, , 1–25 ()

Incremental View Maintenance By Base Relation Tagging in Distributed Databases *

JAMES BAILEY james@dcs.kcl.ac.uk
Department of Computer Science, King's College London, Strand, London WC2R 2LS UK.

GUOZHU DONG dong@cs.mu.oz.au
Department of Computer Science, University of Melbourne, Parkville 3052, Australia.

MUKESH MOHANIA cismkm@cis.unisa.edu.au
School of Computer and Information Science, University of South Australia, Mawson Lakes 5095, Australia.

X. SEAN WANG xywang@isse.gmu.edu
Department of Information and Software Systems Engineering, George Mason University, Fairfax, VA 22030, USA.

Editor:

Abstract. The incremental view maintenance problem deals with the efficient updating of materialized views in response to updates to base relations. This paper considers the problem in a distributed database environment, with communication cost minimization as the primary objective. The views considered are defined based on the relational join operation. The approach is to use "yes"/"no" tags as auxiliary data on tuples in the base relations to indicate whether the tuples participate in joins. These tags will help avoid sending irrelevant data over the network and thus reduce the communication cost. Two basic view maintenance algorithms are proposed using the tags. In addition to reducing communication costs, an important feature of these two basic algorithms is that they derive the "exact change" to views without looking at the old views. This feature allows us to maintain certain aggregates on views without actually materializing the views themselves; this feature is useful in applications such as active databases where many conditions or constraints must be tested whenever updates occur, since a condition is true exactly when some corresponding view has nonzero number of tuples. The paper then combines the use of tags with the counting algorithm to derive a tagged counting algorithm that further reduces the communication cost. The paper illustrates the algorithms by examples and studies their performance via a statistical analysis. The illustrating examples and the performance analysis show that, under uniform distribution with reasonable join participation rates, the use of tags significantly improves the efficiency of view maintenance over similar algorithms without tags. The performance analysis also identifies the situations where a particular algorithm is superior to others. The use of tags for memoing values of subexpressions in a view definition is also explored in the paper.

Keywords: Distributed database, View maintenance, Auxiliary data structure, Base relation tagging, Exact change

* Work of Bailey was performed at University of Melbourne, Australia. Work of Dong was supported in part by the Australian Research Council through research grants, and part of his work was done while visiting George Mason University. Work of Mohania was done at the University of Melbourne and supported by the Australian Research Council through a research grant to Dong. Work of Wang was supported in part by the NSF research initiation award IRI-9409769 and the NSF grant BIR-940483.

1. Introduction

A view in a relational database system is a derived relation from base (or stored) relations specified via a query in, say, SQL or relational algebra. Database systems support views for many reasons, including the enforcement of data security and the provision of conceptual schema suited to users' intuition and needs. In may applications, the view definitions are mostly static, i.e., they do not change in a relatively long period, while the "contents" of the views change from time to time. Materialized views are useful in situations where frequent or rapid access to the data in the views is needed, and recomputation of the entire view from scratch is computationally expensive or even prohibitive. Efficient maintenance for keeping the contents of the views up-to-date is therefore important. In this paper, we study the issue of efficient view maintenance in a distributed database environment where base relations and views may be situated in geographically different locations.

View maintenance has been an active research area for a number of years and many view maintenance strategies have been developed. (See Section 7 for details.) This paper introduces a method that efficiently maintains materialized views in a distributed environment with a small space overhead to the *base* relations (unlike [10, 18, 17, 12] which introduce additional data structures to the views). Roughly speaking, the idea is to let the base relation sites communicate with each other to find out the changes to the views and, at the same time, to use auxiliary data structures on the base relations to minimize such communication. To the best of our knowledge, this paper is the first to advocate such an approach.

We consider distributed views defined by the relational join (followed by relational selection and projection) operation since join is a fundamental relational operation and one of the most difficult relational operations to implement efficiently [14]. We propose two tagged differential incremental algorithms, one for insertion and one for deletion of tuples, for the maintenance of views based on joins. The idea behind these incremental algorithms is to avoid sending irrelevant data by associating tags on tuples in base relations as a screening mechanism. These join-tags indicate whether the associated tuples participate in a join or not; more specifically, a tuple will have the tag value "yes" if it joins with the tuples in the other relation, and will have the tag value "no" otherwise. For example, for the view $R \bowtie S$, we can add, to each tuple in R, a join-tag to indicate whether the tuple joins with any tuple in S. If a tuple with "no" tag is deleted from R, we do not need to do anything since there will not be any change to the view. By adding this extra knowledge to the base relations, which takes very little extra space and can be maintained efficiently, we can reduce the total communication cost significantly.

An extra bonus arising from the above two algorithms is that they produce the *exact change* to the view *without* accessing the view itself. That is, if a deletion of tuple t from the view is produced by the algorithms, then t is guaranteed to exist in the old view and not to exist in the new view, and likewise, if an insertion of tuple t to the view is produced by the algorithms, then t does not exist in the old view and should exist in the new view. This feature is not usually present in other strategies. As we shall see, this exact-change feature helps to maintain views defined in terms of aggregation functions such as `sum` and `count`.

Since our technique gives rise to an efficient method of finding the changes to the view, it can be combined with strategies for centralized view maintenance. For example, we will show that our technique can be combined with the counting method [10] to develop a *tagged counting algorithm*.

In this paper, we also extend our methods to views with more complicated definitions. Furthermore, to avoid recomputing the values of views from scratch, we propose the use of a second type of tag on base relations. These tags are for "remembering" values of certain relational algebra expressions. For example, we can use a tag for tuples in R to remember exactly those tuples that are in $R - S \bowtie T$.

Finally, we give a statistical performance comparison between tagged algorithms and their non-tagged counterparts, between counting algorithm and tagged counting algorithm, and between our tagged algorithm and the counting algorithm. Our results show that our tagged algorithms and the tagged counting algorithms are often significantly better than their tag-free versions. Indeed, association of tags to base relations offers a potentially large saving in terms of communication cost, especially for the case of deletion of tuples from a base relation. The use of tags on base relations is still advantageous, even when insertions and deletions can both possibly occur. We also observe from our analysis that the tagged differential algorithm behaves better than the counting algorithm when join-participation rate and the domain size of atomic values are low, in addition to the exact-change feature discussed earlier.

The outline of the rest of the paper is as follows. Firstly we introduce some notations in Section 2, and our motivating example in Section 3. We describe the tagged differential incremental algorithms in Section 4 and we also improve there the distributed version of the counting algorithm. We next discuss various extensions of these algorithms to more complicated views and memoing tags in Section 5. We then conduct a performance comparison of these algorithms and identify scenarios of high applicability of tagging in Section 6. In Section 7, we discuss related work reported in literature. Finally, we draw conclusions from what has been presented and propose topics for further investigation in Section 8.

2. Definitions

To deal with updates, and old and new instances of relations, we will use the following system of symbols. We will use R, S and T to denote base relation names and their corresponding instances. The symbol δ will denote either an inserted set or a deleted set of tuples, $\triangle$ will denote an inserted set and $\bigtriangledown$ a deleted set. We will use R^o to denote the (<u>o</u>ld) instance of R before an update, and R^n the (<u>n</u>ew) instance of R after the update. We will write $\bar{A}$, $\bar{B}$ etc for sequences of attributes, $\bar{a}$, $\bar{b}$ etc for the corresponding attribute values. We will write t for tuples, and $\langle a, b\rangle$ for tuples when we need to specify the contents of the tuples. By $t[\bar{A}]$ we mean the attribute values of t for the attributes in $\bar{A}$.

Views are defined using relational algebra expressions. Such expressions are recursively built from (base) relation names or constant relations using the two unary operations of σ_ϕ (selection) and $\pi_{\bar{A}}$ (projection) and the three binary operations of $\bowtie$ (join), $-$ (difference), and $\cup$ (union). The condition ϕ in σ_ϕ is called a selection condition, and it is a boolean combination of simple comparisons between attributes and constants. We will use Greek letters such as ϕ, ψ to denote selection conditions.

3. Motivating Example

In this section, we give an example to motivate our tagged differential view maintenance algorithms. The example will be used as a running example throughout the whole paper.

EXAMPLE: To foster economic development, the Commerce Department of the Federal Government may maintain a distributed database as follows. There are a number of sites, including the Melbourne site and the Sydney site. At each site there are two relations, one for the local manufacturers and one for the local potential customers. The manufacturer relation stores information regarding "who is producing what products" and the customer relation stores information regarding "who is interested in buying what major products." (By major products we mean appliances, cars, building materials and so on.)

