Content-Based Filtering for Efficient Online Materialized View Maintenance

Gang Luo Philip S. Yu
IBM T.J. Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532, USA
luog@us.ibm.com psyu@us.ibm.com

ABSTRACT

Real-time materialized view maintenance has become increasingly popular, especially in real-time data warehousing and data streaming environments. Upon updates to base relations, maintaining the corresponding materialized views can bring a heavy burden to the RDBMS. A traditional method to mitigate this problem is to use the where clause condition in the materialized view definition to detect whether an update to a base relation is relevant and can affect the materialized view. However, this detection method does not consider the content in the base relations and hence misses a large number of filtering opportunities. In this paper, we propose a content-based method for detecting irrelevant updates to base relations of a materialized view. At the cost of using more space, this method increases the probability of catching irrelevant updates by judiciously designing filtering relations to capture the content in the base relations. Based on the content-based method, a prototype real-time data warehouse has been implemented on top of IBM's System S using IBM DB2. Using an analytical model and our prototype, we show that the content-based method can catch most (or all) irrelevant updates to base relations that are missed by the traditional method. Thus, when the fraction of irrelevant updates is non-negligible, the load on the RDBMS due to materialized view maintenance can be significantly reduced.

Categories and Subject Descriptors
H.2.4 [Systems]: query processing, relational databases, H.2.7 [Database Administration]: data warehouse and repository
General Terms
Algorithms, Experimentation, Performance
Keywords
Materialized view maintenance, content-based filtering

1. INTRODUCTION

Recently, there has been a growing trend to use data warehouses to make real-time decisions about a corporation's day-to-day operations. Most major RDBMS vendors have spent great efforts on real-time data warehousing, including IBM's business intelligence, Microsoft's digital nervous system, Oracle's Oracle10g, NCR's active data warehouse, and Compaq's zero-latency enterprise. Since conventional RDBMSs cannot provide all the capabilities required by real-time data warehousing, a few startup companies have appeared on this market, e.g., GoldenGate Software Inc., DataMirror, and Information Builders.

CIKM'08, October 26–30, 2008, Napa Valley, California, USA.

A real-time data warehouse needs to handle real-time, online updates in addition to traditional data warehouse query workload. This raises a problem that is present to a lesser degree in traditional data warehouses – when a base relation is updated, maintaining the materialized view(s) defined on it can bring a heavy burden to the RDBMS.

This problem is not limited to real-time data warehousing, as real-time materialized view maintenance is a general requirement of modern database applications. For example, the DB research community has recently identified data stream management systems as a promising approach to support many future data-intensive applications [13]. Storing data streams on disks and building materialized views on them to speed up query processing has been proposed in [5]. In fact, the DB research community has realized the commonality between real-time data warehousing and data stream applications [8, 13]. Also, some commercial RDBMS vendors (e.g., Oracle, Teradata) have started to enhance existing commercial RDBMSs to support data stream applications.

To reduce materialized view maintenance overhead, [2, 4] proposed several methods to detect irrelevant updates to a base relation R that do not affect the materialized view MV defined on R. However, all these methods are "*content-independent*" in the sense that they only consider the where clause condition in MV's definition while ignoring the content in the other base relations of MV. As a result, these methods make over-conservative decisions and miss a large number of filtering opportunities.

For example, consider the following materialized view MV:

create materialized view MV as select * from R, S, T
where $R.a=S.b$ and $S.c=T.d$ and $R.e>20$ and $S.f$="xyz" and $T.g=50$;

Assume that MV records anomaly so that very few tuples in R, S, and T satisfy the where clause condition ($R.a=S.b$ and $S.c=T.d$ and $R.e>20$ and $S.f$="xyz" and $T.g=50$) in MV's definition. Suppose a tuple t_R whose $t_R.e=30$ is inserted into base relation R. Since $t_R.e>20$, the existing methods in [2, 4] cannot tell whether or not MV will change. Therefore, the standard materialized view maintenance method has to be used. S is checked for matching tuple(s) t_S such that $t_S.b=t_R.a$ and $t_S.f$="xyz". If such a matching tuple t_S exists, T is further checked for matching tuple(s) t_T such that $t_T.d=t_S.c$ and $t_T.g=50$. If both S and T are large and cannot be cached in memory, such checking can incur a large number of I/Os and become fairly expensive. However, since MV records anomaly, mostly likely the insertion of t_R into R will not affect MV and thus all the expensive checking is wasted.

To address this problem, we introduce the concept of content-based filtering into materialized view maintenance. More specifically, we identify four important requirements for efficient filtering and propose a content-based method for detecting irrelevant updates to base relations of a materialized view. To the best of our knowledge, no existing summary data structure [1, 3] satisfies all these four requirements. Our key idea is to design *filtering relations* that summarize the most relevant information in the base relations and fulfill these four requirements. These

filtering relations capture the relationship among multiple join attributes and can be efficiently maintained in real time. Upon an update ΔR to a base relation R that has a materialized view MV defined on it, the RDBMS uses the corresponding filtering relations of the other base relations of MV to tell whether ΔR is irrelevant. Checking filtering relations is usually much faster than checking base relations. Also, compared to the where clause condition in MV's definition, filtering relations can provide more precise information about whether ΔR is irrelevant. In this way, the RDBMS can quickly and more precisely detect irrelevant updates to R and hence reduce the materialized view maintenance overhead. As discussed in detail in Section 4, existing RDBMSs do not have the capability of automatically building filtering relations and systematically using them to perform content-based filtering for materialized view maintenance.

We have implemented a prototype real-time data warehouse SRW (Stream-based Real-time Warehouse) on top of IBM's System S [16], a stream processing middleware that provides an application execution environment for processing elements (or applications) developed by users to filter and analyze data streams. Our real-time data warehouse is deployed as a processing element on top of System S. It uses IBM DB2, and materialized views stored there are maintained in real-time using our content-based method. Figure 1 shows the architecture of our prototype. Data continuously comes to System S as streams. Some "base" data streams are stored in the real-time data warehouse and materialized views are built upon them. System S interacts with the real-time data warehouse, using materialized views to speed up the processing of data streams (the processing of certain data streams requires posing queries to the RDBMS [5]). Also, upon arrival of new tuples from the "base" data streams, the corresponding materialized views are refreshed immediately.

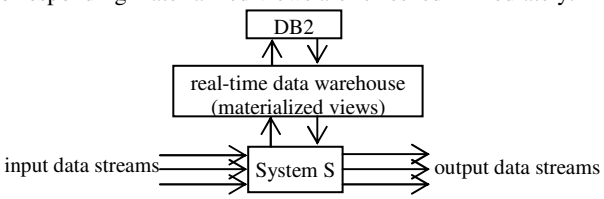


Figure 1. Architecture of our SRW prototype.

We investigate the performance of both the content-based method and the traditional content-independent method with an analytical model. This analytical model provides a means to determine when applying filtering relations is beneficial. The analytical model is validated in our SRW prototype. Our results show that in a large number of cases, with minor overhead, the content-based method can catch most (or all) irrelevant updates to base relations that are missed by the content-independent method. As a result, we can avoid most of the unnecessary load on the RDBMS due to materialized view maintenance.

The rest of the paper is organized as follows. Section 2 describes the content-based detection method for irrelevant updates. Section 3 investigates the performance of both the content-based and the content-independent methods. We discuss related work in Section 4.

2. A CONTENT-BASED DETECTION METHOD FOR IRRELEVANT UPDATES

In this section, we describe our content-based method for detecting irrelevant updates to base relations of a materialized view. We focus on materialized *join views* that store and maintain the join results of multiple base relations. Updates include insertions, deletions, and modifications.

