Efficient Refreshment of Data Warehouse Views*

Lars Bækgaard† Nick Roussopoulos

Institute for Advanced Computer Studies
and
Department of Computer Science
University of Maryland at College Park
A.V. Williams Building, College Park, MD 20742
Phone: 301-405-2687. Fax: 301-405-6707, Email: nick@cs.umd.edu

Abstract: A data warehouse is a view on a set of distributed and possible loosely coupled source databases. For efficiency reasons a warehouse should be maintained as a materialized view. Therefore, efficient incremental algorithms must be used to periodically refresh the data warehouse. It is possible and desirable to separate the process of warehouse refreshment from the process of warehouse use. In this paper we describe and compare view refreshment algorithms that are based on different combinations of materialized views, partially materialized views, and pointers. Our contribution is twofold. First, our algorithms and data structures are designed to minimize network communication and interactions between the warehouse and the source databases. The minimal set of data that is necessary for both warehouse refreshment and warehouse use is stored on the warehouse. Second, we describe the results of an experiment comparing these methods with respect to storage overhead and I/O. Briefly, the experiment show that algorithms based on a combination of partially materialized views and pointers outperforms algorithms based on materialized views.

Key Terms: Incremental view refreshment, data warehousing.

1 Introduction

A data warehouse can be used as an integrated and uniform basis for decision support, data mining, data analysis, and ad-hoc querying across the source databases. A data warehouse is likely to be a growing collection of possible time-stamped raw data. In some cases a data warehouse will be an append-only database. For example, a department store chain may want to support central access to all sales transactions. In some cases a data warehouse will be a temporally constrained database. For example, the department store chain may want to limit the access to sales transactions within the last year.

*This work was supported in part by The Danish Natural Science Research Council, Project No. 9400911 and Project No. 9502074, by the National Science Foundation under grants IRI-9057573 and GDR-85-00108, by NASA/USRA under contracts 5555-09 and NAG 5-2926, by ARPA under contract 003195, by the Institute of Systems Research, and by the University of Maryland Institute for Advanced Computer Studies (UMIACS).

†Department of Mathematics and Computer Science, Aalborg University, Fr. Bajers Vej 7, DK-9220 Aalborg Ø, Denmark. Phone: +45 98 15 85 22. Fax: +45 98 15 81 29. Email: larsb@iesd.auc.dk. The research was done while Lars Bækgaard was visiting University of Maryland

View refreshment is an important issue in distributed database environments where efficient access to a set of source databases is typically supported by replicating views on remote sites. However, it is a central issue in data warehousing environments where views defined on physically separated, heterogeneous, and autonomous databases must be supported [ZGMHW95]. Changes in the source databases must be transformed into a data model format used by the warehouse and integrated into the warehouse.

A data warehouse does not necessarily have to be completely up to date. The reason is that the warehouse is used for analytical, managerial data processing rather than for day-by-day operational data processing. Inmon has even recommended that a data warehouse is refreshed with a 24-hour delay in order to ensure that warehouse data are not confused with operational data [Inm93]. Consequently, it makes sense to assume that warehouse use and warehouse refreshment are separate processes. We believe that such separation makes it easier to answer warehouse queries efficiently and to solve the problems of warehouse consistency [ZGMHW95].

In this paper we address some of the problems of efficient incremental refreshment of warehouse views. We focus on the following questions. *What data should be stored at the warehouse in order to support efficient view refreshment?* These algorithms should be able to process sets of updates received from one or more server databases without requesting additional information from other sources and, therefore, being independent of the availability of the network and the other database systems. Network communication occurs only when a source database signals relevant changes to a warehouse. The warehouse should be able to process the changes without querying source databases. Otherwise, warehouse refreshment may become too dependent on the activity, the availability, and efficiency of the source databases. This *no talk back* requirement implies that the additional information needed for view refreshment must be present at the warehouse.

What data structures and algorithms should be used for view organization and refreshment? Incremental view refreshment should be performed with as little access to raw data as possible. The raw data objects may be very large compared to the attributes that are used in selection and join predicates. Indexing and pointer caching can be used to reduce or avoid raw data access during view refreshment. A combination of three basic methods can be used to support data warehouse views. A computed view is stored as a view definition, ie., as a query expression. A computed view is reconstructed each time it is accessed by methods like query modification [Sto75]. A pointer-based ViewCache contains a set of tuple identifiers that identify and point to the data objects in the underlying relations that be dereferenced to materialize the view [Rou91]. A materialized view is stored as the set of data objects that belong to the view [Han87]. ViewCaches and materialized views must be maintained incrementally.

