Min. Knowl. Discov.*, vol. 1, pp. 29-53, 1997.
- [15] A. Gupta, V. Harinarayan, et. al., Aggregate-Query Processing in Data Warehousing Environments, in *VLDB '95*, 1995, 358-369.
- [16] H. Gupta, V. Harinarayan, et. al., Index Selection for OLAP, in *ICDE'97*, 1997, 208-219.
- [17] Hadoop Project, <http://lucene.apache.org/hadoop/>
- [18] Z. Huang, L. Chen, J. Cai, D. S. Gross, D. R. Musicant, R. Ramakrishnan, J. J. Schauer, S. J. Wright: Mass Spectrum Labeling: Theory and Practice. *ICDM 2004*: 122-129
- [19] T. Johnson and D. Chatziantoniou, Extending complex ad-hoc OLAP, in *CIKM*, 1999, 170-179.
- [20] Microsoft MDX Specification <http://msdn.microsoft.com/>
- [21] R. Pang, V. Yegneswaran, et. al. Characteristics of Internet Background Radiation, In *IMC'04*, 2004
- [22] R. Pike, S. Dorward, R. Griesemer and S. Quinlan, Interpreting the Data: Parallel Analysis with Sawzall, *SOSP'03*
- [23] K.A. Ross, D. Srivastava, et. al., Complex Aggregation at Multiple Granularities, in *EDBT '98*, 1998, 263-277.
- [24] D.B. Shmoys, Tardos, An approximation algorithm for the generalized assignment problem, *Math. Program.*, vol. 62
- [25] A. Shukla, P. Deshpande, et. al., Materialized View Selection for Multidimensional Datasets, in *VLDB '98*, 488-499.
- [26] A. Witkowski, S. Bellamkonda, et. al.. Spreadsheets in RDBMS for OLAP, In *SIGMOD '03*, 2003, 52-63.
- [27] W.P. Yan and P.A. Larson, Eager Aggregation and Lazy Aggregation, in *VLDB*, 1995, 345-357