2.1 Requirements for Summary Data Structures

Consider a base relation R that has a join view JV defined on it. Our goal is to quickly filter out most of the irrelevant updates to R. This filtering process allows false negatives for irrelevant updates but not false positives. In other words, for any update ΔR to R, this filtering process has the following characteristics:

(1) If our method says that ΔR is irrelevant, it must be true that ΔR is irrelevant.

(2) In the case that ΔR is irrelevant, with high probability p, our method can tell that ΔR is irrelevant; with low probability $1-p$, our method says that it does not know whether ΔR is irrelevant.

(3) In the case that ΔR is relevant, our method says that it does not know whether ΔR is irrelevant.

Our key idea is to design effective summary data structures that satisfy the following properties:

Compactness: They are small and likely to be cached in memory. This is crucial for real-time purposes.

Association: They can capture the relationship among multiple join attributes of a base relation – given a join attribute value (e.g., $S.b$ of MV in the introduction), we can use them to find the associated values of other join attributes (e.g., $S.c$).

High filtering ratio: They can quickly and correctly filter out most (or all) of the irrelevant updates to base relations of a join view.

Easy maintenance: Upon updates to base relations, they can be efficiently maintained in real time.

There are several existing summary data structures (e.g., bloom filters, multi-attribute B-tree indices). However, as will be shown in Section 2.5, none of them satisfies all above four properties and is suitable for our filtering purposes. In the following, we first give an overview of our content-based detection method for irrelevant updates. Then we present the details of our algorithm.

2.2 Overview of the Method

Consider a join view JV that is defined on base relations R_1, R_2, …, and R_n ($n \geq 2$). For each R_i ($1 \leq i \leq n$), we create a filtering relation FR_i that summarizes the most relevant information in R_i. Upon an update ΔR_i to a base relation R_i ($1 \leq i \leq n$) of JV, our method performs the following operations:

Operation O_1: Update the filtering relation FR_i accordingly.

Operation O_2: Use the where clause condition in JV's definition and the techniques in [2, 4] to detect whether ΔR_i is irrelevant.

Operation O_3: If *Operation* O_2 cannot tell that ΔR_i is irrelevant, check the filtering relations FR_1, FR_2, …, FR_{i-1}, FR_{i+1}, FR_{i+2}, …, and FR_n to see whether ΔR_i is irrelevant.

Operation O_4: If *Operation* O_3 cannot tell that ΔR_i is irrelevant, check base relations R_1, R_2, …, R_{i-1}, R_{i+1}, R_{i+2}, …, and R_n to see exactly whether ΔR_i is irrelevant. In the case that ΔR_i is relevant, JV is refreshed. This is the standard join view maintenance method.

2.3 Basic Algorithm Description

Suppose that C_w is the where clause condition in the definition of the join view JV. C_w is rewritten into a conjunction of m terms

c_i ($1 \le i \le m$). Each term c_i belongs to one of the following three categories:

Category 1: For each i ($1 \le i \le m_1$), c_i is a conjunctive equi-join condition on two base relations R_j and R_k ($1 \le j < k \le n$). That is, c_i is of the conjunctive form $R_j.a_1=R_k.b_1 \wedge R_j.a_2=R_k.b_2 \wedge \ldots \wedge R_j.a_h=R_k.b_h$ ($h \ge 1$). For different i's ($1 \le i \le m_1$), either the corresponding j's or the corresponding k's are different.

Category 2: For each i ($m_1+1 \le i \le m_2$), c_i is a selection condition on a single base relation R_j ($1 \le j \le n$). For different i's ($m_1+1 \le i \le m_2$), the corresponding j's are different.

Category 3: For each i ($m_2+1 \le i \le m$), c_i is neither a conjunctive equi-join condition on two base relations nor a selection condition on a single base relation.

For example, consider the join view *MV* mentioned in the introduction. The where clause condition in *MV*'s definition is a conjunction of five terms. The first two terms (*R.a=S.b* and *S.c=T.d*) belong to Category 1. The other three terms (*R.e>20*, *S.f*="xyz", and *T.g=50*) belong to Category 2. An example term of Category 3 is *R.x+S.y>T.z*, which does not appear in the where clause condition of *MV*'s definition.

For each base relation R_i ($1 \le i \le n$), we create a filtering relation $FR_i=\pi_D(\sigma_C(R_i))$. The projection list D contains all join attributes of R_i that appear in some term of Category 1. That is, for each term c_j ($1 \le j \le m_1$) that is of the form $R_i.a_1=R_k.b_1 \wedge R_i.a_2=R_k.b_2 \wedge \ldots \wedge R_i.a_h=R_k.b_h$ ($1 \le k \le n$, $k \ne i$, $h \ge 1$), attributes $\{a_1, a_2, \ldots, a_h\} \subseteq D$. Also, we build an index on attributes (a_1, a_2, …, a_h). The selection condition C is the term of Category 2 that is on R_i. That is, if for some j ($m_1+1 \le j \le m_2$), the term c_j is a selection condition on R_i, then $C=c_j$. Otherwise (i.e., if no such c_j exists), we have $C=true$.

For example, consider the join view *MV* mentioned in the introduction. We create three filtering relations, as shown in Figure 2.

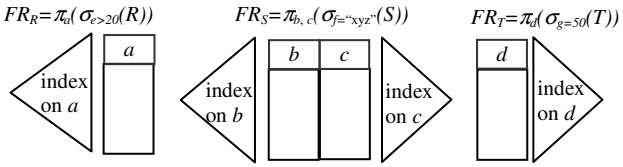


Figure 2. Example filtering relations.

In *Operation* O_3, upon an update ΔR_i to base relation R_i ($1 \le i \le n$), the updated tuples in R_i are joined with the corresponding filtering relations of the other base relations of *JV* (i.e., FR_1, FR_2, …, FR_{i-1}, FR_{i+1}, FR_{i+2}, …, and FR_n). If no join result tuple is generated, our method knows that ΔR_i is irrelevant. Otherwise our method does not know whether ΔR_i is irrelevant unless it checks the other base relations R_1, R_2, …, R_{i-1}, R_{i+1}, R_{i+2}, …, and R_n. This is because in checking the filtering relations, the terms in Category 3 are ignored and hence we may have false negatives.

When the updated tuples in R_i are joined with the filtering relations FR_1, FR_2, …, FR_{i-1}, FR_{i+1}, FR_{i+2}, …, and FR_n, our method only cares whether the join result set J_S is empty. Hence, during the join process, two optimizations are used to reduce the join overhead. First, some attributes are projected out immediately after they are no longer needed. Second, for a filtering relation whose corresponding base relation is joined with only one other base relation of the join view, if there are multiple matching tuples in the filtering relation for an input tuple, our method only finds the first matching tuple rather than all matching tuples. In other words, for each input tuple to such a filtering relation, our method generates at most one join result tuple. These two optimizations essentially compute a subset S_S of the projection of J_S and ensure that $S_S = \phi \Leftrightarrow J_S = \phi$. The details of these two optimizations are straightforward and thus omitted here. Rather, we use two examples to illustrate the point.

Consider the join view *MV* mentioned in the introduction. To illustrate the first optimization, consider an update ΔR to base relation R. In this case, our method only joins $\pi_a(\Delta R)$ with the filtering relation FR_S. Then for the join result $J_r=\pi_a(\Delta R) \bowtie_{a=b} FR_S$, attributes a and b are projected out before J_r is joined with FR_T. If either J_r or $\pi_c(J_r) \bowtie_{c=d} FR_T$ is empty, our method knows that ΔR is irrelevant. Actually in this case, the content-based method can catch all irrelevant updates to base relations. Thus, if we ignore the overhead of checking/updating filtering relations, the content-based method avoids all unnecessary join view maintenance overhead in the content-independent method. (As will be shown in Section 3 below, the overhead of checking/updating filtering relations is often minor.)