We present and analyze five refreshment algorithms that are based on various combinations of materialized views, partially materialized views, and ViewCaches. Existing data warehousing approaches focus solely on materialized views. We present the results of an experiment that clearly shows indicates that refreshment algorithms based on a combination of materialized views, partially materialized views, and ViewCaches outperform algorithms based solely on materialized views. We have assumed that all involved databases are relational databases.

Much research work has focussed on incremental view refreshment. Incremental formulas has been defined for views defined by SPJ-queries [BLT86], [LHM+86], [Rou91], for views defined by relational complete query languages [GMS93], [QW91], for views defined by time-varying selections [BM95c], and for view defined by set differences [BM95b]. View indexing by means of pointer structures has been described in [Bæk95], [BM95a], [BM95b], [BM95c], [JMR91], [QW91], [Rou82a], [Rou82b], [Rou91], [RK91], [Val87]. The use and refreshment of materialized views has been described in [BM90], [CKPS95], [GL95], [GMR95], [HK95], [Han87], [LMSS95], [SJGP90], [SR87], [ZGMHW95]. Incremental techniques has been described for active databases [CSL+90], [CW91], [HCKW90], [Han92], [RCBB89], [WDSY91], for deductive databases [GMS93], and for temporal databases [BM95c], [JMR91], [McK88]. However, we are not aware of existing incremental algorithms that combine materialized view, partially materialized views, and ViewCaches.

In Section 2 we discuss a set of methods that can be used for view organization and refreshment in data warehousing environments. In Section 3 we present two incremental refreshment algorithms for materialized join views and in Section 4 we present two incremental refreshment algorithms for materialized join views. These algorithms exemplifies the conceptual discussion in Section 2. Two of the algorithms are significant more efficient than existing incremental refreshment algorithms in many relevant situations. In Section 5 we use a set of experimental results to evaluate the I/O-efficiency of the incremental view refreshment algorithms. In Section 6 we conclude the paper and suggest directions for future research.

2 Incremental View Refreshment

Our examples are based on two base relations, R and S, with schemata $R(a, b, c)$ and $S(a, b, c)$. We use the notation $R^{mm.dd.yy}$ to denote the relation R as of time mm.dd.yy (month, day, year). The symbol R denotes R as of current time. We assume that all tuples are augmented with immutable and globally unique tuple identifiers. Consequently, each tuple is uniquely identified even though duplicates may occur.

Changes made to base relations are stored on differential files [SL76] from which insertion sets and deletion sets are extracted. The symbol $R_I^{t_1-t_2}$ denotes the insertion set of R relative to the time interval $(t_1, t_2]$. The symbol $R_D^{t_1-t_2}$ denotes the deletion set of R relative to the time interval $(t_1, t_2]$. We assume that insertion and deletion sets are minimal in the sense that $R_I^{t_1-t_2} \cap R^{t_1} = \emptyset$ and $R_D^{t_1-t_2} \subseteq R^{t_1}$.

Figure 1 illustrates the contents of $R^{11.15.95}$, $R_I^{11.15.95-11.17.95}$, $R^{11.17.95}$, $S^{11.15.95}$, $S_I^{11.15.95-11.17.95}$, and $S^{11.17.95}$.

Our discussions of join views in the remaining part of the paper are based on the view definition $V = R \bowtie_{[R.a=S.a]} S$. Figure 2 illustrates the contents of $V^{11.15.95}$ and $V^{11.17.95}$. $R.a$, $R.b$, $R.c$, $S.a$, $S.b$, $S.c$, are called *view attributes* because they occur in defined view. $R.a$ and $S.a$ are called *predicate attributes* because they occur in the join predicate. In general, the set of predicate attributes may or may not be a subset of the set of view attributes. We assume that a materialized view contains all relevant tuple identifiers, view attributes, and predicate attributes.

We use the terms $R1$ and $R0$ to denote the sets of joining and non-joining tuples, respectively, in R. Figure 3 illustrates the contents of $R0$ and $S0$.

$$\begin{aligned} R1 &= \Pi_{R.a,R.b,.R.c}(R \bowtie S) \\ R0 &= R \setminus R1 \end{aligned}$$

The following formulas define the incremental refreshment of join views [BLT86], [Rou91].

$$(R \bowtie S)^{new} = ((R \bowtie S)^{old} \setminus ((R \bowtie S)_D)) \cup ((R \bowtie S)_I) \tag{1}$$

