

# User-Defined Aggregate Functions: Bridging Theory and Practice\*

Sara Cohen  
Faculty of Industrial Engineering  
Technion—Israel Institute of Technology  
Technion City, Haifa 32000, Israel  
sarac@ie.technion.ac.il

## ABSTRACT

The ability to create user-defined aggregate functions (UDAs) is rapidly becoming a standard feature in relational database systems. Therefore, problems such as query optimization, query rewriting and view maintenance must take into account queries (or views) with UDAs. There is a wealth of research on these problems for queries with general aggregate functions. Unfortunately, there is a mismatch between the manner in which UDAs are created, and the information that the database system requires in order to apply previous research. The purpose of this paper is to explore this mismatch and to bridge the gap between theory and practice, thereby enabling UDAs to become first-class citizens within the database. Specifically, we consider query optimization, query rewriting and view maintenance for queries with UDAs. For each of these problems we first survey previous results and explore the mismatch between theory and practice. We then present theoretical and practical insights that can be combined to derive a coherent framework for defining UDAs within a database system.

## 1. INTRODUCTION

In the past, the set of aggregate functions available to users of a relational database system was usually limited to the five standard functions: `min`, `max`, `count`, `sum`, and `avg`. However, over time it became apparent that users often want to aggregate data in additional ways. Therefore, all major database systems have added many more built-in aggregate functions to meet this need. In addition, many database systems now allow the user to extend the set of available aggregate functions by defining her own aggregate functions.

User-defined aggregate functions (or UDAs, for short) are useful in two different scenarios. First, the user may simply be interested in aggregating data in a way that is not possi-

ble using the built-in aggregate functions. Popular examples of useful aggregate functions that are not generally built-in are the aggregate functions `prod` (which computes a product of numbers) and `concat` (which concatenates strings). Second, UDAs can be used in conjunction with user-defined types to create more robust and readable code. In this context, one may actually be interested in a standard aggregate function, but want to apply it to a user-defined type. For example, one may define a type which represents complex numbers, and then want to compute sums of complex numbers. This is possible by defining a UDA.

Usually, user-defined aggregate functions can be used wherever a built-in aggregate function can be used, e.g., in queries and views. Therefore, problems such as query optimization, query rewriting and view maintenance must take into account queries (or views) with general UDAs. There is a wealth of research on these problems for queries with the five standard aggregate functions (e.g., [6, 12, 16, 19, 22]) and much of this research has been extended to general aggregate functions (e.g., [2, 5, 9, 10, 17, 20, 25, 26]).

Given that (1) the capability to define UDAs is present in many relational database systems, and (2) past research has considered many problems pertaining to queries with arbitrary aggregate functions, one would expect UDAs to be first-class citizens in the database. In other words, it would be expected for queries (or views) with UDAs to be optimized (or maintained) similarly to queries with built-in aggregate functions. Unfortunately, there is a mismatch between the manner in which UDAs are defined, and the information that the database system requires to apply previous research on general aggregate functions. The purpose of this paper is to (1) extend previous research for optimizing, rewriting and maintaining queries with arbitrary aggregate functions and (2) bridge the gap between theory and practice. This two-pronged approach will enable UDAs to become first-class citizens within the database.

This paper is organized as follows. In Section 2, we give a motivating example. In Section 3 we discuss the manner in which UDAs are currently created within relational database systems. We present a mathematical model of an aggregate function and show its direct correspondence to UDAs in Section 4. In Sections 5 through 7 we consider the problems of query optimization, query rewriting and view maintenance for queries and views with UDAs. For each of these problems, we briefly review previous work and discuss whether such work can currently be applied for UDAs, given the method by which UDAs are defined within a relational

---

\*This work was partially supported by the Israel Science Foundation (Grant 1032/05).

database system. (Note that our discussion is *not* based on insider knowledge of how a particular database system works. Instead, we consider what past research can possibly be applied given the information that the database system has about its UDAs.) We then present additional useful theoretical results as well as practical insights on how to bridge theory and practice, for the problems considered. In Section 8 we extend our discussion to nondeterministic aggregate functions, which have not been studied in the past, and are of practical importance. Finally, Section 9 concludes.

## 2. MOTIVATION

In this section we present a motivating example demonstrating (1) the usefulness of UDAs and (2) the types of problems that arise when querying (or creating views) with UDAs.

Consider a site where users post information about programs available for download. `Pricing(Pid,Type,Price)` and `Programs(Name,Pid,Category,Features,PostDate)` can be used to store information about available programs, where:

- **Pricing** stores for each program the *price*, for different *types* of use (e.g., academic, site license).
- **Programs** stores for each program: a *name*, an *id*, the *category* it belongs to, a list of *features* (stored in an integer field in which each bit is a flag for a feature, i.e., the program has the features corresponding to the bits of value one), and the *date* posted.

Suppose that we have defined three UDAs:

- **cconcat**: when applied to a column of strings it creates a comma-delimited concatenation of its arguments;
- **recent**: when applied to a column of dates it counts the number of dates that are within one week of the current day;
- **bitAnd**: when applied to a column of integers it computes the bitwise-and of its arguments.

We briefly note the abstract characteristics of the UDAs **cconcat**, **recent** and **bitAnd**. (Abstract characteristics of aggregate functions are discussed formally later on.) The aggregate function **cconcat** is nondeterministic, since its result depends on the order in which its arguments are processed. Similarly, **recent** is nondeterministic since its result depends on the day on which it is evaluated. The aggregate function **bitAnd** is deterministic, i.e., it always returns the same value when called with the same multiset. It is also duplicate insensitive since multiple occurrences of its arguments do not affect the result of **bitAnd**.

The view **V1** uses these UDAs to return, for each category, a comma-delimited list of software in the category, along with the number of recently posted programs, an integer denoting the features common to all software in the category and the minimum price to use a program in the category.

```
V1:  SELECT      Category, cconcat(Name) as Names,
                    recent(PostDate) as New,
                    bitAnd(Features) as AllFeatures,
                    min(Price) as MinPrice
    FROM      Programs P1, Pricing P2
   WHERE     P1.Pid = P2.Pid
   GROUP BY  Category
```

By considering the view **V1**, we demonstrate different challenges that arise when dealing with UDAs.

**Query Optimization.** How can **V1** be efficiently evaluated? For example, can the aggregation be pushed below the join (as in [25]) so as to be first computed on the table **Programs**, and the temporary results later combined while joining with **Users** to create the final result? This type of optimization is useful if there are many pricing options for each program, since eager aggregation decreases the cardinality of intermediate results.

**Rewriting Queries with Views.** Suppose that **V1** has been materialized. When can **V1** be used to rewrite other queries? For example, can **V1** be used to compute the following query **Q1**, which finds the set of features in common to all programs?

```
Q1:  SELECT      bitAnd(Features)
    FROM      Programs P1, Pricing P2
   WHERE     P1.Pid = P2.Pid
```

Note that it is possible to use **V1** only if the query optimizer knows how to combine partial results of **bitAnd**.

Now, consider a query **Q1-b** identical to **Q1**, except that it returns the features *not in common* to all programs. **Q1-b** is formulated similarly to **Q1**, except for its **SELECT** clause, which contains **bitNand(Features)**. (The UDA **bitNand** computes the bitwise-nand of its arguments.) Can **V1** be used to compute **Q1-b**? Clearly this is possible only if the query optimizer is aware of the relationship between the UDAs **bitAnd** and **bitNand**, namely that **bitAnd** can be derived from **bitNand** by flipping all the bits in the result.

**View Maintenance.** As before, we assume that **V1** is materialized. A final challenge of interest is how to efficiently maintain **V1** when there are changes to the underlying tables. Actually, **V1** may have to be updated even if there are no changes to the underlying tables, since **V1** uses the nondeterministic UDA **recent**, whose value is dependent on the current date. In general, incremental maintenance (even for queries with only deterministic UDAs) will only be possible if the query optimizer has sufficient information about the UDAs which appear in the query.

## 3. CREATING A UDA

The ability to create a UDA is a relatively new capability available in many commercial database systems. While the technical details and syntax for creating UDAs differ depending on the systems, the essentials remain the same. In this section we survey the basic technique employed to create a UDA. This is important in order to understand the capabilities and limitations of UDAs, as they are currently created.

To create a UDA, the user must implement four functions. (The exact names of the functions, and the class to which they must belong, differ depending on the particular database system.)

1. **Init**: This function is used to initialize any variables needed for the computation later on. Intuitively, it is similar to a constructor.
2. **Terminate**: This function is used to end the calculation and return the final value of the aggregate function. It may involve some calculations on variables which were defined for use with the aggregate function.

3. **Accumulate**: This function is called once for each value aggregated. Generally, this function will “add” the value to the running total computed so far.
4. **Merge**: This function is called to merge intermediate results from two different computations of the aggregate function. This can be used, for example, to enable parallel computation of the aggregate function.