To illustrate the second optimization, suppose tuple t_S is inserted into S. In the filtering process, our method joins tuple $t_{S1}=\pi_{b,\ c}(t_S)$ first with FR_R, and then with FR_T. When our method searches in FR_R, once it finds the first tuple t_R matching t_{S1}, it generates the join result tuple $t_j=\pi_c(t_R \bowtie_{a=b} t_{S1})$, stops the search in FR_R, and continues to do the join with FR_T. This is because the attributes of FR_R do not include the join attribute c with FR_T. Therefore, from the perspective of determining whether the join result with FR_T is empty, there is no need to obtain more tuples in FR_R that match t_{S1}. (If no tuple in FR_R is found to match t_{S1}, we know that tuple t_S is irrelevant.) Similarly, when our method searches in FR_T, once it finds the first tuple matching t_j, it stops the search in FR_T.

In the traditional join view maintenance method, the work needed when base relation R_i ($1 \le i \le n$) is updated is as follows:

update R_i*;*
Operation O_2*; /* check where clause condition in JV definition */*
If (Operation O_2 *fails)*
  Operation O_4*; (expensive) /* maintain JV using base relations */*

When we say *Operation* O_2 fails, we mean that *Operation* O_2 cannot tell whether the update to R_i is irrelevant.

For comparison, in our content-based detection method, the work needed when base relation R_i ($1 \le i \le n$) is updated is as follows:

update R_i*;*
Operation O_1*; (cheap) /* update* FR_i **/*
Operation O_2*; /* check where clause condition in JV definition */*
If (Operation O_2 *fails)*
  Operation O_3*; (cheap) /* check filtering relations */*
  If (Operation O_3 *fails)*
    Operation O_4*; (expensive) /* maintain JV using base relations */*

Usually, due to selection and projection, filtering relations are much smaller than base relations and thus more likely to be cached in memory. In this case, checking filtering relations is much faster than checking base relations. If a not-very-small percentage of updates to base relations are irrelevant and using filtering relations can filter out most of the irrelevant updates, the extra work of (cheap) *Operations* O_1 and O_3 is dominated by the work saved in the expensive *Operation* O_4. As a result, the total join view maintenance overhead is greatly reduced.

Note that in order to minimize the sizes of filtering relations (the compactness property), the terms in Category 3 are not considered in filtering relations and thus get ignored in the

filtering process. As will be shown in Section 3 below, using the terms in Categories 1 and 2 is usually sufficient to filter out most of the irrelevant updates.

2.4 Improvements to the Basic Algorithm

In order to enhance the compactness of filtering relations, efficiency, and functionality, we present several improvements to the basic algorithm.

2.4.1 Compressing Filtering Relations

The performance advantages of the content-based detection method depend heavily on the sizes of filtering relations. The smaller the filtering relations, the more likely they can be cached in memory and thus the greater performance advantages of the content-based detection method. Therefore, it is beneficial to reduce the sizes of filtering relations.

To achieve this size reduction goal, we use the following hashing method. For each term c_i ($1 \le i \le m_1$) of Category 1 that is of the form $R_j.a_1=R_k.b_1 \wedge R_j.a_2=R_k.b_2 \wedge \ldots \wedge R_j.a_h=R_k.b_h$ ($1 \le j < k \le n$, $h \ge 1$), if the representation of attributes (a_1, a_2, …, a_h) is longer than that of an integer attribute, we use a hash function H to map each (a_1, a_2, …, a_h) into an integer. In the filtering relation FR_j of base relation R_j, we store $H(a_1, a_2, \ldots, a_h)$ rather than (a_1, a_2, …, a_h). Also, in the filtering relation FR_k of base relation R_k, we store $H(b_1, b_2, \ldots, b_h)$ rather than (b_1, b_2, …, b_h).

In practice, a large number of joins are based on key/foreign key attributes and the values of these attributes are usually long strings (e.g., ids). Therefore, hashing can often reduce the sizes of filtering relations significantly.

Suppose a hash function H (or multiple hash functions) has been applied to the filtering relation FR_i of base relation R_i ($1 \le i \le n$). Upon an update ΔR_i to R_i, H is first applied to the corresponding join attributes of the updated tuples ΔR_i. Then ΔR_i is joined with the filtering relations FR_1, FR_2, …, FR_{i-1}, FR_{i+1}, FR_{i+2}, …, and FR_n.

In the above hashing method, due to hash conflicts, we may introduce false negatives in detecting irrelevant updates using filtering relations. However, typical modern computers can represent a large number of distinct integer values (e.g., a 32-bit computer can represent 2^{32} distinct integer values). In practice, if a good hash function is used, the probability of having hash conflicts should be low. As a result, this hashing method will not introduce a large number of false negatives.

2.4.2 Reducing the Number of Filtering Relations

In practice, most updates occur to one (or a few) base relation. The other base relations are rarely updated. In this case, our method can only keep filtering relations for the rarely updated base relations. No filtering relation is kept for the most frequently updated base relation. Then for the update to the mostly frequently updated base relation (i.e., for most updates to the base relations), the filtering relation maintenance overhead is avoided. As a tradeoff, when some rarely updated base relation is updated (i.e., for a few updates to the base relations), the content-based detection method cannot be used. Rather, we go back to the standard join view maintenance method.

Suppose base relation R_i ($1 \le i \le n$) is small enough to be cached in memory in most cases. Also, no hash function has been applied to the corresponding filtering relation FR_i. Then there is no need to keep FR_i. Rather, in *Operation* O_3, when we check filtering relations for irrelevant updates to some other base relation R_j ($1 \le j \le n$, $j \ne i$), we use base relation R_i and filtering relation FR_k's ($1 \le k \le n$, $k \ne i$, $k \ne j$). (We may build some indices on the join attributes of R_i.) This can save the maintenance overhead of FR_i when R_i is updated.

2.4.3 Relaxing the Equi-join Condition of Category 1

For each term of Category 1, we restrict the equi-join condition on two base relations R_j and R_k ($1 \le j < k \le n$) to be of conjunctive form. This condition can be relaxed so that for each term of Category 1, the equi-join condition on R_j and R_k is of disjunctive-conjunctive form $\bigvee_{r=1}^{t} \left(\bigwedge_{s=1}^{h_r} R_j.a_{i_{r,s}} = R_k.b_{i_{r,s}} \right)$, where $t \ge 1$ and $h_r \ge 1$ ($1 \le r \le t$). Then for each r ($1 \le r \le t$), our method keeps attributes $(a_{i_{r,1}}, a_{i_{r,2}}, ..., a_{i_{r,h_r}})$ in the filtering relation FR_j of R_j, and attributes $(b_{i_{r,1}}, b_{i_{r,2}}, ..., b_{i_{r,h_r}})$ in the filtering relation FR_k of R_k. One index is built on $(a_{i_{r,1}}, a_{i_{r,2}}, ..., a_{i_{r,h_r}})$ and another index is built on $(b_{i_{r,1}}, b_{i_{r,2}}, ..., b_{i_{r,h_r}})$. Also, in checking filtering relations for irrelevant updates, our method considers the equi-join conditions on two base relations that are of disjunctive-conjunctive form (e.g., using index OR).

2.4.4 Filtering Out the Irrelevant Portion of an Update

In the basic algorithm, the entire update ΔR_i to base relation R_i ($1 \le i \le n$) is treated as an entity. That is, in *Operation* O_3, ΔR_i is first joined with the filtering relations FR_1, FR_2, …, FR_{i-1}, FR_{i+1}, FR_{i+2}, …, and FR_n. If the join result set is empty, we know that ΔR_i is irrelevant. Otherwise in *Operation* O_4, the entire ΔR_i is joined with the base relations R_1, R_2, …, R_{i-1}, R_{i+1}, R_{i+2}, …, and R_n.