More specifically, at Melbourne there are these two relations:

Melb_Manufacture	*Comp*	*P*	*Type*

Melb_Customer	*Cust*	*P*	*Type*

And at Sydney there are these two relations:

Syd_Manufacture	*Comp*	*P*	*Type*

Syd_Customer	*Cust*	*P*	*Type*

A tuple $\langle c, p, \tau \rangle$ in the Melb_Manufacture relation means: c is a company located in Melbourne which produces product p of type τ. A tuple $\langle c', p', \tau' \rangle$ in the Syd_Customer relation means: c' is a customer located in Sydney who intends to buy product p' of type τ'.

To help selling products produced in Melbourne to customers of Sydney, the following query is frequently posed:

> "List those pairs of companies in Melbourne and customers in Sydney where the customer may wish to buy products produced by the company."

In relational algebra the query is defined by:

$$V = \pi_{\text{Comp, Cust}}(\text{Melb\_Manufacture} \bowtie_{(\text{P, Type})=(\text{P, Type})} \text{Syd\_Customer}).$$

Because this query is needed frequently, it is profitable to use a corresponding materialized view, called V, at Melbourne. A small sample state of the relevant relations and the view V is shown in Figure 1. □

Part of The Melbourne Database

Melb_Manufacture

Comp	P	Type
c_1	p_1	τ_1
c_1	p_1	τ_2
c_2	p_3	τ_1
c_3	p_2	τ_1
c_3	p_4	τ_1

V

Comp	Cust
c_1	d_1
c_1	d_2
c_3	d_1

Part of the Sydney Database

Syd_Customer

Cust	P	Type
d_1	p_1	τ_1
d_1	p_2	τ_1
d_2	p_1	τ_1
d_2	p_1	τ_2
d_3	p_5	τ_2

Figure 1. A Small Sample State

4. Tagged Incremental Algorithms

In this section we discuss the incremental algorithms for maintaining a view defined by the expression $e = \pi_{\bar{A}\bar{D}}(R \bowtie_{\bar{B}=\bar{C}} S)$. (In section 5 we show how to extend the algorithms to some more complex views.) We assume that $R(\bar{A}\bar{B})$ is at site 1, and $S(\bar{C}\bar{D})$ is at site 2. In this section we first present tagged differential incremental algorithms, and then briefly discuss counting and tagged counting algorithms.

We give two main algorithms: the incremental algorithm for deletions, and the incremental algorithm for insertions of tuples. These algorithms find the "exact change" to the view. This means that we do not need to look at the old view in order to compute the changes to it. Since the change to the view must be sent to the view, we will omit this step in all our algorithms. We assume that the update is made to R, the case when the update is made to S being symmetric.

Along with each algorithm, we will briefly discuss the computation cost at each site and ways to optimize the computation. We will then analyze in detail the communication cost. For this purpose, we assume that $\texttt{deg}^A_{avg} = \frac{|R|}{|\pi_A(R)|}$ (thus $\texttt{deg}^A_{avg}$ is the average degree of each A-value in R), and $\texttt{deg}^C_{avg} = \frac{|S|}{|\pi_C(S)|}$. Furthermore, let $p_B = j_B/|R|$, where $j_B = |\{t \in R \mid t[\bar{B}] \in \pi_{\bar{C}}(S)\}|$. (For each set X, we use $|X|$ to denote its cardinality.) Intuitively, j_B is the number of tuples in R that join with some tuples in S under the operation $\bowtie_{\bar{B}=\bar{C}}$. We call p_B the join participation rate. Clearly $p_B \leq 1$. Whenever $p_B < 1$, we can improve the communication cost of the algorithms by using tags on relations R and S as a screening mechanism. For certain applications where a view is used to help match producers/servers of one site with potential users/customers of another site, join participation rate can be quite low. (In realistic situations, the tables Melb_Manufacture and Syd_Customer as given in Section 3 will have a rather low join participation rate.) In such applications, the base relations are usually very large but not all tuples from the base relations participate in the join because of the site restriction (e.g., some customers at Sydney may not be able to find their desired products from Melbourne).

We use tags on the base relations to indicate whether a tuple participates or not in a join. For the join $R \bowtie_{\bar{B}=\bar{C}} S$, we augment the schemas of R and of S by adding a tag attribute to each. We explain the tags for R, the tags for S being similar. A tuple $\langle \bar{a}, \bar{b} \rangle$ in R will have the tag value "*yes*" if $\bar{b}$ occurs in $\pi_{\bar{C}}(S)$, and the value "*no*" otherwise.

Note that for each relation, we need one tag attribute for each maintained view that involves the relation. In our examples, however, we assume without loss of generality that there is only one view to be maintained and hence there is only one corresponding tag attribute for each relation.

EXAMPLE: The tagged relations for Melb_Manufacture and Syd_Customer are now shown in Figure 2. □

Melb_Manufacture

Comp	P	Type	Tag
c_1	p_1	τ_1	*yes*
c_1	p_1	τ_2	*yes*
c_2	p_3	τ_1	*no*
c_3	p_2	τ_1	*yes*
c_3	p_4	τ_1	*no*

Syd_Customer

Cust	P	Type	Tag
d_1	p_1	τ_1	*yes*
d_1	p_2	τ_1	*yes*
d_2	p_1	τ_1	*yes*
d_2	p_1	τ_2	*yes*
d_3	p_5	τ_2	*no*

Figure 2. Tagged Version of the Small Sample State

We now give the tagged differential incremental algorithms for maintaining the view defined by the join above. Naturally, these algorithms also maintain the tags.

4.1. Tagged differential incremental algorithm for deletion

Suppose a set $\bigtriangledown$ of tuples is deleted from R. The following algorithm derives the change to the view.

Algorithm 1 Tagged differential incremental for deletion

1. At the site of R,

1.1. let $I_1 = \{\langle t, x\rangle \mid t \in \bigtriangledown$ and the tag of t in R^o is *yes*, and
x is defined to be *yes* if $t[\bar{B}] \in \pi_{\bar{B}}(R^n)$, and *no* otherwise$\}$;

1.2. let $I_2 = \{t \in R^n \mid t[\bar{A}] \in \pi_{\bar{A}}(I_1)$ and the tag of t in R^n is *yes*$\}$;

1.3. send I_1 and I_2 to the site of S;

2. At the site of S,

2.1. let $J_1 = \pi_{\bar{A}\bar{D}}(I_1 \bowtie_{\bar{B}=\bar{C}} S)$;

2.2. let $J_2 = \pi_{\bar{A}\bar{D}}(I_2 \bowtie_{\bar{B}=\bar{C}} S)$;

2.3. let $\bigtriangledown_e = J_1 - J_2$; this is the deletion from the view.

2.4. for each $t_S \in S$ and $t_R \in I_1$ where $t_S[\bar{C}] = t_R[\bar{B}]$, let $t_S[\text{ Tag}] = t_R[\text{ Tag}]$.

□

Intuitively, the set I_1 derived at step 1.1 consists of all the deleted tuples that used to join (in R^o) with some tuples in S. Clearly, these are the only tuples that will affect the view. The same step also assigns tags to each such tuple as follows: If the tuple has a common $\overline{B}$ value with some tuples not deleted (in R^n), then it is assigned with "yes"; and otherwise, it is assigned with "no". This tag is used at step 2.4 to update the tags on tuples in S. Now $R^n \bowtie S = R^o \bowtie S - I_1 \bowtie S$ and the new view is $\pi_{\overline{A},\overline{D}}(R^n \bowtie S)$, which is equivalent to $\pi_{\overline{A},\overline{D}}(R^o \bowtie S - I_1 \bowtie S)$. Clearly, the new view may be different from the result of $\pi_{\overline{A},\overline{D}}(R^o \bowtie S) - \pi_{\overline{A},\overline{D}}(I_1 \bowtie S)$. The problem is that there may be some tuple t in $R^o \bowtie S$ such that t is not in $I_1 \bowtie S$ but $t[\overline{AD}]$ is in $\pi_{\overline{AD}}(I_1 \bowtie S)$. (The value $t[\overline{AD}]$ is said to have an "alternative" derivation other than those from the tuples in I_1 and S.) Such a tuple t must have the same $\overline{A}$ value as some tuple in I_1 while different $\overline{B}$ value from each tuple in I_1. Also, the tuple t has to be in $R^o \bowtie S$, hence the tag for t in R^o must be yes. This reasoning gives rise to steps 1.2 and 2.2.