We now present a concrete example of an aggregate function, using a syntax similar to that of SQL Server 2005. Note that in SQL Server 2005, a UDA is defined as a class with the four above-described member functions. The definition of the UDA **Prod**, which computes the product of its arguments, appears below. Observe that this class uses a data member to store internal information about its state. Also, observe that the function **Merge** is called with another instance of the same class as its argument.

```
public class Prod {
    double temp;

    public void Init() {
        temp = 1;
    }
    public void Accumulate(double newVal) {
        temp = temp * newVal;
    }
    public double Terminate() {
        return temp;
    }
    public void Merge(Prod other) {
        temp = temp * other.temp;
    }
}
```

Some database management systems let the user specify additional information about UDAs. For example, in SQL Server 2005, one can specify if a UDA is invariant to nulls, invariant to duplicates or invariant to order. In Oracle, one can specify if a UDA is deterministic and in Informix, the user can specify if a UDA can handle null values. Thus, while the method employed to define a UDA (by implementing the four basic functions) is similar in all major database systems, each system has a different set of properties which can be defined for its UDAs.

## 4. FORMAL FRAMEWORK

In this section we present a formal definition of aggregate functions and discuss their basic properties. Only deterministic aggregate functions will be considered in this section. Nondeterministic aggregate functions will be discussed later in Section 8. The main contribution of this section is in presenting a formalism for aggregate functions that (1) is general enough to represent any aggregate function and (2) has a clear correspondence with the definition of a UDA. Once we have such a formalism, it will be easier to model abstract characteristics of aggregate functions and to formally determine how queries with UDAs may be manipulated.

### 4.1 Aggregate Functions

Let  $D_s$  be a domain and let  $\mathcal{M}(D_s)$  denote the set of all *nonempty* multisets of elements from  $D_s$ . An *aggregate function* is simply any function  $\alpha$  whose domain is  $\mathcal{M}(D_s)$ .

(Note that aggregate functions are never applied to empty sets.) We use  $D_t$  to denote the target of  $\alpha$ . In some cases, the target of  $\alpha$  may be the same as the source  $D_s$ , e.g., for **sum**. However, the target of  $\alpha$  may differ from  $D_s$ . For example, **parity** can be defined over the domain  $\mathcal{M}(\mathbb{N})$  (where  $\mathbb{N}$  is the set of natural numbers) and return a value in the target  $D_t = \{\text{even}, \text{odd}\}$ .

The definition of an aggregate function above clearly encompasses all possible aggregate functions. However, there is no direct correspondence between this definition and the manner in which UDAs are created. We must capture this correspondence in order to be able to formally reason about UDAs. For this purpose we define *well-formed* aggregate functions.

**DEFINITION 4.1.** Let  $\alpha$  be an aggregate function over the domain  $\mathcal{M}(D_s)$  with the target  $D_t$ . We say that  $\alpha$  is well-formed if there is a domain  $D_i$  and a triple  $(F, \oplus, T)$  where

- $F : D_s \rightarrow D_i$  is a translating function;
- $\oplus$  is a commutative and associative binary operation over  $D_i$  and
- $T : D_i \rightarrow D_t$  is a terminating function;

such that for all  $\{\{d_1, \dots, d_n\} \in \mathcal{M}(D_s),$

$$\alpha(\{d_1, \dots, d_n\}) = T(F(d_1) \oplus \dots \oplus F(d_n)). \quad (1)$$

We say that  $\alpha$  is defined in terms of  $(F, \oplus, T)$ .

**EXAMPLE 4.2.** The aggregate function **prod** is well-formed. To show this, observe that **prod** can be defined in terms of  $(id, \times, id)$  where  $id$  is the identity function.

The aggregate function **avg** is also well-formed. In this case  $D_i$  is the domain  $\mathbb{Q} \times \mathbb{N}$  (where  $\mathbb{Q}$  is the set of rational numbers). We choose  $F$  as the function that maps a value  $d$  to a pair  $(d, 1)$ . The binary operation  $\oplus$  is defined as  $(d, n) \oplus (d', n') = (d + d', n + n')$ . Finally, the function  $T$  maps a pair  $(d, n)$  to  $d/n$ . Note that the values in the domain  $D_i$  are used to store the description of intermediate states.

The definition of **avg** can be refined in order to deal with null values, denoted  $\perp$ , in the input. For this case, we choose  $D_i = (\mathbb{Q} \cup \{\perp\}) \times \mathbb{N}$ . We define  $F$  as before. We define  $\oplus$  in the following manner:

$$(d, n) \oplus (d', n') = \begin{cases} (d + d', n + n') & \text{if } d \neq \perp \text{ and } d' \neq \perp \\ (d, n) & \text{if } d' = \perp \\ (d', n') & \text{otherwise} \end{cases}$$

Finally, we define  $T(d, n)$  as  $d/n$  if  $d \neq \perp$  and as  $\perp$  otherwise.  $\square$

We call the domain  $D_i$  in Definition 4.1 the *intermediate domain* of  $\alpha$ . Hereafter, we will assume that there is a value  $d_0 \in D_i$  which is a *neutral element* with respect to  $\oplus$ . Formally, this means that  $d_0 \oplus d = d$  for all  $d \in D_i$ . Note that this assumption is without loss of generality, since we may always extend  $D_i$  with an additional value, if it does not contain a neutral element.

Observe that given a multiset  $M \in \mathcal{M}(D_s)$ , a well-formed aggregate function is computed by successively considering one value after another of  $M$ . The next proposition states that, without loss of generality, we may always assume that all aggregate functions are well-formed. Intuitively, the reason why all aggregate functions are well-formed is that the

intermediate domain  $D_i$  can be used to store all values seen so far, when necessary.

PROPOSITION 4.3. *All aggregate functions are well-formed.*

Hereafter, we will assume that all aggregation functions are well-formed. Thus, we will reason about aggregate functions that are defined in terms of triples  $(F, \oplus, T)$ . In order for our results to be carried over to UDAs, we must show a clear correspondence between this formalism and the way that UDAs are defined. Therefore, we now show how to translate one formalism to the other. Defining a UDA, based on an aggregate function  $\alpha$  is the easier direction, so we start with this translation first.

Let  $\alpha$  be an aggregate function over  $\mathcal{M}(D_s)$ , with target  $D_t$ , defined in terms of  $(F, \oplus, T)$ . Let  $D_i$  be the intermediate domain of  $\alpha$  and let  $d_0$  be a neutral element in  $D_i$ . Consider the following definition of the UDA  $\text{Agg}_\alpha$ .

```
public class Agg $\alpha$  {
     $D_i$  temp;

    public void Init() {
        temp =  $d_0$ ;
    }
    public void Accumulate( $D_s$  d) {
        temp = temp  $\oplus$   $F(d)$ ;
    }
    public double Terminate() {
        return  $T(temp)$ ;
    }
    public void Merge(Agg $\alpha$  other) {
        temp = temp  $\oplus$  other.temp;
    }
}
```

We call  $\text{Agg}_\alpha$  the *UDA version* of  $\alpha$ .

It is not difficult to show the following result.

LEMMA 4.4. *Let  $\alpha$  an aggregate function. Then, the UDA version of  $\alpha$  computes the same function as  $\alpha$ .*

EXAMPLE 4.5. Consider the first definition of **avg** in Example 4.2. (We can reason about the second definition similarly.) We define the UDA version of **avg** by substituting the definitions of  $(F, \oplus, T)$  in the code above. Note that  $D_i = \mathbb{Q} \times \mathbb{N}$ . Therefore, **temp** is a variable that contains a pair of values. Technically this can be accomplished by defining **temp** as an object with two data members or by replacing **temp** with two variables **temp1** and **temp2** which each hold one of the required values. Note also that in this case,  $d_0 = (0, 0)$ , since  $(0, 0)$  is a neutral element with respect to the operation  $\oplus$  of **avg**.  $\square$

We now consider the other direction. Suppose that we are given a UDA **Agg** over the domain  $\mathcal{M}(D_s)$ , defined in the manner described in Section 3. We would like to define an aggregate function  $\alpha_{\text{Agg}}$  in terms of a triple  $(F, \oplus, T)$  so that **Agg** and  $\alpha_{\text{Agg}}$  compute the same function. We will call  $\alpha_{\text{Agg}}$  the *abstract version* of **Agg**.

Clearly, the source and target domains of  $\alpha_{\text{Agg}}$  and **Agg** must be the same. UDAs use variables (data members) to store information about their internal state, while aggregating a multiset of elements. These variables will be represented in our computation of  $\alpha_{\text{Agg}}$  by using the intermediate domain  $D_i$ . Therefore, suppose that **Agg** has variables

$v_1, \dots, v_k$  with domains  $D_1, \dots, D_k$ . We will choose  $D_i$  as  $D_1 \times \dots \times D_k$ . It remains to show how we define  $(F, \oplus, T)$ .