In general, if ΔR_i contains multiple tuples, some tuples may be irrelevant while others may be relevant. In this case, treating the entire ΔR_i as an entity is too coarse. A better method is to treat each individual tuple in ΔR_i as an entity. In *Operation* O_3, the irrelevant tuples in ΔR_i are filtered out. Then the remaining tuples in ΔR_i are passed to *Operation* O_4.

The concrete method is as follows. Suppose ΔR_i contains q tuples t_i ($1 \le i \le q$). In *Operation* O_3, for each i ($1 \le i \le q$), the number i is appended as an additional attribute a_a to tuple t_i. When ΔR_i is joined with the filtering relations FR_1, FR_2, …, FR_{i-1}, FR_{i+1}, FR_{i+2}, …, and FR_n, a_a is never projected out. After we obtain the join result set S_j, if $S_j \ne \varnothing$, attribute a_a is extracted from S_j. Then after duplicate elimination, the values of a_a represent the remaining tuples in ΔR_i that need to be passed to *Operation* O_4.

2.4.5 Sharing a Filtering Relation among Multiple Join Views

Suppose multiple join views are built on the same base relation R. A simple method is to build multiple filtering relations of R, one for each join view. If those join views have non-overlapping selection conditions on R, then all the filtering relations of R contain different tuples and any update to a single tuple of R will affect at most one of these filtering relations. This is a good, common case in practice, where no redundancy exists among the filtering relations.

In certain other cases where some of the join views have overlapping selection conditions on R, redundancy may exist among these filtering relations and cause two problems. First, the

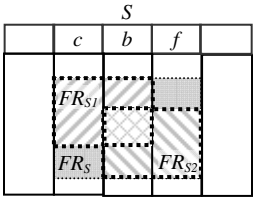
probability that the filtering relations are cached in memory is decreased. As a result, *Operation O_3* becomes more expensive. Second, when R is updated, updating all the filtering relations of R will be costly.

In this case, if possible, it may be better to let multiple join views share the same filtering relation of base relation R. For example, suppose join view JV_1 is defined as follows:

create materialized view JV_1 as select * from R_1, S, T_1
where $R_1.a$=$S.b$ and $S.c$=$T_1.d$ and C_1;

C_1 is a selection condition on $S.f$. Join view JV_2 is defined as follows:

create materialized view JV_2 as select * from R_2, S, T_2
where $R_2.e$=$S.b$ and $S.f$=$T_2.g$ and C_2;



C_2 is a selection condition on $S.c$. Then for base relation S, we may build only one filtering relation $FR_S = \pi_{b,c,f}(\sigma_{C_1 \vee C_2}(S))$ rather than two filtering relations $FR_{S1} = \pi_{b,c}(\sigma_{C_1}(S))$ and $FR_{S2} = \pi_{b,f}(\sigma_{C_2}(S))$. FR_S can be used for both JV_1 and JV_2. Whether FR_S is better than FR_{S1} and FR_{S2} depends on the overlapping degree of C_1 and C_2.

2.4.6 Selectively Skipping Operation O_3

As will be shown in Section 3 below, if either a small percentage of the update ΔR_i to base relation R_i is irrelevant, or ΔR_i is large enough so that hash/sort-merge join becomes the join method of choice for the join with some base relation R_j ($1 \leq j \leq n$, $j \neq i$), the content-based method may perform worse than the traditional content-independent method. In this case, *Operation O_3* can be skipped in the content-based method. This is equivalent to using the content-independent method plus updating the filtering relation FR_i accordingly. The analytical model described in Section 3 below will provide a means to determine the upper bound on the size of ΔR_i (or lower bound on the percentage of ΔR_i that is irrelevant) where performing *Operation O_3* is beneficial.

2.4.7 Using the Information about (Intermediate) Join Results in Operation O_3

Recall that in *Operation O_3*, ΔR_i is joined with the filtering relations FR_1, FR_2, …, FR_{i-1}, FR_{i+1}, FR_{i+2}, …, and FR_n. As a result, we know the (intermediate) join result sizes (e.g., during query execution, the techniques in [7] can be used to collect statistics about the output cardinalities of the operators). If these (intermediate) join result sizes are significantly different from the optimizer's original estimates, we know that the statistics in the database are imprecise.

Then in *Operation O_4*, when the remaining tuples in ΔR_i (after filtering) are joined with the base relations R_1, R_2, …, R_{i-1}, R_{i+1}, R_{i+2}, …, and R_n, the optimizer may use the information that is gained in *Operation O_3* to choose a better query plan.

For example, consider the join view mentioned in the introduction. Base relation R is updated by ΔR. Suppose the optimizer thinks that each tuple in ΔR has only a few matching tuples in base relation S. As a result, in *Operation O_4*, the optimizer chooses index nested loops as the join method for the join with S. However, from the information we gained in *Operation O_3*, we know that each tuple in ΔR has a large number of matching tuples in the filtering relation FR_S (and thus also a large number of matching tuples in S). Then in *Operation O_4*, our method may advise the optimizer to choose hash join as the join method for the join with S.

2.5 Discussions

In this section, we show that bloom filter and multi-attribute B-tree index are generally not suitable for our filtering purposes, as neither of them satisfies all four properties mentioned in Section 2.1. Consider the materialized view *MV* mentioned in the introduction.

Bloom filter [3] is an excellent technique to support membership queries in a set. However, it does not satisfy the association property. For example, when base relation R is updated, given an $S.b$ value, we cannot use a bloom filter to find the associated $S.c$ values.

We could create two multi-attribute B-tree indices on base relation S: I_1 for (b, c, f) and I_2 for (c, b, f). When R is updated, given a b value, performing an index-only scan on I_1 can find the corresponding c values with f="xyz". When T is updated, given a c value, performing an index-only scan on I_2 can find the corresponding b values with f="xyz". However, I_1 and I_2 do not satisfy the compactness property. The selection condition $S.f$="xyz" is not used to reduce their sizes. I_1 and I_2 both include attributes b, c, and f, whose representation could be fairly long, and contain duplicated information. Also, in general the selection condition on S could be arbitrarily complex and thus other attributes (in addition to f) may need to be included in I_1 and I_2.

3. PERFORMANCE EVALUATION

In this section, we evaluate the performance of our content-based method and the traditional content-independent method for detecting irrelevant updates to base relations of a materialized view. (Recall that the traditional content-independent detection method only considers the where clause condition in the materialized view definition.) We first build an analytical model to gain insight into the performance advantage of our content-based method vs. the traditional content-independent method in maintaining join views. Then we describe experimental results in our SRW prototype.

3.1 Analytical Model

Our model considers the effect that usually a significant portion of filtering relations is cached in memory. This is consistent with the approach recently proposed in [10], which considers the content in the buffer pool when computing the cost of a query plan. The goal of this model is not to accurately predict exact performance numbers in specific scenarios. Rather, it is to identify and explore some of the main factors that determine the effectiveness of the content-based method. Moreover, this analytical model provides the conditions for selectively skipping *Operation O_3* that is described in Section 2.4.6. In Section 3.3, we show that our model for the content-based and content-independent methods predicts trends fairly accurately where it overlaps with our experiments with a commercial RDBMS.

Consider the following join view *JV*:

create materialized view *JV* as select * from R, S
where $R.a$=$S.b$ and C_R and C_S;

C_R is the selection condition on base relation R. C_S is the selection condition on base relation S. The selectivity of C_S is q.

We make the following simplifying assumptions in this model:

(1) Base relation R (S) has an index I_R (I_S) on the join attribute. Both $R.a$ and $S.b$ are integer join attributes.