EXAMPLE: Here we illustrate the tagged differential incremental algorithm by considering the deletion of the set $\bigtriangledown = \{\langle c_1, p_1, \tau_1\rangle, \langle c_3, p_4, \tau_1\rangle\}$ from the Melb_Manufacture relation instance of Example 4. The intermediate steps and the new version of relations are shown below. The following steps are performed to maintain the view.

1. Derive I_1 and I_2, and send them to Sydney.

I_1

Comp	P	Type	Tag
c_1	p_1	τ_1	*no*

I_2

Comp	P	Type
c_1	p_1	τ_2

2. At Sydney, join I_1 and I_2 with relation Syd_Customer to yield J_1 and J_2 respectively, and calculate the exact change to the view $\bigtriangledown_V = J_1 - J_2$:

J_1

Comp	Cust
c_1	d_1
c_1	d_2

J_2

Comp	Cust
c_1	d_2
c_3	d_1

$\bigtriangledown V$

Comp	Cust
c_1	d_1

3. For the tag maintenance part, note for example that the value *no* in the tuple $\langle c_1, p_1, \tau_1, no\rangle$ of I_1 means that there are no more tuples in the new Melb_Manufacture with $\langle p_1, \tau_1\rangle$ values. So we change the tags of the tuples, namely $\langle d_1, p_1, \tau_1\rangle$ and $\langle d_2, p_1, \tau_1\rangle$, of Syd_Customer having $\langle p_1, \tau_1\rangle$ values to *no*.

Melb_Manufacture

Comp	P	Type	Tag
c_1	p_1	τ_2	*yes*
c_2	p_3	τ_1	*no*
c_3	p_2	τ_1	*yes*

Syd_Customer

Cust	P	Type	Tag
d_1	p_1	τ_1	*no*
d_2	p_1	τ_1	*no*
d_2	p_1	τ_2	*yes*
d_1	p_2	τ_1	*yes*
d_3	p_5	τ_2	*no*

□

In the above algorithm, the computation at the site of R is to obtain I_1 and I_2. To compute I_1 we need to know, for each deleted tuple t, (1) the tag value of t in the relation R^o, and (2) whether $t[\bar{B}]$ is in $\pi_{\bar{B}}(R^n)$. These can be done by scanning the relation R once for each tuple in $\bigtriangledown$. Hence, the complexity of the task at the site of R is $O(|\bigtriangledown| * |R|)$. When R is large, in order to optimize this process, we may maintain a bit-map index on the tag attribute of R and a B^+-tree index on attributes $\bar{B}$ of R. A bit-map index basically uses one bit to represent each tuple, and 1 indicates the tuple has a tag value "yes" and 0 otherwise. The evaluation for I_2 can also be optimized by using the above bitmap index. The computation at the site of S involves two joins with relation S, hence the complexity of this part of the task is $O(max(|I_1|, |I_2|) * |S|)$. These can also be optimized by a B^+-tree index on attributes $\bar{C}$ of R when S is large.

The communication cost of the above algorithm is as follows:

- when the view is at the site of R or a third site, the cost is $|I_1|+|I_2|+|J_1-J_2| \approx p_B * |\bigtriangledown|(1+\mathtt{deg}^A_{avg}+\mathtt{deg}^C_{avg})$ under the uniform distribution.
- when the view is at the site of S, the cost is $|I_1|+|I_2| \approx p_B * |\bigtriangledown|(1+\mathtt{deg}^A_{avg})$ under the uniform distribution.

4.2. Tagged differential incremental algorithm for insertion

Let a set $\triangle$ of tuples be inserted to R. The following algorithm derives the change to the view.

Algorithm 2 Tagged differential incremental for insertion

1. At the site of R,

   1.1. for tag consistency: for each $t_n \in \triangle$,
   let $t_n[\text{ Tag}] = t_o[\text{ Tag}]$ for each $t_o \in R^o$ where $t_n[\bar{B}] = t_o[\bar{B}]$;
   let $t_n[\text{ Tag}] =$ unknown if t_n still has no tag value;

   1.2. let $I_1 = \{t \in \triangle \mid$ the tag of t in R^n is *yes* or unknown$\}$;

   1.3. let $I_2 = \{t \in R^o \mid t[\bar{A}] \in \pi_{\bar{A}}(I_1)$ and the tag of t in R^n is *yes*$\}$;

   1.4. send I_1 and I_2 to the site of S.

2. At the site of S,

   2.1. let $J_1 = \pi_{\bar{A}\bar{D}}(I_1 \bowtie_{\bar{B}=\bar{C}} S)$;

   2.2. let $J_2 = \pi_{\bar{A}\bar{D}}(I_2 \bowtie_{\bar{B}=\bar{C}} S)$;

   2.3. let $\triangle_e = J_1 - J_2$; this is the insertion to the view.

   2.4. change the tag of each tuple t in S from *no* to *yes* if $t[\bar{C}] \in \pi_{\bar{B}}(I_1)$;

   2.5. send $K = \{t[\bar{B}] \mid t \in I_1, t$'s tag is unknown and $t[\bar{B}] \in \pi_{\bar{C}}(S)\}$ to the site of R.

3. Update the tags of the corresponding tuples in R using the set K: For each tuple t in R whose tag is unknown, change the tag to *yes* if t occurs in K and change the tag to *no* otherwise.

□

The above algorithm works similarly as Algorithm 1. Indeed, I_1 is used to obtain the *possible* insertions to the view, while I_2 is used to obtain tuples that give alternative derivations other than those using the inserted tuples. It is easily seen that the tag is used in constructing both I_1 and I_2; however, for I_1 we also need to deal with new tuples without matching $\bar{B}$ values in R^o.

In the above algorithm, it is easily seen that the computational complexity of the task at the site of R is $O(|\triangle| * |R|)$. When R is large ($\triangle$ is usually small), we may use B^+-tree indices on attributes $\bar{B}$ to optimize step 1.1. For steps 1.2 and 1.3, the bitmap index as discussed earlier for Algorithm 1 can be used. The computational complexity at the site of S is $O(max(|I_1|, |I_2|) * |S|)$. It is easily seen that B^+-tree indices on attributes $\bar{C}$ can be used to optimize the computation of the joins in steps 2.1 and 2.2 when S is large. Also, a bitmap index on the tag attributes of S can help optimize steps 2.4 and 2.5. Note that the tag value "unknown" is not used as a tag value for the tag attributes of R and S, but only for the tuples in $\triangle$ and later I_1.

The communication cost of the above algorithm is as follows:

- when the view is at the site of R or a third site, the cost is $|I_1| + |I_2| + |J_1 - J_2| + |K| \approx p_B * |\triangle|(1 + \texttt{deg}^A_{avg} + \texttt{deg}^C_{avg}) + |\triangle|$ under the uniform distribution.
- when the view is at the site of S, the cost is $|I_1| + |I_2| + |K| \approx p_B * |\triangle|(1 + \texttt{deg}^A_{avg}) + |\triangle|$ under the uniform distribution.

It is also possible to improve the above algorithm if one is willing to sacrifice the exact change feature. This can be done by sending just I_1 to the site of S and sending back J_1 to be compared against the view.

As an aside, we note that under special conditions the communication cost of both of our algorithms can be reduced further. For example, when $\bar{A}$ (and hence the view schema) contains a key of R, then there is no need for I_2 and J_2 in the algorithms; so only I_1 is sent to the site of S.

4.3. Counting and tagged counting algorithms

In [10], Gupta et. al. proposed a counting algorithm for view maintenance in centralized databases by storing the number of alternative derivations for each tuple in the materialized view. That is, a counter is maintained for each tuple in the view to indicate how many ways the tuple can be derived. In this algorithm, the set of tuples in the update is sent to the other relation for join. Finally, the results are sent to the view and the counter value of each view tuple is modified.

We can combine the counting algorithm of [10] for distributed databases by making use of the tags described in section 4. By maintaining a tag value with each tuple in the base relations, we can send only relevant tuples to other sites and reduce the communication cost. Indeed, this algorithm sends those tuples to the other site which are guaranteed to participate in the join. After join, the results are sent to the view site and count of view tuples is modified. Note that we sacrifice the feature of exact change in this algorithm. The performance analysis between distributed version of the counting and tagged counting algorithms is shown in section 6.

5. Other Views and Memoing Tags

In this section we generalize our algorithms to deal with more complicated views. We also introduce the memoing tag for maintaining the values of intermediate expressions in the view definition.