In order to define  $(F, \oplus, T)$ , we will be interested in the state of an object **A** of type **Agg** at different points in time. Formally, the state of **A** at a given point in time is a tuple  $(d_1, \dots, d_k)$  where  $d_i$  is the value of  $v_i$ . Now, let **A** be an object of type **Agg**. We define  $F(d)$ , for  $d \in D_s$  as the state of **A** after first calling **Init** and then calling **Accumulate**(**d**). We define  $(d_1, \dots, d_k) \oplus (d_1^o, \dots, d_k^o)$  as the state of **A** after calling **Merge** with an argument **other** with state  $(d_1^o, \dots, d_k^o)$ , when **A** itself has the state  $(d_1, \dots, d_k)$ . Finally, we define  $T(d_1, \dots, d_k)$  as the value returned by calling **Terminate**, when **A** has the state  $(d_1, \dots, d_k)$ .<sup>1</sup>

LEMMA 4.6. *Let **Agg** be a UDA. Then the abstract version of **Agg** computes the same function as **Agg**.*

EXAMPLE 4.7. Consider the UDA **Prod** defined in Section 3. The abstract version of **Prod** is defined in terms of a triple  $(F, \oplus, T)$  as follows. The function  $F(d)$  is defined as the state of **Prod** after calling **Init** and then **Accumulate**(**d**). By looking at the code, it is easy to see that  $F$  is the identity function, since the state of **Prod** (i.e., the value of **temp**) will be equal to  $d$  after calling **Init** and then **Accumulate**(**d**). Similarly, it is easy to see that  $\oplus$  is the operation  $\times$  and that  $T$  is the identity function.  $\square$

## 4.2 Properties of Aggregate Functions

Aggregate functions differ one from another both in the values that they compute and in their abstract characteristics. For example, some aggregate functions ignore multiple occurrences of the same value (e.g., **max**, **bitAnd**), whereas other aggregate functions are sensitive to such duplications (e.g., **sum**). As another example, for some aggregate functions partially computed values can easily be combined together to return the result of aggregating an entire multiset of values (e.g., **sum**), but for other aggregate functions this may be more difficult, or may not be possible (e.g., **avg**).

The abstract characteristics of an aggregate function  $\alpha$  affect the ability to optimize or manipulate  $\alpha$ -queries, i.e., queries with  $\alpha$  in the **SELECT** clause. Although various characteristics of aggregate functions have been considered in the past, there is no standard terminology to refer to these characteristics. For example, functions definable in terms of structural recursion [20] and decomposable aggregate functions [25] refer to the same characteristic. (We will refer to such aggregate functions as stateless, below.)

In this section we introduce our terminology for various properties of aggregate functions which are defined in terms of a triple  $(F, \oplus, T)$ . In order to carry these properties over to UDAs, we will say that a UDA has a given property (e.g., is duplicate insensitive, stateless, etc.) if the abstract version of the UDA has the given property.

**Duplicate Insensitivity.** Some aggregate functions are not sensitive to duplicate occurrences of values in their arguments. This property is useful for query optimization, e.g., since it allows for additional query plans which do not necessarily preserve the same multiplicities as the original query [2, 5, 25].

Let  $\alpha$  be an aggregation function that is defined in terms of  $(F, \oplus, T)$ . We say that  $\alpha$  is *duplicate insensitive* if the

<sup>1</sup>To simplify our discussion, we assume that **Init** initializes all values and that **Merge** is defined over all possible states.

operation  $\oplus$  is *idempotent*. (Formally this means that  $d \oplus d = d$ , for all elements  $d$  in the domain of  $\oplus$ .) Note that all UDAs that use the **distinct** clause are duplicate insensitive.

EXAMPLE 4.8. The aggregate functions **max** and **bitOr** are duplicate insensitive. To show this observe that they can be defined, respectively, in terms of the triples  $(id, \text{max}_b, id)$  and  $(id, |, id)$ , where  $id$  is the identity function,  $\text{max}_b$  is the binary operation that chooses a maximum and  $|$  is the binary bitwise-or. Both  $\text{max}_b$  and  $|$  are idempotent operations.  $\square$

**Neutral Elements.** There may be elements in the domain of an aggregate function  $\alpha$  which do not have any affect on the computation of  $\alpha$ . Such elements will be formally defined below as *neutral elements*. Neutral elements in the intermediate domain were briefly mentioned above. Although neutral elements for aggregate functions were not generally considered in the past, we show later on that they can be useful to identify cases in which a view is independent of an update.

Let  $\alpha$  be an aggregate function over  $\mathcal{M}(D_s)$ , defined in terms of  $(F, \oplus, T)$ . We say that a value  $d_0 \in D_s$  is a *neutral element* for  $\alpha$  if, for all  $d \in D_s$

$$F(d_0) \oplus F(d) = F(d)$$

An aggregate function can have any number of neutral elements. As a special case, if  $D_s$  contains the null value, often aggregate functions are defined so that  $\perp$  is a neutral element. See, for example, the definition of **avg** from Example 4.2.

EXAMPLE 4.9. The value  $\perp$  is a neutral element for the standard SQL built-in aggregate functions **count**, **sum**, **avg**, **min** and **max**. The value 0 is also a neutral element for **sum**. The aggregate functions **avg** and **parity** do not have non-null neutral elements. Note that **even** is not a neutral element for **parity** since **even** is not in the domain of **parity**. The value **even** is a neutral element in the intermediate domain of **parity**.  $\square$

**Invertible and Stateless Aggregate Functions.** Up until now we have explored special properties of the binary operation  $\oplus$  and how they affect the characteristics of an aggregate function. Now, we consider special characteristics of the terminating function  $T$ .

Let  $\alpha$  be an aggregate function defined in terms of  $(F, \oplus, T)$ . We say that  $\alpha$  is *invertible* if the function  $T$  has an inverse. In the special case that  $T$  is the identity function, we say that  $\alpha$  is *stateless*. Intuitively, if  $T = id$ , then each application of  $\oplus$  incrementally computes the actual value of the aggregate function. In this case, we call  $\alpha$  stateless since it does not store any information besides its current value. If  $T \neq id$ , then  $\oplus$  is used to update information on the current state of the computation. The actual computation itself is realized only when  $T$  is called.

EXAMPLE 4.10. Compare the definition of **avg** from Example 4.2 with the definition of **max** (or **bitOr**) from Example 4.8. By definition, **avg** is not stateless (and is not even invertible), whereas **max** is stateless. Indeed, at no point during the application of the  $\oplus$  operator for **avg** is an actual partial average computed. Only values of the form  $(d, n)$  are computed. On the other hand, when values are aggregated with the  $\text{max}_b$  operator, the result is always the maximum of all values seen thus far.  $\square$

The two properties considered above are important for query rewriting and for view maintenance. [20, 25] each considered a property that coincides with statelessness, although they defined this property a bit differently. (These papers do not model aggregate functions as triples, as we do here.) Recall that we say that a UDA is stateless, if its abstract version is stateless. The following lemma follows from the definition of an abstract version of a UDA.

LEMMA 4.11. *Let **Agg** be a UDA. If **Agg** has a single data member and all that the function **Terminate** does is to return that data member, then **Agg** is stateless.*

This result is very important since it provides us with an easily verified syntactic condition which determines whether a UDA is stateless. For example, one can verify that the UDA **Prod** from Section 3 is stateless.

### 4.3 Associated Aggregate Functions

In this section we consider several aggregate functions that are naturally associated with an aggregate function  $\alpha$ . These associated aggregate functions will be of importance later.

**Accumulating Functions and Merging Functions.** Let  $\alpha$  be an aggregate function over  $\mathcal{M}(D_s)$  defined in terms of  $(F, \oplus, T)$ . Recall that  $\oplus$  is commutative and associative. Therefore, we can extend  $\oplus$  to multisets of values, by simply aggregating them together using the binary operation  $\oplus$ . We associate  $\alpha$  with two aggregate functions:

- **Merge $^\alpha$** : This aggregate function is defined over multisets of elements in the (intermediate) domain  $\mathcal{M}(D_i)$ , and is simply the natural extension of  $\oplus$  to multisets of elements. **Merge $^\alpha$**  can be defined in terms of the triple  $(id, \oplus, id)$ . Intuitively, **Merge $^\alpha$**  is used to merge intermediate results (i.e., elements of the intermediate domain).
- **Acc $^\alpha$** : This aggregate function is defined over  $\mathcal{M}(D_s)$  and has the target domain  $D_i$ . It is defined in terms of the triple  $(F, \oplus, id)$ . Intuitively, **Acc $^\alpha$**  is used to accumulate elements in the domain of  $\alpha$ .

Note that **Merge $^\alpha$**  and **Acc $^\alpha$**  are actually the natural extension of the UDA functions **Merge** and **Accumulate** (which get a single argument) to multisets of arguments.

Sometimes  $\alpha$  coincides with **Merge $^\alpha$**  or with **Acc $^\alpha$** . For example, if  $\alpha$  is stateless, then  $\alpha$  and **Acc $^\alpha$**  coincide, e.g., **max**, **bitOr**. However,  $\alpha$  may not coincide with either **Merge $^\alpha$**  or **Acc $^\alpha$** , e.g., **avg**.