(2) $|\Delta R|$ tuples are inserted into R in a single transaction. These $|\Delta R|$ tuples are uniformly distributed on the join attribute, and all satisfy the selection condition C_R. The where clause condition in *JV*'s definition cannot be used to detect any irrelevant updates.

(3) The percentage of irrelevant updates in these $|\Delta R|$ tuples is p. That is, $p \times |\Delta R|$ tuples are irrelevant.

(4) For each tuple t_R, there are N matching tuples t_S in base relation S that satisfy $t_R.a=t_S.b$.

(5) The overhead of inserting a tuple into a filtering relation is a constant *INSERT*.

(6) In the content-independent method, the overhead of searching the index I_S once is a constant *SEARCH*. If N tuples t_S of base relation S are found to match a tuple t_R through index search, the overhead of fetching these N tuples t_S, applying the selection condition C_S, and then joining them with tuple t_R is (i) $N \times FETCH$, if index I_S is non-clustered or (ii) *FETCH*, if index I_S is clustered. (In the case of clustered index, we are assuming that all N tuples fit on a single page. The model could be easily extended to capture cases where t_R joins with more tuples than that can fit on a single page; however, this would not change the conclusions that we draw from our model.)

(7) In the content-based method, the overhead of searching the filtering relation $FR_S = \pi_b(\sigma_{C_S}(S))$ once is a constant *SEARCH*.

3.1.1 Total Workload

For each tuple t_R, we use as the cost metric the total workload *TW*, which is defined to be the total work done. This is a useful basic metric because we can derive other metrics, such as response time, from it easily (as shown in Section 3.1.2 below).

For both the content-independent method and the content-based method, the same updates must be performed on the base relations and on the join view. Because of this, our model omits the cost of these updates. Then the costs that must be captured are the extra update of the filtering relation $FR_R = \pi_a(\sigma_{C_R}(R))$ that is required by the content-based method, and the differences between the two methods in the cost of finding the join result tuples that need to be inserted into the join view. We now turn to quantify those costs, which we refer to as *TW*.

For the content-independent method, upon an insertion of a tuple t_R,

(a) Searching the index I_S once has overhead *SEARCH*.

(b) Fetching the N matching tuples t_S of base relation S, applying the selection condition C_S, and then joining them with tuple t_R has overhead (i) $N \times FETCH$, if index I_S is non-clustered or (ii) *FETCH*, if index I_S is clustered.

Thus for the content-independent method, the total workload *TW* for each tuple t_R is (i) $SEARCH+N \times FETCH$, if index I_S is non-clustered or (ii) *SEARCH+FETCH*, if index I_S is clustered.

For the content-based method, upon an insertion of a tuple t_R,

(a) Inserting tuple $\pi_a(t_R)$ into filtering relation FR_R has overhead *INSERT*.

(b) Searching filtering relation FR_S has overhead *SEARCH*.

(c) With probability $1-p$, tuple t_R is relevant. In this case, we need to further perform the procedure in the content-independent method.

So for the content-based method, the average total workload *TW* for each tuple t_R is (i) $INSERT+SEARCH+(1-p) \times (SEARCH+N \times FETCH)$, if index I_S is non-clustered or (ii) $INSERT+SEARCH+(1-p) \times (SEARCH+FETCH)$, if index I_S is clustered.

Compared to the content-independent method, the content-based method incurs an extra $INSERT+(1-p) \times SEARCH$, while saving (i) $p \times N \times FETCH$, if index I_S is non-clustered or (ii) $p \times FETCH$, if index I_S is clustered. As the percentage of irrelevant updates p grows, the savings in *FETCH* become significant compared to the extra overhead of $INSERT+(1-p) \times SEARCH$.

In general, for a large base relation T, the aggregate size of its filtering relation, its integer-attribute index, and the index on its filtering relation is a small percentage of the size of T. For example, a tuple of a base relation may be 200 bytes long while a tuple of a filtering relation may be only 4 (say, one 32-bit integer join attribute) or 8 (say, two 32-bit integer join attributes) bytes long. $4/200=2\%$ and $8/200=4\%$. Therefore, in a typical case, a large portion of filtering relations and indexes is cached in memory while large base relations are stored on disk. As a result, the time spent on *FETCH* is much larger than that spent on *INSERT* and *SEARCH*. In the following, we will assume that both *INSERT* and *SEARCH* take 0.01 I/O, and *FETCH* takes 1 I/O. (A page can contain a large number of tuples. Hence, the average logging overhead for inserting a tuple into a relation is a small percentage of one I/O.) Our conclusion would remain unchanged by small variations in these assumptions.

3.1.2 Response Time

The model in Section 3.1.1 is accurate only if for the join with base relation S, the join method is index nested loops, for which the cost is directly proportional to the number of tuples inserted. If $|\Delta R|$ is large enough, an algorithm such as sort-merge may perform better than index nested loops. To explore this issue, our model is extended to handle this case. We use sort-merge join as an alternative to index nested loops here; we believe our conclusions would be the same for hash joins. The point is that for both sort-merge and hash join, the join time is dominated by the time to scan a relation. Unless the number of modified tuples is a sizeable fraction of the base relations, the join time is independent of the number of modified tuples. (To keep our model simple, we assume that for the content-based method, we use the index nested loops join method for the join with filtering relation FR_S, as usually a significant portion of FR_S is cached in memory [10]. Again, the model could be easily extended to handle the sort-merge/hash join method for the join with FR_S. However, this would not change the conclusions that we draw from our model, as the cost of the join with FR_S is likely to be dominated by the cost of the join with S.)

Let $||x||$ denote the size of x in pages. Let M denote the size of available memory in pages. In addition, we make the following simplifying assumptions:

(1) The number of page I/Os is used to measure the performance. Then when the join method of choice is index nested loops, the total workload *TW* for each tuple t_R is (i) $0.01+N$ I/Os for the content-independent method when index I_S is non-clustered, (ii) 1.01 I/Os for the content-independent method when index I_S is clustered, (iii) $0.02+(1-p) \times (0.01+N)$ I/Os for the content-based method when index I_S is non-clustered, or (iv) $0.02+(1-p) \times 1.01$ I/Os for the content-based method when index I_S is clustered.

(2) ΔR can be held entirely in memory.

Given these assumptions, *TW* for the two methods for the multiple-tuple insertion is just $|\Delta R|$ times the *TW* for a single-tuple insertion. Calculating the response time is more interesting. We can express the response time (in number of I/Os) for either method by considering the work required by index nested loops join and sort-merge join.

For the content-independent method,

(a) If the join method of choice is sort-merge,
  (i) If index I_S is non-clustered, the sort-merge join time is dominated by the time of first scanning *S* and applying the selection condition C_S, and then sorting the remaining tuples of *S*. Therefore, the sort-merge join time is approximated by $\|S\|+\|S\|\times q\times log_M(\|S\|\times q)$ I/Os. Recall that *q* is the selectivity of C_S.
  (ii) If index I_S is clustered, the sort-merge join time is dominated by the time of scanning *S* and is approximated by $\|S\|$ I/Os.

(b) If index nested loops is the algorithm of choice, the index join time is approximated by $|\Delta R|\times(0.01+N)$ I/Os (if index I_S is non-clustered) or $|\Delta R|\times 1.01$ I/Os (if index I_S is clustered).