5.1. Aggregation views

In order to explain our incremental view maintenance techniques for views involving aggregation functions, we introduce the following notation. Let r be a relation with attributes $R = R_1 \cup \{A\}$, where $A \notin R_1$. Then $f_{R_1}(r)$, where f is an aggregation function in $\{\texttt{sum}, \texttt{avg}, \texttt{max}, \texttt{min}, \texttt{count}\}$, is a relation on R such that $\langle b_1, \ldots, b_k, a\rangle$ is in $f_{R_1}(r)$ iff $\langle b_1, \ldots, b_k\rangle$ is in $\pi_{R_1}(r)$ and a is the result of applying function f to the set $\pi_A(\sigma_{R_1=\langle b_1,\ldots,b_k\rangle}(r))$. That is, the tuples in r are partitioned into groups such that two tuples are in one group if and only they have the same projection on R_1, and then for each such group, one tuple is obtained with the values under R_1 not changed and the value under A being the aggregation value from all the A values in the group.

Now consider the view $f_{R_1}(E)$, where E is a relational algebra expression. Clearly, if E is materialized and incrementally maintained, then the view $f_{R_1}(E)$ can be incrementally maintained easily. However, we can do better in certain situations. Indeed, if we can derive the exact change to E with each update to the base relations, then we can simply derive the new view for $f_{R_1}(E)$ without materializing E if f is in $\{\texttt{sum}, \texttt{count}, \texttt{max}, \texttt{min}\}$. More specifically, from the exact change derived for E, we know which groups (the partition result mentioned in the previous paragraph) are affected, i.e., tuples inserted or deleted. It is not difficult to see that with these exact changes to E and the previous materialized view for $f_{R_1}(E)$, we can easily update the materialized view. If $f = \texttt{avg}$, then the exact change to E and the previous view is not enough. In this case, we may simply maintain the views $\texttt{sum}_{R_1}(E)$ as well as $\texttt{count}_{R_1}(E)$, for $\texttt{avg}_{R_1}(E)$ can be easily derived from these two views.

As an example, consider the view $\texttt{count}_{\,\mathrm{Comp}}(V)$, where V is as shown in Example 3. This view gives the total number of customers for each company. By using our tagged algorithm, the exact changes to V can be obtained after each update. Hence, the view $\texttt{count}_{\,\mathrm{Comp}}(V)$ can be incrementally updated without materializing the view V.

5.2. *Adding selection and projection*

Given an SPJ view $\pi_{\bar{P}}\sigma_\psi(R \bowtie_{\bar{B}=\bar{C}} S)$, we rewrite ψ as the conjunction $\phi_1 \wedge \phi_2 \wedge \phi_3$, where ϕ_1 contains all the comparisons involving only attributes of R, ϕ_2 contains those involving only attributes of S, and ϕ_0 contains all the other comparisons. (We assume there are no equality comparisons between attributes of R and attributes of S, since they can be specified using the equi-join condition.) Hence, the SPJ view can be written as

$$e = \pi_{\bar{P}}\sigma_{\phi_0}(\sigma_{\phi_1}(R) \bowtie_{\bar{B}=\bar{C}} \sigma_{\phi_2}(S)).$$

For this view, we only need to modify our earlier algorithms by inserting σ_{ϕ_1} as the last operations in the definitions of I_1 and I_2, inserting σ_{ϕ_2} wherever S occurs, and replacing $\pi_{\bar{A}\bar{D}}$ by $\pi_{\bar{P}}\sigma_{\phi_0}$ in the definitions of J_1 and J_2. For example, to change Algorithm 1 for the current view, we should replace I_i by $I'_i = \sigma_{\phi_1}(I_i)$, and J_i by $J'_i = \pi_{\bar{P}}\sigma_{\phi_0}(I'_i \bowtie_{\bar{B}=\bar{C}} \sigma_{\phi_2}(S))$ for $1 \leq i \leq 2$.

5.3. *Generalizing two-way joins*

Earlier we gave our algorithms for two-way joins. We now briefly consider other situations. For three-way joins such as $\pi_{AD}(R \bowtie S \bowtie T)$, where the schemas are $R(AB), S(BC), T(CD)$, we can also generalize our earlier algorithms. For example, we can use the tagged incremental algorithm for two-way joins iteratively: Suppose a deletion $\bigtriangledown_R$ to R is made. We first find, using the tagged incremental algorithm for two-way joins, the set $\bigtriangledown_1$ of tuples that should be deleted from the value of the expression $\pi_{AC}(R \bowtie S)$ in response to this deletion of R. Then, by considering $\pi_{AC}(R \bowtie S)$ as one relation, we find the set of tuples that should be deleted from the value of $\pi_{AD}(R \bowtie S \bowtie T) = \pi_{AD}(\pi_{AC}(R \bowtie S) \bowtie T)$ in response to the deletion of $\bigtriangledown_1$, again using the tagged incremental algorithm for two-way joins.

The maintenance of a general multiple-join view is often complicated, due to the commutativity and associativity of the join operation. There are many join plans (in fact exponential number) to consider. A detailed research of this problem is beyond the scope of this paper.

5.4. *Memoing tags*

It is sometimes profitable to maintain the values of some intermediate expressions in the definition of views by tagging base relations only. Indeed, assume that a view is defined as $(R - (S \bowtie T)) \bowtie T')$. We can have a tag associated with R for memoing the value of the expression $R - (S \bowtie T)$. A tuple t in R is associated with the "*yes*" tag iff the tuple t is actually in the difference $R - (S \bowtie T)$. By doing so, we can use the tagged incremental view update algorithms to maintain the whole view. Furthermore, if $R - (S \bowtie T)$ is the view definition and the view is needed at the site of R, then the view can be maintained by only using the above memoing tags.

It is no difficult to see that we can memo values using tags for the following class of expressions:

- base relations;
- $e_1 - e_2$, provided we can memo value of e_1 using tags;
- $e_1 \cup e_2$, provided we can memo values of both e_1 and of e_2 using tags; and
- $\sigma_\phi e$ and $\pi_{\bar{P}} e$, provided we can memo value of e using tags.
- $\pi_{\bar{X}}(e_1 \bowtie_{\bar{B}=\bar{C}} e_2)$, provided there is an $1 \le i \le 2$ such that we can memo the value of e_i using tags and $\bar{X}$ contains a key of e_i.

Clearly the memo tags will be an efficient way to quickly get, without recomputing, the value of some intermediate expressions. Furthermore, the space overhead is very small: each tuple only becomes one bit longer for such a tag. We now illustrate how to maintain such tags. For example, consider $\pi_{AB}(R) - (S \bowtie T)$. Given an insertion $\triangle$ to R, we use the tag for the value of the expression of $\pi_{AB}(R) - (S \bowtie T)$

to find out the subset N of tuples of $\pi_{AB}(\triangle)$ which are not in the old value of the expression. Then N is sent to the site of S. We do a join of N and S, and only the subset M of tuples in S which participated in this join will be sent to T, together with N. The join of N, M and T will indicate which tuples of N are in the new value of $\pi_{AB}(R) - (S \bowtie T)$. Finally, we update the tag for the value of $\pi_{AB}(R) - (S \bowtie T)$ by changing the tag for the tuples in $\pi_{AB}(\triangle)$ according to the last step.

6. Statistical Performance Comparison

In this section we demonstrate the extra savings in the communication cost because of tagging. We do this by giving statistical comparisons between the cost of tagged algorithms and the tag-free ones. We first compare the cost for the algorithms without the exact-change feature, and we discuss what happens if the exact-change algorithms are compared instead. Furthermore, we examine what happens to the counting algorithm if tagging is used, and how our tagged algorithms compare with the counting algorithms. In the appendix we show how the formulas for communication cost are derived.

To simplify the discussion and notation, we perform the comparisons for the view $V = \pi_{AD}(R \bowtie_{B=C} S)$. Assume $R(AB)$ is located at site 1, and $S(CD)$ is at site 2. We also assume that there are only single tuple insertions and single tuple deletions. Furthermore, tuple values are uniformly distributed over the domain. Communication cost is measured in terms of the number of tuples sent over the network. We regard any projection of a tuple as costing the same to send as the tuple itself. To further simplify the analysis, we assume that all insertions and deletions are made at site 1. However, we will assume that the view may be located at either site 1 or site 2. Using this assumption, we effectively reproduce the effect of insertions and deletions at site 2 as well.