Since **Merge $^\alpha$**  and **Acc $^\alpha$**  are both aggregate functions, one can examine their abstract properties. The distinction between the abstract properties of an aggregate function  $\alpha$ , and those of **Merge $^\alpha$**  and **Acc $^\alpha$** , is critical to correctly manipulate queries using  $\alpha$ . Obviously, both **Merge $^\alpha$**  and **Acc $^\alpha$**  are stateless (regardless of  $\alpha$ ). Also, both aggregate functions are duplicate insensitive if and only if  $\alpha$  is duplicate insensitive. In addition,  $\alpha$  and **Acc $^\alpha$**  have the same neutral elements. However,  $\alpha$  and **Merge $^\alpha$**  may have different neutral elements.

EXAMPLE 4.12. Consider the aggregate function **parity**. Recall that **parity** has no neutral elements. Consider the function **Merge $^{\text{parity}}$** . Its domain is  $\mathcal{M}(\{\text{even}, \text{odd}\})$ . It is easy to see that **even** is a neutral element for **Merge $^{\text{parity}}$** .  $\square$

**Expanding Functions.** Let  $\alpha$  be an aggregate function over  $\mathcal{M}(D_s)$ , defined in terms of  $(F, \oplus, T)$ . We associate with  $\alpha$  an aggregate function  $\alpha_*$  over  $\mathcal{M}(D_s \times \mathbb{N})$ . Observe that the elements in the multisets for  $\alpha_*$  are pairs  $(d, n)$ , where  $d \in D_s$  and  $n \in \mathbb{N}$  is a natural number. Intuitively,  $\alpha_*$  computes, over a multiset  $M$ , the value that  $\alpha$  returns over a multiset  $M'$  that contains  $n$  copies of  $d$ , for each pair  $(d, n) \in M$ . In other words, the number  $n$  is used as a compact representation of the multiplicities of an element.

Formally, let  $M$  be a multiset in  $\mathcal{M}(D \times \mathbb{N})$ . We associate  $M$  with a multiset  $M_*$  in  $\mathcal{M}(D)$  in the following fashion:

$$M_* = \biguplus_{(d,n) \in M} \underbrace{\{d, \dots, d\}}_{n \text{ times}},$$

where  $\uplus$  is bag union. For example,  $\{(a, 2), (b, 1), (a, 1)\}_* = \{a, a, a, b\}$ . The *expanding function*  $\alpha_*$  of  $\alpha$  is the aggregate function over  $\mathcal{M}(D_s \times \mathbb{N})$  such that for all  $M \in \mathcal{M}(D_s \times \mathbb{N})$

$$\alpha_*(M) = \alpha(M_*).$$

**EXAMPLE 4.13.** The expanding function  $\alpha_*$  over  $\mathcal{M}(D_s \times \mathbb{N})$  of an aggregate function  $\alpha$  can always be defined in terms of the triple  $(F, \oplus, T)$  where:  $F(d, n) = \{(d, n)\}_*$  (the bag that contains  $n$  copies of  $d$ ),  $\oplus$  is bag union and  $T = \alpha$ .  $\square$

Defining  $\alpha_*$  in the trivial manner, as in Example 4.13, implies an extremely inefficient way to compute  $\alpha_*$  on a multiset of bags. In particular the values in the intermediate domain will be exponential in the size of the input. (Its size is exponential since  $n$  is written in the argument of  $\alpha_*$  in  $\log n$  digits, whereas every element  $d$  appears  $n$  times in the intermediate domain.)

If  $\alpha$  is duplicate insensitive, then  $\alpha_*$  can be defined in a far more efficient manner (without resorting to duplicating values), as the following proposition states.

**PROPOSITION 4.14.** *Let  $\alpha$  be a duplicate insensitive aggregate function over  $\mathcal{M}(D_s)$ , defined in terms of  $(F, \oplus, T)$ . Then, the expanding function  $\alpha_*$  over  $\mathcal{M}(D_s \times \mathbb{N})$  can be defined in terms of the triple  $(F', \oplus, T)$  where  $F'(d, n) = F(d)$ , for all  $(d, n) \in D_s \times \mathbb{N}$ .*

In [25], a special case of efficiently expressible expanding functions was considered. Let  $\alpha$  be an aggregate function defined in terms of  $(F, \oplus, T)$ . Then  $\alpha$  is a *class C* function if  $\alpha_*$  is definable in terms of  $(F', \oplus, T)$  where  $F'(d, n) = F(d) \times n$ . (This is an adaptation of the definition of [25] to our terminology.) For example, **sum** and **count** are class C functions. Even if  $\alpha$  is not a class C function, there may be efficient ways to define  $\alpha_*$ , as we demonstrate in the following example.

**EXAMPLE 4.15.** Consider the aggregate function **prod** over  $\mathcal{M}(\mathbb{Q})$ . The expanding function **prod** $_*$  of **prod** can be defined in terms of the triple  $(F, \times, id)$  where  $F(d, n) = d^n$ .  $\square$

Note that the domain of an expanding function is always a multiset of pairs in  $\mathcal{M}(D_s \times \mathbb{N})$ . By abuse of notation, when using an expanding function  $\alpha_*$  in an SQL query, we will write it as  $\alpha_*(C1, C2)$  where  $C1$  and  $C2$  are the columns corresponding the elements of  $D_s$  and of  $\mathbb{N}$ , respectively. Note also that every aggregate function has a corresponding expanding function. In particular, we will often be interested in the expanding function of  $\text{Acc}^\alpha$  and  $\text{Merge}^\alpha$ , written respectively as  $\text{Acc}_*^\alpha$  and  $\text{Merge}_*^\alpha$ .

## 5. OPTIMIZING QUERIES WITH UDAS

In this section, we consider the query-optimization problem for queries with UDAs. We start by briefly surveying optimization techniques studied in the past. We show how to generalize such techniques to arbitrary aggregate functions. We then discuss in which cases these techniques can currently be (efficiently) applied to queries with UDAs. Finally, we present insights that will allow us to bridge theory and practice for query optimization.

Previous work considered optimizing query evaluation for aggregate queries. Sometimes very general aggregate queries, e.g., recursive queries, were considered. For example, [17] extended the magic set technique to queries with duplicate semantics and aggregation. This work was further extended in [20], which showed how to efficiently evaluate recursive queries with monotonic aggregation. For such queries, [20] extended previous query evaluation techniques, such as magic sets and semi-naïve evaluation.

In this paper we focus on a nonrecursive aggregate queries. For such queries, optimization techniques usually fall into the broad category of “push-down” or “pull-up” transformations. Intuitively, the idea is to allow aggregation to be performed “early” or “late” (i.e., before or after computing joins between relations). Depending on the particular relations and data, different decisions on when to perform aggregation can yield more efficient query plans.

In [12] an optimization technique for aggregate queries that involved moving predicates through the query tree was presented. These results were rather limited and focused specifically on the standard built-in aggregate functions. Pull-up and push-down transformations for aggregate queries with general aggregate functions were studied in [25]. Three types of query transformations were presented in [25], namely, eager group-by, eager count, and double eager.<sup>2</sup> These same transformations were considered independently in [2]. The transformations of [25] are part of the general framework for query optimization considered in [5].

We present an extended version of the three transformations of [25]. In particular, their version of eager group-by was applicable only to queries with stateless aggregate functions and their version of eager count was applicable only to queries with aggregate functions that are duplicate insensitive or belong to class C. Finally, their version of double eager was applicable only to queries with the properties required for both of the previous transformations. Our transformations, on the other hand, allow for arbitrary aggregate functions in all three transformations. This is achieved due to our particular definition of an aggregate function as a triple and by making use of the aggregate functions  $\text{Acc}^\alpha$ ,  $\text{Merge}^\alpha$ ,  $\text{Acc}_*^\alpha$  and  $\text{Merge}_*^\alpha$ . Formal proofs of correctness of our transformations are not given, due to lack of space. Intuitively, the proofs rely on our definition of an aggregate function  $\alpha$  in terms of a triple  $(F, \oplus, T)$  and can be shown similarly to the correctness proofs in [25].

Consider the tables **T1**(**G1**, **J1**, **A1**, **R1**) and **T2**(**G2**, **J2**, **R2**), and the query **Q2** defined as follows:

```
Q2: SELECT  G1, G2,  $\alpha$ (A1)
      FROM    T1, T2
```

<sup>2</sup>Eager split was also considered by [25]. We do not discuss this transformation since it is a rather straightforward generalization of the other techniques to queries with multiple aggregate functions.