For the content-based method,

(a) If for the join with base relation *S*, the join method of choice is sort-merge, then
  (i) If index I_S is non-clustered, the sort-merge join time is approximated by $\|S\|+\|S\|\times q\times log_M(\|S\|\times q)$ I/Os. Thus, the response time of the content-based method is approximated by $|\Delta R|\times 0.02+\|S\|+\|S\|\times q\times log_M(\|S\|\times q)$ I/Os.
  (ii) If index I_S is clustered, the sort-merge join time is dominated by the time of scanning *S* and is approximated by $\|S\|$ I/Os. Hence, the response time of the content-based method is approximated by $|\Delta R|\times 0.02+\|S\|$ I/Os.

(b) If for the join with *S*, the join method of choice is the index join algorithm, then the response time of the content-based method is approximated by $|\Delta R|\times(0.02+(1-p)\times(0.01+N))$ I/Os (if index I_S is non-clustered) or $|\Delta R|\times(0.02+(1-p)\times 1.01)$ I/Os (if index I_S is clustered).

If $|\Delta R|$ is large enough that $\|S\|+\|S\|\times q\times log_M(\|S\|\times q)<|\Delta R|\times(1-p)\times(0.01+N)$ (if index I_S is non-clustered) and $\|S\|<|\Delta R|\times(1-p)\times 1.01$ (if index I_S is clustered) are satisfied, then the sort-merge join algorithm is preferable to index nested loops.

The above analysis shows that when sort-merge is the join algorithm of choice, the content-independent method outperforms the content-based method. This is because either method has the same join cost for the join with base relation *S* (sorting/scanning *S*), while the content-based method has the extra overhead of the updates to FR_R and the join with FR_S. Similarly, if most updates are relevant and *p* is close to 0, the content-independent method also outperforms the content-based method. (As mentioned in Section 2.4.6, in these two cases, it may be better for the content-based method to skip *Operation* O_3. It is straightforward to apply the analytical model and get an estimate of the conditions on when it would be better for the content-based method to skip *Operation* O_3.) In the discussion of the experiments with the analytical model below, we discuss the implications of these facts when choosing a method for join view maintenance.

3.2 Experiments with Analytical Model

Setting $\|S\|=200$, $M=50$ pages, $q=50\%$, $N=4$ (except in Figure 3), and $p=85\%$ (except in Figures 4~7), we present in Figures 3~8 the resulting performance of both the content-independent method and the content-based method.

Figure 3 shows the average *TW* for a single-tuple insert vs. the number of matching tuples *N*. Note that Figure 3 uses logarithmic scale for both the x-axis and the y-axis. If index I_S is clustered, *TW* is a constant 1.01 and 0.17 for the content-independent method and the content-based method, respectively. If index I_S is non-clustered, for both methods, *TW* increases linearly with the number of matching tuples *N*. In either case, when $N\geq 1$, *TW* for the content-based method is smaller than that for the content-independent method. This is because the content-based method can filter out a large percentage of irrelevant updates so that only a small percentage of updates need to be joined with base relation *S*. In the case that index I_S is non-clustered, the larger the *N*, the more significant the performance advantage of the content-based method (note that the y-axis is on logarithmic scale).

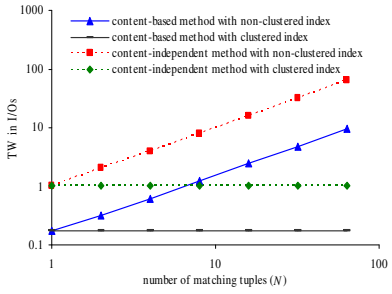


Figure 3. *TW* vs. number of matching tuples.

Figure 4 shows the average *TW* for a single-tuple insert vs. the percentage of irrelevant updates *p*. For the content-independent method, *TW* is a constant 4.01 (if index I_S is non-clustered) and 1.01 (if index I_S is clustered). For the content-based method, *TW* decreases linearly with the percentage of irrelevant updates *p*. When *p* is close to 0, the content-independent method slightly outperforms the content-based method. This is due to the insignificant "filtering" effect of the content-based method, and the extra overhead of the updates to FR_R and the join with FR_S. However, whether or not index I_S is clustered, once $p\geq 2\%$, *TW* for the content-based method becomes smaller than that for the content-independent method. The larger the *p*, the more irrelevant updates can be filtered out by the content-based method and thus the smaller the overhead of the index join with base relation *S*. Consequently, the larger the *p*, the more significant the performance advantage of the content-based method.

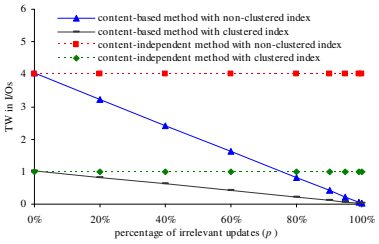


Figure 4. *TW* vs. percentage of irrelevant updates.

Figure 5 shows the execution time of one transaction with 40 inserted tuples, where for the join with base relation *S*, the join method of choice is the indexed nested loops join algorithm. The shapes of the curves in Figure 5 are similar to that in Figure 4, as the transaction execution time = $40\times TW$ *for a single tuple*.

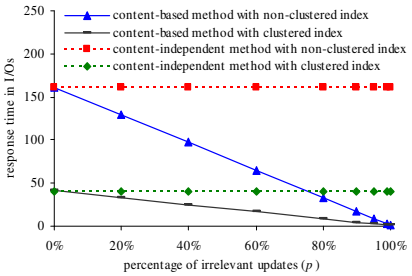


Figure 5. Execution time of one transaction with 40 tuples (index join).

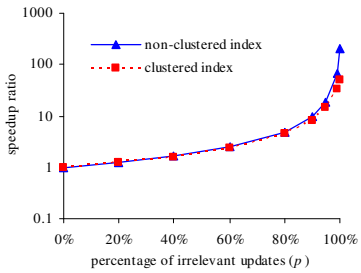


Figure 6. Speedup ratio gained by the content-based method.

Figure 6 shows the speedup ratio gained by the content-based method over the content-independent method for a transaction with 40 inserted tuples. Note that the y-axis uses logarithmic scale. When p is close to 0, the content-independent method slightly outperforms the content-based method and hence this speedup ratio is a little bit less than 1. However, once $p \geq 2\%$, the content-based method outperforms the content-independent method and thus this speedup ratio becomes greater than 1. Moreover, this speedup ratio increases rapidly with p. When index I_S is non-clustered, this speedup ratio is greater than that when index I_S is clustered.

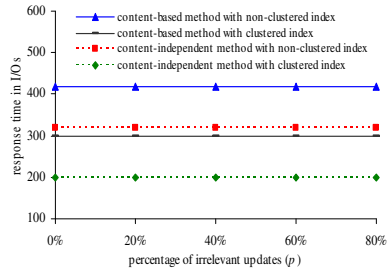


Figure 7. Execution time of one transaction with 5,000 tuples (sort-merge join).

Figure 7 shows the execution time of one transaction with 5,000 inserted tuples, where for the join with base relation S, the join method of choice is the sort-merge join algorithm. Whether index I_S is clustered or non-clustered, both the execution time of the content-independent method and that of the content-based method are a constant that is independent of the percentage of irrelevant updates p. Moreover, the constant for the content-based method is greater than that for the content-independent method. This is because the cost of the sort-merge join with S is the same for both methods, while the content-based method has the extra overhead of maintaining FR_R and the join with FR_S. For the content-based/content-independent method, the constant when index I_S is non-clustered is greater than that when index I_S is clustered.

Note that there is nothing special about the number 5,000 other than that the relevant portion of it (i.e., $(1\text{-}p)\times 5000$) is greater than the number of pages in S. This indicates that if the expected update transaction inserts a number of tuples, and the number of relevant tuples approximately equals to the number of pages in S, the content-independent method is the method of choice.

It is an interesting empirical question whether or not such large update transactions are likely. Anecdotal evidence suggests that they are not – data warehouses typically store data from several years of operation, so it seems highly unlikely that individual update transactions (of which there are presumably many each day) insert more than a very small fraction of the warehoused data.