We will mainly compare the cost of the algorithms without emphasizing the exact change feature. To this end we will simplify the tagged insertion algorithm, by removing I_2 and J_2 (and we will use the tagged deletion algorithm as given in Section 4). That is, the simplified version is modified from Algorithm 2 by (a) omitting the steps 1.3 and 2.2, and (b) changing step 2.3 to "let $\triangle_e = J_1$; this is the insertion to the view". For reader's convenience, we present this simplified algorithm as follows:

Algorithm 1 Simplified tagged incremental for insertion

1. At the site of R,
   - for tag consistency: for each $t_n \in \triangle$,
     let $t_n[\text{ Tag}] = t_o[\text{ Tag}]$ for each $t_o \in R$ where $t_n[\bar{B}] = t_o[\bar{B}]$;
     let $t_n[\text{ Tag}] =$ unknown if t_n still has no tag value;
   - let $I_1 = \{t \in \triangle \mid$ the tag of t in R^n is *yes* or unknown$\}$;
   - send I_1 to the site of S.

2. At the site of S,
   - let $J_1 = \pi_{\bar{A}\bar{D}}(I_1 \bowtie_{\bar{B}=\bar{C}} S)$;
   - let $\triangle_e = J_1$; this is the insertion to the view.
   - change the tag of each tuple t in S from *no* to *yes* if $t[\bar{C}] \in \pi_{\bar{B}}(I_1)$;
   - send $K = \{t[\bar{B}] \mid t \in I_1, t$'s tag is unknown and $t[\bar{B}] \in \pi_{\bar{C}}(S)\}$ to the site of R.
3. Update the tags of the corresponding tuples in R using the set K.

□

A tag-free algorithm dealing with insertion is as follows:

Algorithm 2 Tag-free incremental for insertion

1. At the site of R, send $I_1 = \triangle$ to the site of S.
2. At the site of S, let $J_1 = \pi_{\bar{A}\bar{D}}(I_1 \bowtie_{\bar{B}=\bar{C}} S)$;
   let $\triangle_e = J_1$; this is the insertion to the view.

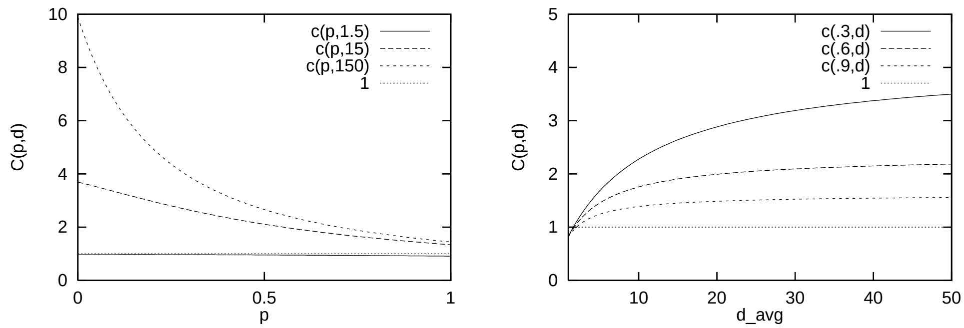
□

The principal parameters relevant to our analysis are the following:

- three average degrees: Let $\mathtt{deg}^A_{avg} = \frac{|R|}{|\pi_A(R)|}$, $\mathtt{deg}^B_{avg} = \frac{|R|}{|\pi_B(R)|}$, and $\mathtt{deg}^C_{avg} = \frac{|S|}{|\pi_C(S)|}$. Thus $\mathtt{deg}^A_{avg}$ is the average degree of each A-value in R, $\mathtt{deg}^B_{avg}$ is the average degree of each B-value in R, and $\mathtt{deg}^C_{avg}$ is the average degree of each C-value in S.
- two join ratios: Let p_B be the probability that a tuple of R can join with at least one tuple of S (i.e., the probability that the tag says yes), and let p_C be the probability that a tuple of S can join with at least one tuple of R.
- two operational probabilities: the probability of a deletion instead of an insertion (p_{del}) and the probability of the view being at site 1 instead of site 2 (p_{site1}). The former is significant because our algorithm gives more savings for deletion than for insertion. The latter is important because more work is needed if the view is stored at the same site as where modifications to the base relations are taking place. (See the communication cost analyses after Algorithms 1 and 2.)

The ratio C of the cost of the tag-free algorithms to the cost of the tagged algorithms can be expressed as follows (the derivation can be found in the appendix):

$$\frac{p_{site1} \cdot q_1 + (1 - p_{site1}) \cdot q_2}{p_{site1} \cdot q_3 + (1 - p_{site1}) \cdot q_4}$$



A) Fixed d_{avg} and varying p B) Fixed p and varying d_{avg}

Figure 3. $\frac{\text{cost of tag-free algorithms}}{\text{cost of tagged algorithms}}$, assuming $p_{in} = 1 - e^{-0.1}$

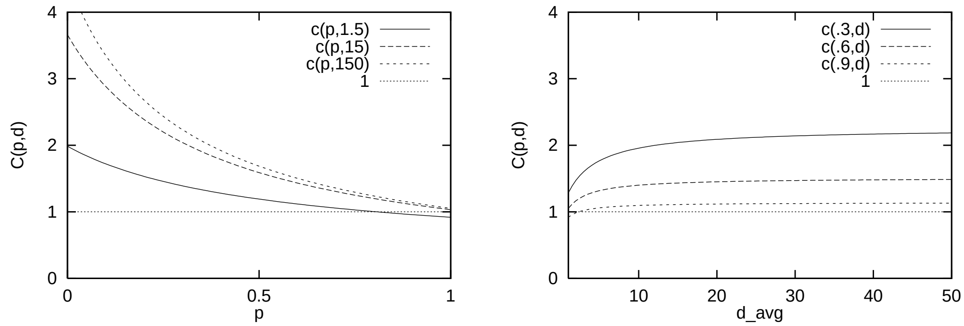
where

$$q_1 = p_{del} \cdot (\mathtt{deg}^A_{avg} + p_B \cdot \mathtt{deg}^C_{avg}) + (1 - p_{del}) \cdot (1 + p_B \cdot \mathtt{deg}^C_{avg})$$
$$q_2 = p_{del} \cdot \mathtt{deg}^A_{avg} + (1 - p_{del})$$
$$q_3 = p_{del} \cdot (p_B(1 + p_B(\mathtt{deg}^A_{avg} - 1)) + p_B \cdot \mathtt{deg}^C_{avg}) \\ +(1 - p_{del}) \cdot (p_{in} \cdot p_B(1 + \mathtt{deg}^C_{avg}) + (1 - p_{in})(1 + p_{in} \cdot \mathtt{deg}^C_{avg} + 1))$$
$$q_4 = p_{del} \cdot p_B(1 + p_B(\mathtt{deg}^A_{avg} - 1)) + (1 - p_{del}) \cdot (p_{in} \cdot p_B + 2(1 - p_{in}))$$

with $p_{in} = 1 - e^{-\frac{|R|}{|\mathtt{dom}_B|}}$ and $|\mathtt{dom}_B|$ being the size of the underlying (not the active) domain of B, i.e., the set of all possible values that might appear in $\pi_B(R)$. Intuitively, p_{in} is the probability of whether the B-value of a newly inserted tuple into R occurs in $\pi_B(R)$ already.

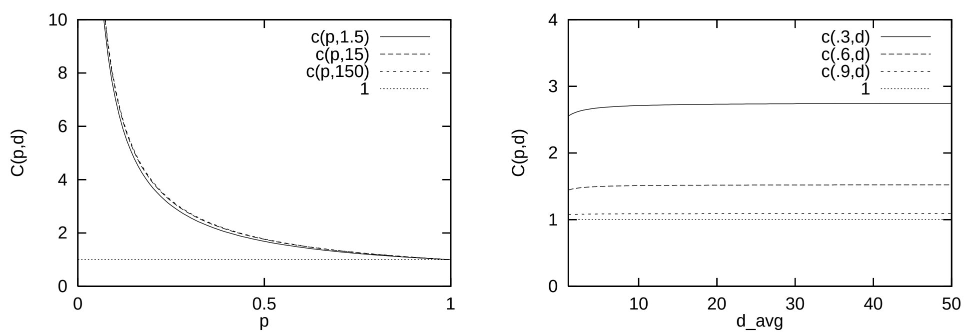
We will now examine the behavior of the ratio by drawing some diagrams. In order to draw these diagrams in two dimensions, we need to decrease the number of parameters: we let $p = p_B = p_C$ and let $d_{avg} = \mathtt{deg}^A_{avg} = \mathtt{deg}^B_{avg} = \mathtt{deg}^C_{avg}$; and we fix $p_{del} = 0.35$ and $p_{site1} = 0.5$. We also give $\frac{|R|}{|\mathtt{dom}_B|}$ different fixed values of 0.1, 1 and 10, that is, we give p_{in} three different values of $1 - e^{-0.1}, 1 - e^{-1}, 1 - e^{-10}$. In Figures 3A,4A,5A, we fix d_{avg} and let p (the x-axis) vary; the y-axis corresponds to the ratio C. In Figures 3B,4B,5B, we fix p and let d_{avg} (the x-axis) vary; the y-axis again corresponds to the ratio C.