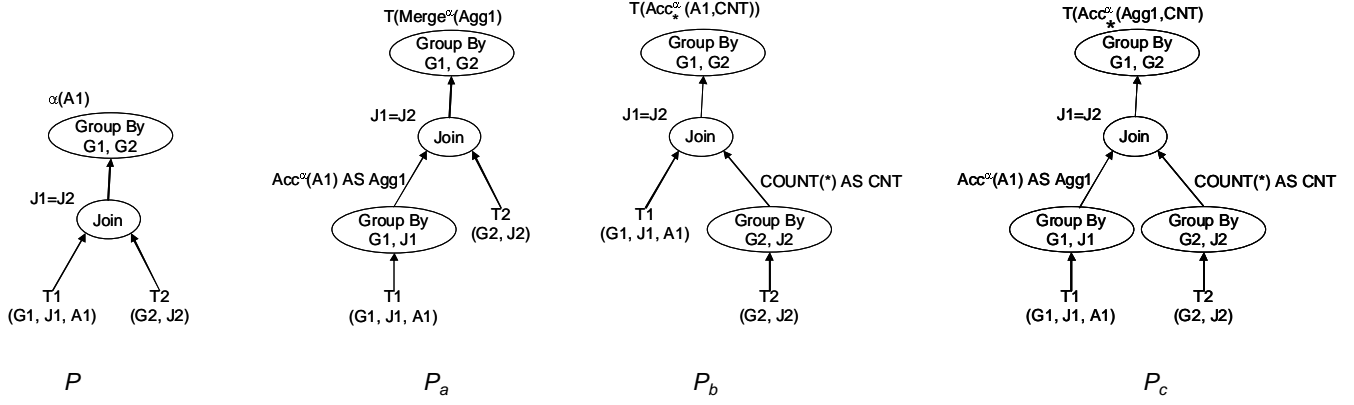


Figure 1: Query plans for Q2, Q2-a (eager group-by), Q2-b (eager count) and Q2-c (double eager).

WHERE J1 = J2  
GROUP BY G1, G2

Observe that G1 and G2 are grouping columns, J1 and J2 are join columns, A1 is the aggregated column and R1 and R2 are the remaining columns in the table (which are not used in the query).

Evaluating this query in a straightforward fashion results in query plan *P* from Figure 1. We now discuss alternative query plans derived by three different transformations. These query plans also appear in Figure 1. This figure is a generalized version of a similar figure from [25].

**Eager Group-By.** An alternative way to evaluate Q2 is to perform the group-by as early as possible. This can be advantageous since the group-by operator generally reduces the number of tuples in its input. In general, the *eager group-by* transformation pushes the group-by below the join operation. If there are several joins, the group-by can be pushed below all, or some, of the joins. An eager group-by of Q2 results in the following query Q2-a, whose query plan *P<sub>a</sub>*, appears in Figure 1.

Q2-a: SELECT G1, G2, T(Merge<sup>α</sup>(Agg1))  
FROM (SELECT G1, J1, Acc<sup>α</sup>(A1) as Agg1  
FROM T1  
GROUP BY G1, J1),  
T2  
WHERE J1 = J2  
GROUP BY G1, G2

Observe that eager group-by involves applying the aggregate function early to some of the tables—in this case to T1. However, several aggregate results may need to be combined together to derive the final result of the query. In order to be able to combine the aggregate results properly, we do not actually apply  $\alpha$  below the join. Instead, we apply  $\text{Acc}^\alpha$  (which is  $\alpha$  without the application of  $T$ ). Therefore, in the result of the eager aggregation of the subquery, we retain the values in the intermediate domain (i.e., the state of the computation). Then, in the outer query we can combine the intermediate results by using  $\text{Merge}^\alpha$ . We derive the final value by applying  $T$  to the results of  $\text{Merge}^\alpha$ .

**Eager Count.** There may be many tuples in T2 that agree with a single tuple of T1 on the join column. Each value in column A1 is aggregated together as many times as there are matching tuples in T2. Hence, the *eager count*

transformation first counts the number of matching tuples, and then uses this count to compute the result of the aggregate function. Similarly to eager group-by, the eager count transformation also reduces the size of the intermediate results. Therefore, eager count sometimes proves a more efficient way to evaluate a query. This transformation uses an expanding function to compute the actual result from the count values. For our example query Q2, performing an eager count results in the following query Q2-b, with query plan *P<sub>b</sub>*, in Figure 1.

Q2-b: SELECT G1, G2, T(Acc<sup>α</sup>(A1, CNT))  
FROM T1,  
(SELECT G2, J2, COUNT(\*) as CNT  
FROM T2  
GROUP BY G2, J2),  
WHERE J1 = J2  
GROUP BY G1, G2

**Double Eager.** The double eager transformation combines the ideas of eager group-by and eager count by both aggregating tuples from T1 and counting tuples from T2 below the join. As before, we use  $\text{Acc}^\alpha$  to perform the eager aggregation. However, in this case we use  $\text{Merge}_*^\alpha$  (instead of  $\text{Merge}^\alpha$ ) to perform the final computation in the outer query. This is necessary in order to take into consideration the count values derived from the eager count. Applying this transformation to Q2 yields the query Q2-c with query plan *P<sub>c</sub>*, in Figure 1.

Q2-c: SELECT G1, G2, T(Merge<sup>α</sup>(A1, CNT))  
FROM (SELECT G1, J1, Acc<sup>α</sup>(A1) as Agg1  
FROM T1  
GROUP BY G1, J1),  
(SELECT G2, J2, COUNT(\*) as CNT  
FROM T2  
GROUP BY G2, J2),  
WHERE J1 = J2  
GROUP BY G1, G2

We now consider the practical implications of the three transformations: eager group-by, eager count, double eager. In particular, we answer the following question: *Can these three transformations be applied efficiently given the current definition of a UDA?*

We start by considering the eager group-by transformation. In order to perform this transformation, the query op-

timizer must be able to apply the aggregate functions  $\text{Acc}^\alpha$  and  $\text{Merge}^\alpha$  for a UDA  $\alpha$ . This is obviously possible, since  $\text{Acc}^\alpha$  and  $\text{Merge}^\alpha$  are computed by multiple applications of the functions **Accumulate** and **Merge**, respectively.

Now, consider the eager-count transformation. In order to perform this transformation, the query optimizer must be able to apply  $\text{Acc}_*^\alpha$ . Currently, the only way that  $\text{Acc}_*^\alpha$  can be applied to a multiset  $\{(d_1, n_1), \dots, (d_m, n_m)\}$  is by calling the function **Accumulate**  $n_i$  times for each  $d_i$ . Obviously this is highly inefficient, and it corresponds to a trivial implementation of  $\text{Acc}_*^\alpha$ . Even if the expanding function  $\text{Acc}_*^\alpha$  can be efficiently implemented, there is no way to “tell” this to the query optimizer. We conclude that it is not currently possible to efficiently apply the eager-count transformation.

The double eager transformation combines the ideas underlying eager group-by and eager count. To implement double eager, the aggregate function  $\text{Merge}_*^\alpha$  is required. Once again, there is currently no method to efficiently implement the expanding function  $\text{Merge}_*^\alpha$ . Hence, the only way that this transformation can be applied is by calling the function **Merge** many times (as many times as dictated by the **count** values). This is clearly inefficient.

The discussion above illustrates the theory-practice gap. In theory, we have shown that all three transformations can be applied to queries with arbitrary aggregate functions. In practice, only the first transformation can be efficiently applied. A straightforward way to bridge the gap is to allow the user to implement, for each UDA, the following two functions (in addition to the four standard functions):

- **AccMany(newVal, nTimes)**: Accumulates **newVal** a total of **nTimes** times.
- **MergeMany(other, nTimes)**: Merges **other** with the current value **nTimes** times.

For example, the definition of the UDA **prod** from Section 2 would be extended with the following two functions:

```
public void AccMany(double newVal, int nTimes) {
    temp = temp * exp(newVal, nTimes);
}

public void MergeMany(Prod other, int nTimes) {
    temp = temp * exp(other.temp, nTimes);
}
```

According to Proposition 4.14, if  $\alpha$  is duplicate insensitive, then  $\alpha_*$  can ignore its second argument. In addition, recall that if  $\alpha$  is duplicate insensitive, then both  $\text{Acc}^\alpha$  and  $\text{Merge}^\alpha$  are duplicate insensitive. Therefore, if the query only contains duplicate insensitive aggregate functions, the eager-count and double-eager transformations can be simplified in the following manner: (1) the subquery does not need to compute the **count** value; (2) the outer query does not need to use an expanding function (i.e., eager count can use  $\text{Acc}^\alpha$  instead of  $\text{Acc}_*^\alpha$  and double eager can use  $\text{Merge}^\alpha$  instead of  $\text{Merge}_*^\alpha$ ). Of course, these optimizations can be applied only if the query processor knows if a particular UDA is duplicate insensitive. Therefore, it is very important to allow the user to declare a UDA as duplicate insensitive. Currently, this is not possible in some of the relational database systems.

## 6. REWRITING QUERIES WITH UDAS

In this section we consider the query rewriting problem for aggregate queries. This problem is formalized as follows. Given an aggregate query  $Q$  and views  $V_1, \dots, V_n$ , can  $Q$  be rewritten (i.e., expressed equivalently) using  $V_1, \dots, V_n$ ? In general, we allow  $Q$  to be rewritten using both the base relations and the views. The query rewriting problem is important in many contexts. First, solutions to this problem can be useful for query optimization. If the views are materialized, then computing  $Q$  using the answers to  $V_1, \dots, V_n$  may be more efficient than evaluating  $Q$  from scratch. Second, this question is pivotal in the context of information integration. In this scenario, the relations mentioned in  $Q$  may not be available or may not even exist. The only way to answer  $Q$  is by using the answers to  $V_1, \dots, V_n$ .