Figure 8 shows the execution time of one transaction where the number of inserted tuples varies from 1 to 2,500. For the content-independent method, the execution time increases rapidly with the number of inserted tuples. For the content-based method, the execution time increases much more slowly. For the join with base relation S, the join time of both methods reaches a constant when the number of inserted tuples is large enough for the sort-merge join method to become the join method of choice. The content-based method reaches this point much later than the content-independent method. This is because of the "filtering" effect of the content-based method: after filtering out irrelevant updates using FR_S, fewer tuples are joined with S. However, once again, as the number of relevant inserted tuples approaches the number of pages of S, the content-based method is indeed worse than the content-independent method.

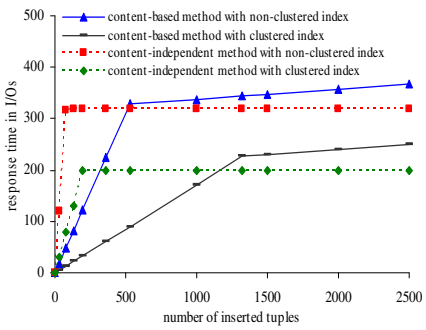


Figure 8. Execution time vs. number of inserted tuples.

It is straightforward to apply the above analytical model to the situation of a join view on multiple base relations. Experiments with this model did not provide any insight not already given by the two-relation model, so we omit them here.

3.3 Evaluation of the Content-based Detection Method in Our SRW prototype

We now turn to describe experiments we performed in our SRW prototype using IBM DB2 Version 8.2. Our measurements were performed on a computer with two 3GHz processors, 2GB main memory, one 111GB disk. The buffer pool size of DB2 was set to be 1.2GB.

The three relations used for the experiments followed the schema of the standard TPC-R Benchmark relations [14]:

customer (custkey, acctbal, …),
orders (orderkey, custkey, orderpriority, shippriority, …),
lineitem (orderkey, …).

Table 1. Test data set.

	number of tuples	total size
customer	8M	2.6GB
orders	16M	4.16GB
lineitem	160M	25.2GB

We wanted to test the performance of insertion into the *lineitem* relation in the presence of join views. Two join views were chosen for testing:

(1) JV_1 records the lineitem information of certain orders:
create materialized view JV_1 as select *
from lineitem l, orders o
where l.orderkey=o.orderkey and C_1;
Here, C_1 is the selection condition on the *orders* relation that is of the form $o.orderpriority=p_o$ and $o.shippriority=p_s$.

(2) JV_2 records the lineitem information of certain orders that are bought by certain customers:
create materialized view JV_2 as select *
from lineitem l, orders o, customer c
where l.orderkey=o.orderkey and o.custkey=c.custkey and C_1 and C_2;
Here, C_2 is the selection condition on the *customer* relation that is of the form $c.acctbal<b_a$.

We repeated our experiments with other kinds of join views/workloads. The results were similar and thus omitted.

The join view maintenance consists of three steps: updating the base relation, computing the changes to the join view, and

updating the join view. As the first step and the third step were the same for both the content-independent method and the content-based method, we only measured the time spent on the second step.

Because DB2 does not currently support the content-based detection method, a query rewriting approach was used to resolve this problem. We evaluated the performance of join view maintenance when 400 tuples were inserted into the *lineitem* relation (these tuples each has one matching tuple in the *orders* relation) in the following way:

(1) For both the *orders* relation and the *customer* relation, we created a non-clustered index on each selection/join attribute.

(2) We created a new relation *delta_lineitem* that had the same schema as *lineitem*.

(3) 400 tuples were inserted into *delta_lineitem*.

(4) For join view JV_1, we created relation $orders\_FR_1=\pi_{orderkey}(\sigma_{C_1}(orders))$ as the filtering relation for *orders*. For join view JV_2, we created two relations $orders\_FR_2=\pi_{orderkey,\ custkey}(\sigma_{C_1}(orders))$ and $customer\_FR=\pi_{custkey}(\sigma_{C_2}(customer))$ as filtering relations for *orders* and *customer*, respectively. We created a first non-clustered index on the *orderkey* attribute of the $orders\_FR_1$ relation, a second non-clustered index on the *orderkey* attribute of the $orders\_FR_2$ relation, and a third non-clustered index on the *custkey* attribute of the *customer_FR* relation. No filtering relation was created for the *lineitem* relation, as *lineitem* is the most frequently updated base relation in the database.

(5) The execution time of the following two SQL statements was measured:

Q_1: select * from delta_lineitem l, orders o
where l.orderkey=o.orderkey and C_1;

Q_2: select * from delta_lineitem l, orders o, customer c
where l.orderkey=o.orderkey and o.custkey=c.custkey
and C_1 and C_2;

These two SQL statements implemented the content-independent method for join views JV_1 and JV_2, respectively, while 400 tuples were inserted into the *lineitem* relation.

(6) We created a temporary relation *tmp_lineitem* that had the same schema as *lineitem*. Initially, *tmp_lineitem* is empty. To implement the content-based method for join view JV_1, Q_1 was replaced with the following two SQL statements:

Q_3: insert into tmp_lineitem select distinct(l.*)
from delta_lineitem l, $orders\_FR_1$ o
where l.orderkey=o.orderkey;

Q_4: select * from tmp_lineitem l, orders o
where l.orderkey=o.orderkey and C_1;

To implement the content-based method for join view JV_2, Q_2 was replaced with the following two SQL statements:

Q_5: insert into tmp_lineitem select distinct(l.*)
from delta_lineitem l, $orders\_FR_2$ o, customer_FR c
where l.orderkey=o.orderkey and o.custkey=c.custkey;

Q_6: select * from tmp_lineitem l, orders o, customer c
where l.orderkey=o.orderkey and o.custkey=c.custkey
and C_1 and C_2;

(7) For join view JV_1, we varied the selectivity s_1 of the selection condition C_1 from 1% to 100%. The percentage of irrelevant updates to the *lineitem* relation is $1\text{-}s_1$, which is equal to the filtering ratio of the content-based method. The ratio of the size of $orders\_FR_1$ to that of *orders* is $s_1 \times 4/251$, where 4B is the size of the *orderkey* attribute, and 251B is the average tuple size of *orders*.

(8) For join view JV_2, the selectivity of the selection condition C_1 was fixed as 60%. We varied the selectivity s_2 of the selection condition C_2 from 1% to 100%. There is no correlation between C_1 and C_2. Hence, the percentage of irrelevant updates to the *lineitem* relation is $1\text{-}60\% \times s_2$, which is equal to the filtering ratio of the content-based method. The ratio of the size of $orders\_FR_2$ to that of *orders* is $60\% \times 8/251 = 0.019$. The ratio of the size of *customer_FR* to that of *customer* is $s_2 \times 4/325$, where 325B is the average tuple size of *customer*.

(9) Before we tested the content-independent method, we first kept inserting tuples into the *lineitem* relation and using the content-independent method to maintain the join view JV_1/JV_2 until the system became stable. Then the experiment was run once. The same method was used to test the performance of the content-based method.

(10) For each experiment, the reported number is averaged over a large number of runs.

Figure 9 shows the join view maintenance time of JV_1 that is predicted by the analytical model. All the numbers in Figure 9 are scaled by a constant factor (the time unit is 80 I/Os) so only the relative ratios between them are meaningful. Figure 10 shows the experimental join view maintenance time of JV_1. Figure 9 and Figure 10 match well. The speedup gained by the content-based method over the content-independent method increases with the percentage of irrelevant updates to the *lineitem* relation.