In these graphs, wherever the value of C is greater than 1, it indicates a saving in communication cost due to tagging. On the other hand, a value less than 1 indicates an increase in communication due to tagging. It is obvious that the tagging algorithm offers greatest improvement for small p and large d_{avg}. This is intuitively reasonable, since if p is small, tuples in R are unlikely to join with those in S and so they do not need to be sent over the network to maintain the view. In fact, for small values of p a huge improvement is achieved by the tagging technique. In the case when d_{avg} is large, the work that must be done to maintain the view becomes larger (for deletion) and so tagging screens out more tuples before they need to be sent.



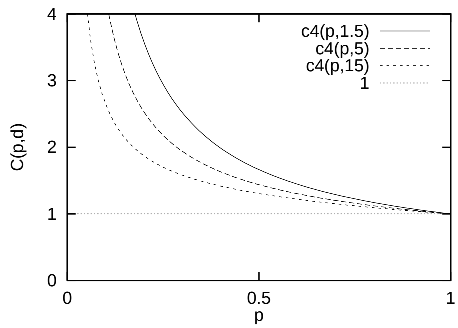
A) Fixed d_{avg} and varying p B) Fixed p and varying d_{avg}

Figure 4. $\frac{\text{cost of tag-free algorithms}}{\text{cost of tagged algorithms}}$, assuming $p_{in} = 1 - e^{-1}$

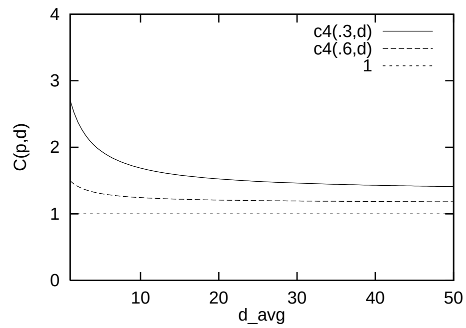


A) Fixed d_{avg} and varying p B) Fixed p and varying d_{avg}

Figure 5. $\frac{\text{cost of tag-free algorithms}}{\text{cost of tagged algorithms}}$, assuming $p_{in} = 1 - e^{-10}$

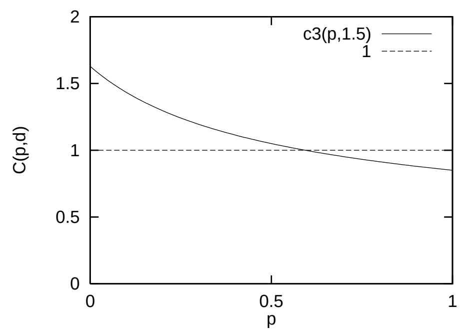


A) Fixed d_{avg} and varying p



B) Fixed p and varying d_{avg}

Figure 6. $\frac{\text{cost of counting}}{\text{cost of tagged counting}}$, assuming $p_{in} = 1 - e^{-10}$



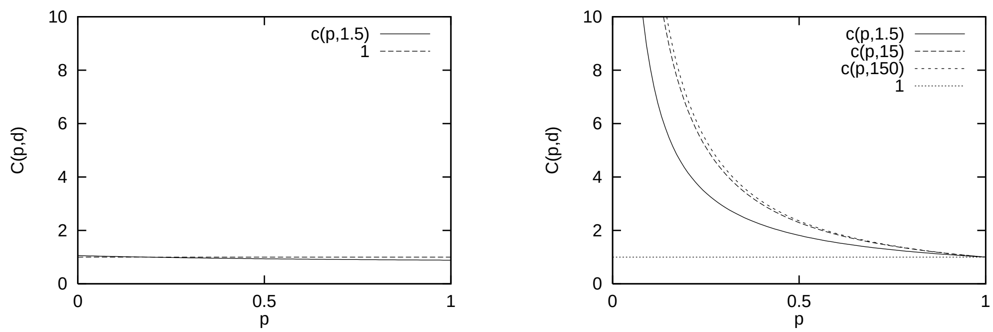
Fixed d_{avg} and varying p

Figure 7. $\frac{\text{cost of counting}}{\text{cost of tagged algorithms}}$, assuming $p_{in} = 1 - e^{-1}$

Observe that in Figure 5 the two dotted lines for c(p,15) and c(p,150) almost become identical.

Similarly, in Figure 6A and Figure 6B we have compared the cost of the distributed version of the counting algorithm [10] versus tagged counting algorithm. The parameters in these graphs are same as in Figure 4. The formula is similar to the one given earlier for the algorithms without counting (see Appendix for the formula and its derivation). Figure 7 shows that our tagged algorithms perform better than the counting algorithms.

We note that deletion operations result in more savings than insertion operations (see Figure 8). For a deletion, the amount of communication necessary to preserve the view is rather high and the work necessary to maintain tags is low in comparison. For insertion, however, tag maintenance costs are more significant compared to view maintenance costs and the result is a lower improvement. In our analysis, however,



A) Fixed d_{avg} and varying p for $p_{del} = 0$ B) Fixed d_{avg} and varying p for $p_{del} = 1$

Figure 8. $\frac{\text{cost of tag-free algorithms}}{\text{cost of tagged algorithms}}$, assuming $p_{in} = 1 - e^{-1}$: More Deletions, More savings

we have tended towards overestimation of the costs involved for tagging. If the relative sizes of various attributes were incorporated then it is likely that larger savings would be possible. Also, we have only examined the case where 35% of operations are deletions. This could be enlarged if updates were also considered and modelled as insertions + deletions. Increasing this value would improve the savings achieved since the tagging approach yields more benefit from deletions.

It should be pointed out that, when p_{in} gets small tagging leads to less savings or no savings; however, even for very small p_{in} tagging is about the same as no tagging in the worst case. Conversely, for large p_{in} tagging leads to more and significant savings.

Use of a uniform distribution of tuple values is a critical assumption of the foregoing discussion. If we were to use a skewed distribution then the cost behavior could be subject to large fluctuations. This is a subject for further research.

6.1. Exact-change Algorithms

In the above analysis, we have used a simplified method for insertion (Algorithms 1 and 2). This has the advantage of being usually cheaper than the exact-change method (Algorithm 2). It does, however, sacrifice the exact change feature. If we wish to preserve this feature, then Algorithm 2 should be used instead. The added cost this will incur arises from the fact that we are now additionally sending the sets $|I2|$ and $|J2|$. If we were to compare the cost of Algorithm 2 with that of Algorithm 2, then the tagging approach would only be beneficial for values of $p \leq 0.43$. This is not as bad as it seems, since we would be comparing apples and oranges – an exact change algorithm and a non-exact change one. Whenever we compare like with like though, tagging will result in more significant gains.

7. Related Research

A recent survey on strategies for view maintenance is given in [11]. The basic idea of most strategies is to update the views *incrementally* by using the (relatively small) change to the base relations, as well as the base relations and the old views themselves, to derive the change to the views (e.g., [1, 16, 21, 7, 10, 8, 6, 5]). More advanced strategies use auxiliary data structures to help the process. The tagging strategy reported in this paper falls into this category. Other auxiliary data structures adopted by view maintenance strategies include [1, 21, 10] using counters and [18] using pointers. These strategies all associate auxiliary data structures to the view. In contrast, our strategy uses tags on base relations. We showed how our strategy can be combined with the counting strategy as in [10]. With respect to [18], we might enhance its strategy by storing pointers in the base relations pointing to records in the ViewCache. These pointers can help reduce the communication cost since, for example, if a deletion occurs on a tuple with no pointers to the ViewCache, then there is no need to do anything to update the view. It is a further research issue as how to incrementally maintain these pointers, how costly is the space consumption (unlike our tags which requires two bits, a pointer takes more space), and how much improvement such pointers bring to view maintenance.

Recently, with the advent of data warehouse applications and mobile computing, managing materialized views derived from multiple, geographically distributed, underlying data sources has become a popular subject. In such a distributed environment, the above ideas of incremental maintenance can still be useful. However, since base relations are perhaps stored in geographically different sites, care must be taken to reduce the communication cost in obtaining the changes to views. The concept of *self-maintainable* views is suggested in [9] where the view site only needs the update to the underlying source relations in order to update the views. This technique cuts the communication cost since no source data needs to be transferred over the communication links other than the changes themselves. However, to achieve self-maintainability, it is sometime necessary to introduce certain auxiliary data or even duplicate some base relations. For this purpose, [17, 12] developed techniques to decide optimal trade-offs between the space and time needed for view maintenance under different scenarios. However, the view $R \bowtie S$ is not self-maintainable under insertion without assuming that R and S reside at the site of the view or that R and/or S satisfy certain constraints. Hence, our method can be viewed as complementary to the strategies studied in [10, 17, 12].