The problem of rewriting aggregate queries has been studied from both a theoretical and practical perspective. Theoretical studies identified complete characterizations for determining whether a rewriting exists for certain types of aggregate functions, and special classes of queries (and views). For example, [4] considered rewriting conjunctive and disjunction queries with **count**, **sum** and **max** and [8] considered the same problem for single-relation queries and the aggregate functions **count**, **sum**, **prod**, **avg** and **percent**. [3] considered conjunctive queries with arbitrary aggregate functions. However, [3] does not consider the practical problem of integrating their results within the framework of a query optimizer. In addition, they do not consider computing aggregate functions from other aggregate functions, as we do below.

From a more practical standpoint, it is often highly inefficient to rewrite queries by taking into consideration a complete characterization of the rewriting problem. Therefore, many papers considering the query-rewriting problem, in the context of a real database system, present algorithms based on a syntactic comparison of the body of the query and views. The syntactic comparison is performed in an effort to identify views with bodies that are identical (more exactly, isomorphic) to a part of the query. When such views are found, the part of the query body that is covered by (i.e., matches) the view is replaced by the view. This syntactic approach is sound (i.e., returns rewritings equivalent to the given query) but is not complete (i.e., may miss equivalent rewritings). Rewriting aggregate queries based on syntactic comparisons was considered in [2, 6, 9, 11, 22, 23, 26] and efficient algorithms for the rewriting problem were presented in [13, 18]. These papers differ in the type of aggregate functions that they allow and the class of queries and views considered.

In Section 6.1 we discuss how the syntactic comparisons of the query and view can be applied for rewriting queries, given the current way in which UDAs are defined. As explained above, using such syntactic comparisons have been considered extensively in the past. Our contribution in Section 6.1 is in bridging the theory-practice gap for this method. In Section 6.2 we consider the problem of formally modeling relationships among aggregate functions, in order to rewrite queries with views that may use aggregate functions not appearing in the query. Rewriting queries with *arbitrary aggregate functions* by using views with different aggregate functions was not considered in the past. We present both theoretical results for this problem and a practical suggestion for implementation.



## 6.1 Using Views in a Query

Given an  $\alpha$ -query  $Q$ , the query processor must decide which types of views can be useful in rewriting  $Q$ . An  $\alpha$ -view may be useful to partially evaluate the aggregate function of  $Q$ . Therefore, such views are considered by the query processor as potential candidates for use in a rewriting. Since aggregate functions are generally sensitive to multiplicities, it is important to retain multiplicities when rewriting a query. The only aggregate function that retains multiplicities is `count(*)`. Hence, `count(*)`-views can often be useful to rewrite an aggregate query. To summarize, the query processor may choose to use an  $\alpha$ -view, `count`-views, or both  $\alpha$ - and `count`-views.<sup>3</sup> Intuitively, these three options correspond to the three transformations considered in the previous section—eager group-by, eager count and double eager. In fact, rewriting a query with views is similar to a two step process where: first transformations are applied to the query to derive a query with subqueries, and then the subqueries are replaced by view definitions.

There are several conditions that must hold in order to be able to rewrite a query  $Q$  with a given view, e.g, the columns in the `SELECT` of  $Q$  must not be hidden by the view, aggregation must be a lower granularity, etc. We do not present these conditions in detail since they have been considered extensively in the past, e.g. [3, 9]. Instead, we focus on the problems directly pertaining to the use of UDAs. Particularly, we focus on how the computations of the aggregate values change when part of the query is replaced by a view.

**Using `count(*)`-Views.** When using a `count(*)`-view to rewrite a query  $Q$ , the aggregate value of  $Q$  must be computed while taking into consideration the count values returned by the view. As in Section 5, this is achieved by using an expanding function. Recall the query `Q2-b`. Suppose that we have a view `V3(G2, J2, CNT)`, defined exactly by the subquery of `Q2-b`. Obviously,  $Q$  can be written equivalently as

```
R1:  SELECT  G1, G2, T(Acc $\star$  $\alpha$ (A1, CNT))
      FROM    T1, V3
      WHERE   J1 = J2
      GROUP BY G1, G2
```

This rewriting of  $Q$  uses the expanding function `Acc $\star$  $\alpha$` . Using the same reasoning as before, we conclude that this rewriting can be evaluated (by calling `Acc $\star$  $\alpha$`  many times), but only inefficiently (since no efficient implementation of `Acc $\star$  $\alpha$`  is available). As before, allowing the user to define the function `AccMany` will solve this problem.

**Using  $\alpha$ -Views.** Given an  $\alpha$ -query  $Q$ , we may want to use an  $\alpha$ -view to partially compute values of  $Q$ . As an example, recall the query `Q1` and view `V1` from Section 2. It is not difficult to see that the following query which uses `V1` is equivalent to `Q1`.

```
R2:  SELECT bitAnd(AllFeatures)
      FROM V1
```

In order to achieve such rewritings, the query optimizer must know how to combine together partial results returned by `V1`, in order to create the final value. In the special case of `R2`, this is accomplished by applying `bitAnd` to the results of `V1`. In general, to use  $\alpha$ -views in order to rewrite  $\alpha$ -queries, we must answer the question: *How can partial results of  $\alpha$  be combined, in order to derive the final value of  $\alpha$ ?*

<sup>3</sup>If  $\alpha$  is duplicate insensitive, nonaggregate views may be useful. We do not discuss this further due to lack of space.

At first glance, the answer to our question seems obvious. It would seem that `Merge $\alpha$`  can be used to solve our problem. This is incorrect. We cannot use `Merge $\alpha$`  since its domain is  $D_i$ , whereas the results returned by the view are in the target domain  $D_t$ . To be more exact, the aggregate function `Merge $\alpha$`  is the natural extension of the function `Merge` to multisets of values. `Merge` must receive as an argument another object of the same type as the aggregate function. `Merge` may use the variables of the other object which store its intermediate state for the computation. A materialized  $\alpha$ -view loses its internal state and retains only the final  $\alpha$ -values. These cannot be combined using the function `Merge` (or any other function currently defined in the database).

We come to the unfortunate conclusion that it is not possible for the query processor to use a view which returns  $\alpha$  in order to rewrite a query that returns  $\alpha$ , if  $\alpha$  is a UDA. To overcome this problem, the user must be given the ability to define how a value in the target domain of  $\alpha$  can be converted back to derive a value in the intermediate domain. It is not difficult to see that this is possible if and only if  $\alpha$  is invertible (Section 4). If indeed  $\alpha$  is invertible, the user should be allowed to overload the function `Init` with a version that receives an element in the target domain, and initializes the variables of the class (i.e., populates the variables in the intermediate domain), based on the given value. For example, this function would be defined as follows, for the UDA `Agg $\alpha$`  on Page :

```
public void Init(Dt d) {
    temp = T-1(d);
}
```

Now, for each value returned by the view, an instance of the UDA can be created and initialized by calling this version of `Init`. The instances can be aggregated together using the function `Merge`.

Interestingly, there is an important case where the definition of the method `Init` is obvious. For stateless UDAs (which can easily be recognized, according to Lemma 4.11), the method simply assigns its argument to the single data member.

**EXAMPLE 6.1.** The UDA `bitAnd` can be defined similarly to the UDA `Prod` (appearing in Section 3) with the following adjustments: (1) `temp` should be defined as an integer and initialized as a number with all bits equal to 1, and (2) both `Accumulate` and `Merge` should combine values with `temp` by using `&` (bitwise-and) instead of `*`. Clearly, if `bitAnd` is defined in this manner, then it is stateless. Therefore, the method `Init` should be defined as follows:

```
public void Init(int val) {
    temp = val;
}
```

Now, to compute `Q1` using `V1`, we proceed in the following manner. For each bit-and value  $d$  returned by `V1`, we create an instance `Ad` of the class `bitAnd` and initialize it by calling `Init(d)`. Then, we use `Ad` in the argument of `Merge` to combine the values returned by `V1`, thereby deriving the result of `Q1`.  $\square$

**Using  $\alpha$ - and `count(*)`-Views.** Due to the similarity of this case to the previous cases, we only discuss this case briefly. When using both  $\alpha$ - and `count(*)`-views in a rewriting, we must be able to deal with both problems which arose

when considering each type of view separately. In particular, it is not possible to perform such rewritings in general, due to our inability to merge  $\alpha$ -values returned by a view (as discussed above). In order to perform such rewritings in the general case, we must implement a version of **Init** that gets a value from the target domain. (As before, this type of rewriting is possible only if  $\alpha$  is invertible.) In order to efficiently take advantage of the **count**(\*) values returned, we need an efficient implementation of **Merge**<sub>\*</sub> $^\alpha$ . Once again, this can be achieved by defining the function **MergeMany**.