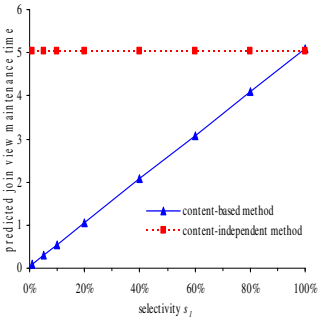


Figure 9. Predicted join view maintenance time (JV_1).

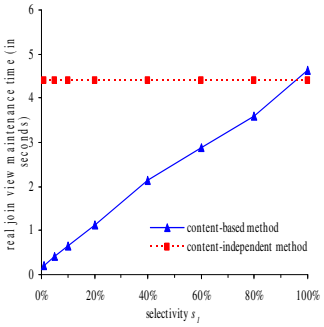


Figure 10. Real join view maintenance time (JV_1).

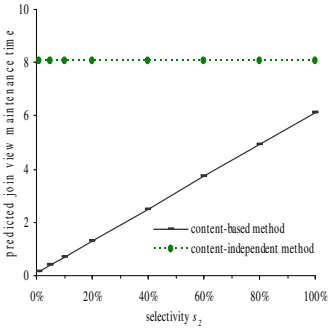


Figure 11. Predicted join view maintenance time (JV_2).

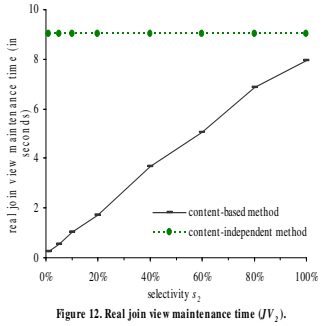


Figure 12. Real join view maintenance time (JV_2).

For join view JV_2, Figure 11 shows the maintenance time that is predicted by the analytical model (the time unit is 80 I/Os), and Figure 12 shows the experimental maintenance time. Figure 11 and Figure 12 match well.

We also ran experiments with large update transactions, where our analytical model predicts that the content-independent method will perform better than the content-based method when sort-

merge/hash becomes the join method for the join with base relation *S*. We did indeed observe the trend that as the transaction size increased, the performance of the content-independent method first approached and then exceeded that of the content-based method. However, the analytical model was less accurate for large updates than for small. This is because our cost model for sort-merge join is too simple (e.g., it does not take special consideration of the portion of base relations that is cached in memory). Also, it is difficult to estimate precisely the size of available memory for the sort-merge/hash join. For these reasons the large update results are not presented here.

The difficulty of duplicating in DB2 the analytical model results for large updates does not affect our conclusions. The model is accurate for reasonably sized updates; these are the ones that are common in practice and also are the ones for which the content-based method dramatically outperforms the content-independent method.

4. RELATED WORK

Semi-joins (and bloom-joins) in distributed RDBMSs [1] use filtering to reduce communication overhead. That filtering method is different from our filtering method:

(1) In our method, the updated tuples of a base relation, Δ, are joined with the filtering relations of the other base relations. There is one filtering target (Δ) and multiple summary data structures (filtering relations). In comparison, semi-join joins the projection of a relation, *P*, with other relations. There are multiple filtering targets (other relations) and one summary data structure (*P*).

(2) Our method mainly performs inexpensive in-memory operations, as filtering relations are likely to be cached in memory. In contrast, semi-join performs a large number of expensive I/Os, as base relations are stored on disk.

(3) Some of our techniques either do not apply to semi-joins because they are used in different contexts (materialized view maintenance vs. distributed query processing), or have not been used in semi-joins before. For example, for the latter part, semi-joins neither use hashing to reduce the sizes of join attributes, nor stop processing after finding the first matching tuple in certain relations. (Bloom-joins use bloom filters. As mentioned in Section 2.5, bloom filter does not satisfy the association property.)

(4) Filtering relations need to be maintained in the presence of updates to the base relations of the join view. In contrast, there is no need to maintain the data structure used in semi-joins.

[15] proposed using join indices to speed up join query processing. A join index links the row ids of matching tuples in multiple base relations. In our content-based detection method, a filtering relation links multiple join attributes (if any) of a single base relation.

If the hashing-based compression method is not considered, a filtering relation is simply a selection and projection of a base relation. [9] and [6] proposed using auxiliary relations to speed up join view maintenance in distributed data warehouses and parallel RDBMSs, respectively. An auxiliary relation is also a selection and projection of a base relation. However, auxiliary relations are larger than filtering relations, as auxiliary relations keep both join attributes and non-join attributes while filtering relations only keep join attributes. Also, we can use auxiliary relations to compute the update to the join view. In contrast, we can only use filtering relations to tell whether or not the update to the join view is empty.

[12] use partial indices to reduce index maintenance overhead. Filtering relations can be regarded as another kind of indices – given a join attribute value, we can use the filtering relation to find other join attribute values.

[11] proposed maintaining additional materialized views in order to reduce the total maintenance cost of a target materialized view. Our filtering relations can be regarded as additional "mini" materialized views. However,

(1) Similar to the case of auxiliary relations, we can use the additional materialized views in [11] to compute the update to the target materialized view, while we can only use filtering relations to tell whether or not the update to the join view is empty. Filtering relations themselves cannot be used to compute the update to the target materialized view.

(2) The additional materialized views in [11] are large and reside on disk. They do not satisfy the compactness property (see Section 2.1). Reading/maintaining the additional materialized views will cause a large number of expensive I/Os, while reading/maintaining filtering relations basically only requires cheap in-memory operations (with some minor logging overhead during maintenance).

5. REFERENCES

[1] P.A. Bernstein, D.W. Chiu. Using Semi-Joins to Solve Relational Queries. JACM 28(1): 25-40, 1981.

[2] J.A. Blakeley, N. Coburn, and P. Larson. Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates. TODS 14(3): 369-400, 1989.

[3] B.H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. CACM 13(7): 422-426, 1970.

[4] J.A. Blakeley, P. Larson, and F.W. Tompa. Efficiently Updating Materialized Views. SIGMOD Conf. 1986: 61-71.

[5] S. Chandrasekaran, M.J. Franklin. Streaming Queries over Streaming Data. VLDB 2002: 203-214.

[6] G. Luo, J.F. Naughton, and C.J. Ellmann et al. A Comparison of Three Methods for Join View Maintenance in Parallel RDBMS. ICDE 2003: 177-188.

[7] G. Luo, J.F. Naughton, and C.J. Ellmann et al. Toward a Progress Indicator for Database Queries. SIGMOD Conf. 2004: 791-802.

[8] G. Luo, J.F. Naughton, and C.J. Ellmann et al. Transaction Reordering and Grouping for Continuous Data Loading. BIRTE 2006.

[9] D. Quass, A. Gupta, and I.S. Mumick et al. Making Views Self-Maintainable for Data Warehousing. PDIS 1996: 158-169.

[10] R. Ramamurthy, D.J. Dewitt. Buffer-pool Aware Query Optimization. CIDR 2005.

[11] K.A. Ross, D. Srivastava, and S. Sudarshan. Materialized View Maintenance and Integrity Constraint Checking: Trading Space for Time. SIGMOD Conf. 1996: 447-458.

[12] M. Stonebraker: The Case for Partial Indexes. SIGMOD Record 18(4): 4-11, 1989.

[13] M. Stonebraker. Stream Applications. telegraph.cs.berkeley.edu/swim/stonebraker-swim-app.ppt, 2003.

[14] TPC Homepage. TPC-R benchmark, www.tpc.org.

[15] P. Valduriez. Join Indices. TODS 12(2): 218-246, 1987.

[16] K. Wu, P.S. Yu, and B. Gedik et al. Challenges and Experience in Prototyping a Multi-Modal Stream Analytic and Monitoring Application on System S. VLDB 2007: 1185-1196.