Another line of research into efficient view maintenance is to use a differential file which logs changes to the base relation and use certain tests to screen out irrelevant updates to the view in order to reduce the communication cost. Example of such strategies can be found in [20, 3]. In comparison with our algorithms, the work of [20] is limited by being only applicable to projection-selection views which must contain one candidate key and are defined on only one base table. The work of [3] is more recent and extends [20] to handle views defined with joins. However, the algorithm in [3] does not consider the issue of reducing communication cost when the participating base relations are in different sites. Also, it does not have the exact-change feature and hence requires counting as in [10] or certain semi-joins to decide the exact changes. On the other hand, [3] deals with views that are refreshed periodically.

There is other work related to view maintenance in distributed database systems. The work of [19] considered the issue of determining optimal policies for the timing of view updates, which include periodical, on-demand, random and hybrid. A more recent work, [22], describes techniques for view maintenance in a data warehousing environment. The algorithms in that paper, however, are principally directed towards methods dealing with the control of concurrent updates and query requests between the warehouse and the source.

Finally, the view maintenance problem in distributed databases is related to query optimization. Similar to query optimization in distributed databases [2, 13], our primary concern is to minimize the communication cost. Furthermore, the tagging strategy can be viewed as a version of the "semi-join" algorithm [2] improved for efficiently maintaining views. Indeed, for the view $R \bowtie S$, the tuples in R with "yes" tags are all those that are in the result of R semi-joining with S.

8. Conclusions and Further Research

We have seen that the incremental maintenance of views in distributed databases is an important problem for both traditional distributed database applications and many new real-life distributed database applications such as mobile computing, data warehousing, in-home digital services etc. As for distributed query processing, the main issue for distributed view maintenance is to minimize the communication cost. Our proposed mechanism is useful in such applications where views are mostly statically defined and dynamically maintained.

We have proposed two tagged differential incremental algorithms, one for insertion and other for deleting a set of tuples, using join-participation tags on base relations as a filter. These algorithms produce the exact change to the view. This feature makes it possible to maintain statistics on a view without materializing the view itself, since there are no references made to it. This feature is also useful in many active database applications where large number of rules and triggers are required, such as designing of workflow systems, audit trails, planning applications, index management etc. We have also improved the distributed version of the counting algorithm [10] by associating tags on base relations and further reduce the communication cost while maintaining the materialized view. This algorithm is called the tagged counting algorithm.

We note here that our algorithms will work even if base relations may contain duplicate tuples, i.e., a tuple may appear more than once. However, in order to obtain the exact change property, we assumed that the materialized views do not contain any duplicates. When exact change is not desired, the algorithms can be extended easily to handle duplicates in the materialized views.

We have also discussed extending our tagged differential algorithms for complicated views, and introduced memoing tags to 'remember' the values of intermediate subexpressions in view definitions. The space overhead incurred by tagging is very small. Indeed, each tuple becomes at most two bits longer for either the join tag or the memoing tag.

A statistical performance comparison between the tagged differential and non-tagged algorithms, and between counting and tagged counting algorithms has been done. The comparison showed the tagged version of each algorithm gives smaller communication cost than the non-tagged ones. The comparison also showed that the tagged differential algorithms are best used when join participation rate is less than 80%. From our analysis, we have also observed that tagged algorithm gives better results than the counting algorithm when join participation rate is low. However, the tagged counting and counting algorithms do not have the 'exact-change' advantage.

Observe that, when views are initially defined, there are initial cost to initialize the tags. The associated cost is comparable to the cost of deriving the contents of the view, using semi-joins. Thus we believe our approach is still useful for such situations as well.

We now list some potential topics for further research. Some of these were briefly touched upon in our current paper.

- Find the best methods to maintain views having complicated definitions.
- Study the behavior of the algorithms for non uniform distribution.
- Examine how constraints can help maintain views more efficiently.
- Similar to counters on view tuples [10], can one attach counters (or more general aggregates) to base relations as auxiliary data for additional saving of communication?
- Examine how to maintain views defined by multiple joins. Since our method is in a way similar to semi-join, the research about semi-join plans for multiple joins (e.g., [15]) is relevant here.
- Since the mechanism proposed is basically for views whose definitions are relatively static, how can one extend the idea to treat dynamically defined views?

References

1. J. A. Blakeley, P.-A. Larson, and F. W. Tompa. Efficiently updating materialized views. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 61–71, 1986.
2. S. Ceri and G. Pelagatti. *Distributed Databases: Principles and Systems.* McGraw-Hill Book Co, 1986.
3. D. Chao, G. Diehr, and A. N. Saharia. Maintaining join-based remote snapshots using relevant logging. In *Proc. of the Workshop on Materialized Views: Techniques and Applications (in cooperation with ACM SIGMOD)*, June 1996.
4. G. Dong and J. Su. Incremental and decremental evaluation of transitive closure queries by first-order queries (extended abstract). In *Proc. 16th Australian Computer Science Conference*, 1993. Full version appears in [5].
5. G. Dong and J. Su. Incremental and decremental evaluation of transitive closure by first-order queries. *Information and Computation*, 120(1):101–106, July 1995. Preliminary version in [4].

6. G. Dong, J. Su, and R. Topor. Nonrecursive incremental evaluation of Datalog queries. *Annals of Mathematics and Artificial Intelligence*, 14(2-4):187–223, 1995.
7. G. Dong and R. Topor. Incremental evaluation of Datalog queries. In *Proc. Int'l Conference on Database Theory*, pages 282–296, Berlin, Germany, October 1992. Full version appears as part of [6].
8. T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 1995.
9. A. Gupta, H.V. Jagadish, and I. S. Mumick. Data integration using self-maintainable views. In *Proc. First Int. Conf. on Extending Database Technology*, 1996.
10. A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 157–166, 1993.
11. A. Gupta and I. S. Mumick. Maintenance of materialized views: problems, techniques, and applications. *IEEE Data Engineering Bulletin, Special Issue on Materialized Views and Warehousing*, 18(2), 1995.
12. R. Hull and G. Zhou. A framework for supporting data integration using the materialized and virtual approaches. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 481–492, 1996.
13. H. F. Korth and A. Silberschatz. *Database System Concepts.* McGraw-Hill, 1991.
14. P. Mishra and M.H. Eich. Join processing in relational databases. *ACM Computing Surveys*, 24(1):63–113, 1992.
15. M.T. Özsu and P. Valduriez. *Principles of Distributed Database Systems.* Prentice-Hall, 1991.
16. X. Qian and G. Wiederhold. Incremental recomputation of active relational expressions. *IEEE Trans. on Knowledge and Data Engineering*, 3(3):337–41, 1991.
17. K. A. Ross, D. Srivastava, and S. Sudarshan. Materialized view maintenance and integrity constraint checking: Trading space for time. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, page to appear, 1996.
18. N. Roussopoulos. The incremental access method of ViewCache: Concept, algorithm, and cost analysis. *ACM Trans. on Database Systems*, 16(3):535–563, 1991.
19. A. Segev and W. Fang. Currency based updates to distributed materialized views. In *Proceedings of the IEEE International Conference on Data Engineering*, 1990.
20. A. Segev and J. Park. Maintaining materialized views in distributed databases. In *Proceedings of the IEEE International Conference on Data Engineering*, 1989.
21. O. Wolfson, H. M. Dewan, S. J. Stolfo, and Y. Yemini. Incremental evaluation of rules and its relationship to parallelism. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 78–87, 1991.
22. Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 316–27, 1995.

Appendix

We now show how to derive the formulas used in Section 6 under the uniform distribution assumption.

Recall that we are considering the view defined by $\pi_{AD}(R \bowtie_{B=C} S)$ over the two base relations $R(AB)$ and $S(CD)$, where R is at site 1 and S at site 2.

The following analysis contains some approximations; these decisions are needed to enable us to derive a reasonable formula. One of such approximations is that we assumed that the active domain (such as $\pi_B(R)$) is roughly the same as the underlying domain (e.g. $dom(B)$) where the attribute values are drawn from.