## 6.2 Computation Rules

In the previous section, we considered the problem of efficiently rewriting an  $\alpha$ -query using  $\alpha$ - and **count**(\*)-views. We showed that this can be done if  $\alpha$  is invertible, and the method **Init** is overloaded. Also, defining **AccMany** and **MergeMany** will speed up the execution. In this section, we extend the set of potential views to be used in a rewriting by considering relationships among aggregate functions (i.e., computing one aggregate function value from another).

Sometimes it is possible to compute an aggregate function over a given column **C** by using values of other aggregate functions defined over **C**. Such relationships among aggregate functions allow for additional methods of rewriting queries. For example, most query optimizers recognize that **avg**(**C**) can be computed from **sum**(**C**) and **count**(**C**). Therefore, an aggregate query involving **avg** may sometimes be rewritten using aggregate views with **sum**(**C**) and **count**(**C**). (In such a case the views containing **sum** and **count** are dealt with similarly to  $\alpha$ -views, as described above.) No previous work has considered the issue of computing arbitrary aggregate functions from other aggregate functions. In this section, we present a systematic approach to state and reason about such computations.

Let  $\alpha_1, \dots, \alpha_n, \beta$  be aggregate functions over the domain  $\mathcal{M}(D_s)$ . We say that  $\beta$  can be computed from  $\alpha_1, \dots, \alpha_n$  if there is a function  $g$  such that for any multiset  $M \in \mathcal{M}(D_s)$

$$\beta(M) = g(\alpha_1(M), \dots, \alpha_n(M)).$$

For example, **avg** can be computed from **sum** and **count** since, for all  $M$ , it holds that  $\text{avg}(M) = \text{sum}(M)/\text{count}(M)$ .

Abstractly, we express relationships among aggregate functions using *computation rules*. Let  $\alpha_1, \dots, \alpha_n$  and  $\beta_1, \dots, \beta_m$  be aggregate functions over the same domain. The *computation rule*

$$\alpha_1, \dots, \alpha_n \rightarrow \beta_1, \dots, \beta_m$$

states that  $\beta_i$  can be computed from  $\alpha_1, \dots, \alpha_n$ , for all  $i \leq m$ . The function  $g$ , for each of the  $\beta_i$ -s, may be different. For example, we have the following computation rules:

$$\text{count}, \text{sum} \rightarrow \text{avg} \quad (\text{r1})$$

$$\text{top2} \rightarrow \text{max} \quad (\text{r2})$$

$$\text{top3} \rightarrow \text{top2} \quad (\text{r3})$$

$$\text{bitNand} \rightarrow \text{bitAnd} \quad (\text{r4})$$

where **topK** returns the  $k$  greatest values. Note that if we can “tell” the query processor Rule (r4), then we can rewrite **Q1-b** using **V1** (Section 2).

It is sometimes possible to deduce that an aggregate function can be computed from another aggregate function by looking at several computation rules. For example, Rules (r2) and (r3) imply  $\text{top3} \rightarrow \text{top2}, \text{max}$ . It may also be possible to

deduce that an aggregate function  $\alpha$  can be computed from  $\alpha_1, \dots, \alpha_k$  by examining the abstract properties of  $\alpha$  and  $\alpha_1, \dots, \alpha_k$ . For example, for any value of  $k$ ,  $\text{topK} \rightarrow \text{max}$ . Similarly, by considering the abstract properties of **count** and **parity**, one may deduce that  $\text{count} \rightarrow \text{parity}$ .

It is not possible to examine the code of a UDA and determine whether it can be computed from other UDAs. However, given a set of computation rules  $R$ , additional computation rules can efficiently be derived. Note that our notation is similar to that used for functional dependencies. The following proposition states that if we are interested in computation rules that are implied by a set of computation rules, then Armstrong’s Axioms is a sound proof system.

**PROPOSITION 6.2.** *If  $\alpha_1, \dots, \alpha_n \rightarrow \beta_1, \dots, \beta_m$  can be derived from a set of computation rules  $R$  using Armstrong’s Axioms, then  $\beta_i$  can be computed from  $\alpha_1, \dots, \alpha_n$ , for all  $i \leq m$ .*

An automatic method to derive computation rules from a set of computation rules is useful to determine views that can be used to rewrite an  $\alpha$ -query, i.e., by considering views that have functions from which  $\alpha$  can be computed.

Proposition 6.2 allows us to build on existing complexity results for functional dependencies to prove results about computation rules. In particular, we are interested in the following questions:

1. *Can  $\beta$  be computed from  $\alpha_1, \dots, \alpha_n$ ?* Answering this question tells us whether a  $\beta$ -query may be rewritten using views with  $\alpha_1, \dots, \alpha_n$ .
2. *What is the smallest set of functions  $\alpha_1, \dots, \alpha_n$  that is enough to compute all of  $\beta_1, \dots, \beta_m$ ?* Answering this question is a first step towards the view-selection problem (i.e., finding the best set of views to materialize given a workload) [21].

By applying results known for functional dependencies [15, 24], we derive the complexity of answering these questions, presented in Lemma 6.3. Obviously, we are only interested in derivations that follow from the computation rules and not from the abstract properties of the aggregate functions.

**LEMMA 6.3.** *Let  $R$  be a set of computation rules.*

- *It is possible to determine whether  $\alpha_1, \dots, \alpha_n \rightarrow \beta$  follows from  $R$  in time  $\mathcal{O}(|R|)$ .*
- *It is possible to find a minimal set of aggregate functions  $\alpha_1, \dots, \alpha_n$  that can be used to compute  $\beta_1, \dots, \beta_m$  in time  $\mathcal{O}(|R|^2)$ .*
- *Determining whether there is a set of at most  $k$  aggregate functions that can be used to compute  $\beta_1, \dots, \beta_m$  is NP-complete.*

Computation rules are useful for finding additional ways to rewrite queries using views. To make such rules practically useful, there must be a method of defining them within a database system. We suggest the following syntax:

```
CREATE AGGREGATION_RULE <name> AS
  <AGG>=<FUNCTION(AGG1, ..., AGGN)>
```

where **AGG**, **AGG1**, **AGGN** are aggregate functions (built-in or UDAs), and **FUNCTION** is built-in or user-defined function. For example:

```
CREATE AGGREGATION_RULE RULE1 AS BitAnd=~(BitNand);
```

## 7. VIEW MAINTENANCE

We briefly discuss the problem of incrementally maintaining materialized views with UDAs. The goal is to incrementally maintain a view after changes to the database by computing exactly which tuples must be inserted into, or removed from, the view. This should be achieved without recomputing the entire view. Incremental maintenance algorithms were presented in [16,19] for the five standard aggregate functions and in [7,10] for general aggregate functions.

Since the algorithm of [10] is defined for general aggregate views, we would like to be able to actually apply it to views with UDAs. In this section, we focus on the theory-practice gap for applying this algorithm. In particular, we focus on how to combine (respectively, remove) inserted (respectively, deleted) values from aggregate columns of a view. If this can be accomplished using the mechanisms provided for UDAs, then algorithms, such as that of [10], can be used for maintaining queries with UDAs.

**Single and Batch Insert.** Let  $V$  be a view with the clause  $\text{SELECT } C_1, \dots, C_K, \alpha(A)$ . Let  $(c_1, \dots, c_k, d)$  be a tuple in  $V$ , where  $c_1, \dots, c_k$  are grouping values and  $d$  is result of computing  $\alpha$  on a multiset  $M$  of values. Suppose that a tuple is inserted into the base relations of  $V$ , thereby causing a single value  $d'$  to be inserted into the multiset  $M$ . We would like to maintain  $V$  by computing the new value that should replace  $d$ .

Intuitively, it seems that **Accumulate** should be used for this task. However, this is not possible since **Accumulate** combines a value in the source domain, with a value in the intermediate domain. Observe that  $d'$  is in the source domain, but  $d$  is in the target domain and not the intermediate domain. Therefore, even very basic maintenance of views with UDAs cannot be performed. As in Section 6, the solution of this problem is to implement (when possible) the overloaded **Init** which translates a value in the target domain to a value in the intermediate domain. After this translation is performed on  $d$ , we can use **Accumulate** to derive the new value for  $V$ .

Using similar reasoning, suppose that we have a batch update that adds values  $d'_1, \dots, d'_m$  into  $M$ . In this case, to update  $V$ , we can do the following: Use **Init** to derive a value in the intermediate domain for  $d$ . Use **Init** to derive a value in the intermediate domain for  $\alpha(\{d'_1, \dots, d'_m\})$ . Use **Merge** to combine these values.