Let $C1_{nt,del}$ be the expected number of tuples that must be sent to derive the change to the view which is at site 1, when tagging is not used, for a deletion operation at site 1. Let I_1, I_2, J_1, and J_2 be those relations produced in the process, by the tag-free algorithm described immediately after Algorithm 1. (The average will always be taken over all tuples in R.) Then

$$
\begin{aligned}
C1_{nt,del} &= \texttt{avg}(|I_1| + |I_2| + |J_1 - J_2|) \\
&\approx \texttt{avg}(|I_1| + |I_2|) + \texttt{avg}(|J_1 - J_2|) \\
&= \texttt{deg}^A_{avg} + \texttt{avg}(|J_1 - J_2|) \\
&\approx \texttt{deg}^A_{avg} + \texttt{avg}(|J_1| \cdot (1 - \tfrac{1}{|\pi_A(R)|\cdot|\pi_B(R)|\cdot|\pi_C(S)|\cdot|\pi_D(S)|})), \\
&\qquad \text{since } 1 - \tfrac{1}{|\pi_A(R)|\cdot|\pi_B(R)|\cdot|\pi_C(S)|\cdot|\pi_D(S)|} \text{ is approximately the probability} \\
&\qquad \text{that a tuple is not in } J_2 \\
&\approx \texttt{deg}^A_{avg} + \texttt{avg}(|J_1|), \text{ assuming the domains are large} \\
&= \texttt{deg}^A_{avg} + p_B \cdot \texttt{deg}^C_{avg}.
\end{aligned}
$$

Let $C1_{nt,ins}$ be the expected number of tuples that must be sent to derive the change to the view which is at site 1, when tagging is not used, for an insertion operation at site 1. Let I_1 and J_1 be those relations produced in the process, by the tag-free algorithm described immediately after Algorithm 1. Then

$$C1_{nt,ins} = \texttt{avg}(|I_1| + |J_1|) = 1 + p_B \cdot \texttt{deg}^C_{avg}.$$

Let $C2_{nt,del}$ (and $C2_{nt,ins}$) be the expected number of tuples that must be sent to derive the change to the view which is at site 2, when tagging is not used, for an insertion (deletion) operation at site 1. Let I_1 and I_2 be those relations produced in the process, by the tag-free algorithm described immediately after Algorithm 1. Then

$$
\begin{aligned}
C2_{nt,del} &= \texttt{avg}(|I_1| + |I_2|) = \texttt{deg}^A_{avg}. \\
C2_{nt,ins} &= \texttt{avg}(|I_1|) = 1.
\end{aligned}
$$

Let $C1_{t,del}$, $C1_{t,ins}$, $C2_{t,del}$, and $C2_{t,ins}$ be the corresponding expected communication costs when the tagging algorithms are used. To derive these costs for the insertion cases, we need two useful intermediate probabilities.

Let p_{in} be the probability that a B value b is already in $\pi_B(R)$. Then

$$
\begin{aligned}
p_{in} &= 1 - \text{the probability that all values in } \pi_B(R) \text{ are different from } b \\
&\approx 1 - (1 - \tfrac{1}{|\texttt{dom}_B|})^{|R|} \\
&\approx 1 - e^{-\frac{|R|}{|\texttt{dom}_B|}}, \text{ since } |R| \gg 1 \text{ and } |\texttt{dom}_B| \gg 1
\end{aligned}
$$

Using p_{in}, we can now derive the other costs:

$$
\begin{aligned}
C1_{t,del} &= \texttt{avg}(|I_1| + |I_2| + |J_1 - J_2|), \\
&\qquad \text{where } I_1, I_2, J_1 \text{ and } J_2 \text{ are as produced by Algorithm 1} \\
&\approx \texttt{avg}(|I_1| + |I_2|) + \texttt{avg}(|J_1 - J_2|) \\
&= p_B(1 + p_B(\texttt{deg}^A_{avg} - 1)) + \texttt{avg}(|J_1 - J_2|) \\
&\approx p_B(1 + p_B(\texttt{deg}^A_{avg} - 1)) + p_B \cdot \texttt{deg}^C_{avg}, \\
&\qquad \text{recall that } \texttt{avg}(|J_1 - J_2|) \approx p_B \cdot \texttt{deg}^C_{avg} \text{ from the derivation of } C1_{nt,del}.
\end{aligned}
$$

$$
\begin{aligned}
C1_{t,ins} &= p_{in} \cdot \texttt{avg}(|I_1^i| + |J_1^i|) + (1 - p_{in}) \cdot \texttt{avg}(|I_1^n| + |J_1^n| + |K|),\\
&\qquad \text{where } I_1^i \text{ and } J_1^i \text{ are as produced by Algorithm 1}\\
&\qquad\qquad \text{when the } b \text{ value of the inserted tuple is in } \pi_B(R)\\
&\qquad \text{and } I_1^n,\ J_1^n \text{ and } K \text{ are as produced by Algorithm 1}\\
&\qquad\qquad \text{when the } b \text{ value of the inserted tuple is not in } \pi_B(R)\\
&= p_{in} \cdot p_B(1 + \texttt{deg}^C_{avg}) + (1 - p_{in})(1 + p^C_{in} \cdot \texttt{deg}^C_{avg} + 1)\\
&= p_{in} \cdot p_B(1 + \texttt{deg}^C_{avg}) + (1 - p_{in})(2 + p^C_{in} \cdot \texttt{deg}^C_{avg})
\end{aligned}
$$

$$
\begin{aligned}
C2_{t,del} &= \texttt{avg}(|I_1| + |I_2|),\\
&\qquad \text{where } I_1 \text{ and } I_2 \text{ are as produced by Algorithm 1}\\
&= p_B(1 + p_B(\texttt{deg}^A_{avg} - 1))
\end{aligned}
$$

$$
\begin{aligned}
C2_{t,ins} &= p_{in} \cdot \texttt{avg}(|I_1^i|) + (1 - p_{in}) \cdot \texttt{avg}(|I_1^n| + |K|),\\
&\qquad \text{where } I_1^i \text{ is as produced by Algorithm 1}\\
&\qquad\qquad \text{when the } b \text{ value of the inserted tuple is in } \pi_B(R)\\
&\qquad \text{and } I_1^n \text{ and } K \text{ are as produced by Algorithm 1}\\
&\qquad\qquad \text{when the } b \text{ value of the inserted tuple is not in } \pi_B(R)\\
&= p_{in} \cdot p_B + 2(1 - p_{in})
\end{aligned}
$$

Finally,

$$
\begin{aligned}
q_1 &= p_{del} C1_{nt,del} + (1 - p_{del}) C1_{nt,ins}\\
q_2 &= p_{del} C2_{nt,del} + (1 - p_{del}) C2_{nt,ins}\\
q_3 &= p_{del} C1_{t,del} + (1 - p_{del}) C1_{t,ins}, \text{ and}\\
q_4 &= p_{del} C2_{t,del} + (1 - p_{del}) C2_{t,ins}.
\end{aligned}
$$

Since we also gave comparisons for the tagged/tag-free counting algorithms, below we give the analysis of their costs. Let $CG1_{t,del}$, $CG1_{t,ins}$, $CG2_{t,del}$, and $CG2_{t,ins}$ be the corresponding expected communication costs when the tagged counting algorithms are used. Let $CG1_{nt,del}$, $CG1_{nt,ins}$, $CG2_{nt,del}$, and $CG2_{nt,ins}$ be the corresponding expected communication costs when the tag-free counting algorithms are used. For simplicity, assume there $p_B = p_C = p$. Then:

$$
\begin{aligned}
CG1_{nt,ins} &= C1_{nt,ins}\\
CG1_{nt,del} &= C1_{nt,ins}\\
CG1_{t,ins} &= C1_{t,ins}\\
CG1_{t,del} &= p * CG1_{nt,del}\\
CG2_{nt,ins} &= 1\\
CG2_{nt,del} &= 1\\
CG2_{t,ins} &= C2_{t,ins}\\
CG2_{t,del} &= p * CG2_{nt,del}
\end{aligned}
$$

We now derive the formulas for the cost of the algorithms which ensure the exact-change feature. We modify the corresponding symbols for the earlier symbols by adding the superscript e: $C1^e_{t,ins}, C1^e_{t,del}, C1^e_{nt,ins}, C1^e_{nt,del}, C2^e_{t,ins}, C2^e_{t,del}, C2^e_{nt,ins}, C2^e_{nt,del}$.

It is easy to see that the following hold for each $1 \le i \le 2$:

$$Ci^e_{t,del} = Ci_{t,del}$$
$$Ci^e_{nt,del} = Ci_{nt,del}$$
$$Ci^e_{nt,ins} = Ci_{nt,del}$$
$$Ci^e_{t,ins} = Ci_{t,del} + 2(1 - p_{in}).$$