**Single and Batch Delete.** Now, suppose that a deletion was performed on the base relations of  $V$ , thereby causing a single value  $d'$  to be deletion from the multiset  $M$ . Using the currently available mechanisms for UDAs, there is no way to update  $d$  without recomputing  $\alpha$  on  $M \setminus \{d'\}$ . Theoretically, deletion can be performed efficiently if  $\alpha$  is based on a triple  $(F, \oplus, T)$  such that  $\oplus$  has an inverse operation. Therefore, to allow for efficient maintenance with respect to deletion, we suggest to implement (when possible) the following functions:

- **UnAccumulate:** The inverse of **Accumulate**.
- **UnMerge:** The inverse of **Merge**.

These functions can be undefined or throw an exception, when deletion cannot be performed. We can use these functions (together with the overloaded **Init**) for single and batch deletions, respectively, to efficiently recompute the value for  $\alpha$ . (We need **Init** for the same reasons as above.)

**EXAMPLE 7.1.** Consider the UDA **Prod** from Section 3. We define the function **UnAccumulate** as follows. **UnMerge** can be defined similarly.

```
public void UnAccumulate(double oldVal) {
    if (oldVal == 0) then throw Exception;
    temp = temp / oldVal;
}
```

We can use **UnAccumulate** and **UnMerge**, together with our overloaded **Init** to efficiently maintain **Prod**-views with respect to deletion, except when deleting 0.  $\square$

**Views Independent of Updates.** Previous work has considered the problem of determining whether a view is independent of an update, for nonaggregate views, e.g., [1,14]. This work can be extended to aggregate views in a straightforward manner.

Suppose that  $V$  has the clause  $\text{SELECT } C_1, \dots, C_K, \alpha(A)$ . Then,  $V$  is independent of an update if one of the following two conditions holds. (1) Let  $V'$  be the view identical to  $V$ , except  $\alpha(A)$  is replaced by  $A$ . If  $V'$  is independent of an update, then so is  $V$ . (2) If the values that are inserted or deleted into the aggregated column of  $V$  are neutral elements of  $\alpha$ , then  $V$  is independent of the update. (For example, **Prod**-views are independent to insertions or deletions of the value 1 in the aggregated column.)

## 8. NONDETERMINISM

Some of the major database systems allow the user to define nondeterministic aggregate functions. There are several different sources of nondeterminism in UDAs. First, a UDA may call a nondeterministic function such as **random**. Second, the **Accumulate** and **Merge** functions may not be definable in terms of a commutative and associative binary operation. We consider special cases for each of these types of nondeterminism that allow for optimization.

**Calling a Nondeterministic Function.** Suppose that the UDA **Agg** calls a nondeterministic function  $g$ . If  $g$  is guaranteed to return the same value throughout the execution of an **Agg**-query, we say that **Agg** is *pseudo-deterministic*. Observe that a pseudo-deterministic function **Agg** is actually deterministic throughout the runtime of an **Agg**-query. Thus, **Agg**-queries can be optimized using the techniques discussed in Section 5. For example, the scalar function **current\_user** will return the same value throughout the runtime of a given query. Therefore, a UDA that calls **current\_user** can be optimized similarly to a deterministic UDA. (Contrast this to a UDA which calls **random**, which cannot necessarily be optimized while ensuring correctness.)

**Noncommutative Binary Operations.** Consider an aggregate function  $\alpha$  that is definable in terms of a triple  $(F, \oplus, T)$ , but  $\oplus$  is not commutative. Then,  $\alpha$  is nondeterministic, since its result depends on the order of its arguments. Suppose that the user is willing to accept the result of an  $\alpha$ -query as long as it corresponds to *some* ordering of the elements in the multiset. Once again, we can apply the optimization techniques of Section 5 to  $\alpha$ -queries. A prime example of such a UDA is **cconcat**, which can be optimized using the techniques considered in this paper.

## 9. CONCLUSION

We expect our methods to improve the efficiency in which queries with UDAs can be dealt, since (1) they give the

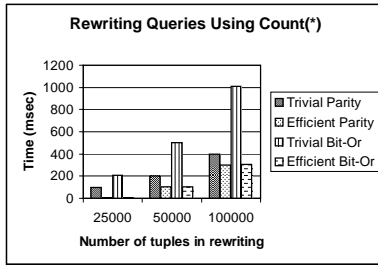


Figure 2: Experimentation

query processor additional ways to optimize, rewrite and maintain such queries and (2) many of our techniques extend previously presented work that has already been shown to be useful. As a small demonstration, Figure 2 presents the runtime of a **bitor**- and **parity**-query in one of the major database systems, when using a materialized **count(\*)**-view. We compare the cases in which expanding functions are trivially or efficiently implemented.

In this paper we make the following contributions.

- We present a formal mathematical model of an aggregate function. We show that this model exactly corresponds to a UDA (as definable in a database system). This model is useful for studying the abstract properties of UDAs, in order to better determine how queries with UDAs can be manipulated.
- We extend previous work on query optimization and query rewriting. In particular, we extend three transformations (eager group-by, eager count, double eager) to queries with arbitrary aggregate functions. We present a formal framework for modeling computations of aggregate functions using other aggregate functions, and show how to reason within this framework.
- For the query optimization, query rewriting and view maintenance problems we show how to bridge the theory-practice gap by defining additional methods for UDAs. We show that defining **AccMany**, **MergeMany** and overloading **Init** is important for all three of these problems. The functions **UnAccumulate** and **UnMerge** are useful for single and batch delete operations.
- We discuss how our work can be extended to non-deterministic aggregate functions, which are important in practice.

As future work, we intend to study cost estimation for queries with UDAs and to perform more extensive experimentation. In addition, we will try to integrate the computation rules of Section 6 directly into a rewriting algorithm.

## 10. REFERENCES

- [1] J. Blakeley, N. Coburn, and P.-A. Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. *ACM TODS*, 14(3):369–400, 1989.
- [2] S. Chaudhuri and K. Shim. Optimizing queries with aggregate views. In *EDBT*, 1996.
- [3] S. Cohen, W. Nutt, and Y. Sagiv. Rewriting queries with arbitrary aggregation functions using views. *ACM TODS*. Accepted for publication.
- [4] S. Cohen, W. Nutt, and A. Serebrenik. Rewriting aggregate queries using views. In *PODS*, 1999.
- [5] C. Galindo-Legaria and M. Joshi. Orthogonal optimization of subqueries and aggregation. In *SIGMOD*, 2001.
- [6] J. Goldstein and P.-A. Larson. Optimizing queries using materialized views: A practical, scalable solution. In *SIGMOD*, 2001.
- [7] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *SIGMOD*, 1995.
- [8] S. Grumbach and L. Tininini. On the content of materialized aggregate views. *JCSS*, 66(1):133–168, 2003.
- [9] A. Gupta, V. Harinarayan, and D. Quass. Aggregate-query processing in data warehousing environments. In *VLDB*, 1995.
- [10] A. Gupta, I. Mumick, and V. Subrahmanian. Maintaining views incrementally. In *SIGMOD*, 1993.
- [11] V. Harinarayan, A. Rajaraman, and J. Ullman. Implementing data cubes efficiently. In *SIGMOD*, 1996.
- [12] A. Levy, I. Mumick, and Y. Sagiv. Query optimization by predicate move-around. In *VLDB*, 1994.
- [13] A. Levy, A. Rajaraman, and J. Ordille. Querying heterogeneous information sources using source description. In *VLDB*, 1996.
- [14] A. Levy and Y. Sagiv. Queries independent of updates. In *VLDB*, pages 171–181, 1993.
- [15] C. Lucchesi and S. Osborn. Candidate keys for relations. *JCSS*, 17(2):270–279, 1978.
- [16] M. Mohania and Y. Kambayashi. Making aggregate views self-maintainable. *DKE*, 32(1):87–109, 2000.
- [17] I. Mumick, H. Pirahesh, and R. Ramakrishnan. The magic of duplicates and aggregates. In *VLDB*, 1990.
- [18] R. Pottinger and A. Halevy. MiniCon: a scalable algorithm for answering queries using views. *The VLDB Journal*, 10(2–3):20–26, 2001.
- [19] D. Quass. Maintenance expressions for views with aggregation. In *Workshop on Materialized Views*, 1996.
- [20] R. Ramakrishnan, K. Ross, D. Srivastava, and S. Sudarshan. Efficient incremental evaluation of queries with aggregation. In *SLP*, 1994.
- [21] K. Ross, D. Srivastava, and S. Sudarshan. Materialized view maintenance and integrity constraint checking: Trading space for time. In *SIGMOD*, 1996.
- [22] D. Srivastava, S. Dar, H. Jagadish, and A. Levy. Answering queries with aggregation using views. In *VLDB*, 1996.
- [23] D. Theodoratos and T. Sellis. Answering multidimensional queries on cubes using other cubes. In *SSDBM*, 2000.
- [24] J. D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume I. Computer Science Press, 1988.
- [25] W. Yan and P.-A. Larson. Eager aggregation and lazy aggregation. In *VLDB*, 1995.
- [26] M. Zaharioudakis, R. Cochrane, G. Lapis, H. Pirahesh, and M. Urata. Answering complex SQL queries using automatic summary tables. In *SIGMOD*, 2000.