$R^{11.15.95}$

tid	a	b	c
r_1	1	10	100
r_2	2	11	101
r_3	3	12	102
r_4	4	13	103

$R_I^{11.15.95-11.17.95}$

tid	a	b	c
r_5	5	14	104
r_6	7	15	105

$R^{11.17.95}$

tid	a	b	c
r_1	1	10	100
r_2	2	11	101
r_3	3	12	102
r_4	4	13	103
r_5	5	14	104
r_6	7	15	105

$S^{11.15.95}$

tid	a	b	c
s_1	1	20	200
s_2	2	21	201
s_3	5	22	202
s_4	6	23	203

$S_I^{11.15.95-11.17.95}$

tid	a	b	c
s_5	3	24	204
s_6	8	25	205

$S^{11.17.95}$

tid	a	b	c
s_1	1	20	200
s_2	2	21	201
s_3	5	22	202
s_4	6	23	203
s_5	3	24	204
s_6	8	25	205

Figure 1: Base relations and insertion sets

$$(R \bowtie S)_D = (R1^{old} \bowtie S_D) \cup (R_D \bowtie S1^{old}) \tag{2}$$
$$(R \bowtie S)_I = (R^{new} \bowtie S_I) \cup (R_I \bowtie S^{new}) \tag{3}$$

Formula (2) shows that the processing of R_D and S_D can be based on the data that is contained in the view because $R1$ and $S1$ are contained in the view. Depending on the view representation method different access paths like indices may be needed for efficiency reasons but no extra data from R and S is needed.

Formula (3), however, shows that non-view data is needed for the processing of R_I and S_I. More specifically, R_I must be joined with both the joining tuples and the non-joining tuples of S. Likewise, S_I must be joined with both the joining tuples and the non-joining tuples of R.

If V is stored as a materialized join view as illustrated in Figure 2 access to the predicate attributes, the view attributes, and the tuple identifiers of $R0^{new}$, $R1^{new}$, $S0^{new}$, and $S1^{new}$ is necessary. Note, that the predicate attributes, the view attributes, and the tuple identifiers of $R1^{old}$ and $S1^{old}$ are stored on the materialized view. $R0$ and $S0$ are not contained in the materialized view and must be stored and accessed elsewhere.

If V is stored as a join ViewCache as illustrated in Figure 4, access to the view attributes of $R1$ and $S1$ is necessary for view materialization. Briefly, a join ViewCache contains a pair of tuple identifiers for each joining tuple pair [Rou91]. Access to the predicate attributes and the tuple identifiers of $R0^{new}$, $R1^{new}$, $S0^{new}$, and $S1^{new}$ is necessary for the propagation. The predicate attributes are not contained in the ViewCache and must be stored and accessed elsewhere.

3 Refreshment Algorithms for Materialized Join Views

In this section we present two incremental algorithms for the refreshment of materialized join views. Both algorithms are based on data structures that store all data that is needed for view refreshment

$V_{mat}^{11.15.95}$

$R.a$	$R.b$	$R.c$	$S.a$	$S.b$	$S.c$
1	10	100	1	20	200
2	11	101	2	21	201

$V_{mat}^{11.17.95}$

$R.a$	$R.b$	$R.c$	$S.a$	$S.b$	$S.c$
1	10	100	1	20	200
2	11	101	2	21	201
3	12	102	3	24	204
5	14	104	5	22	202

$V_{matout}^{11.15.95}$

$R.a$	$R.b$	$R.c$	$S.a$	$S.b$	$S.c$
1	10	100	1	20	200
2	11	101	2	21	201
3	12	102	-	-	-
4	13	103	-	-	-
-	-	-	5	22	202
-	-	-	6	23	203

$V_{matout}^{11.17.95}$

$R.a$	$R.b$	$R.c$	$S.a$	$S.b$	$S.c$
1	10	100	1	20	200
2	11	101	2	21	201
3	12	102	3	24	204
5	14	104	5	22	202
4	13	103	-	-	-
7	15	105	-	-	-
-	-	-	6	23	203
-	-	-	8	25	205

Figure 2: Materialized join views and outer join views

$R0^{11.15.95}$

a	b	c
3	12	102
4	13	103

$R0^{11.17.95}$

a	b	c
4	13	103
7	15	105

$S0^{11.15.95}$

a	b	c
5	22	202
6	23	203

$S0^{11.17.95}$

a	b	c
6	23	203
8	25	205

Figure 3: Non-joining tuples

at the warehouse.

3.1 Materialized Join View

The algorithm *MatView* uses R_I and S_I to generate $(R \bowtie S)_I$. After removal of duplicates $(R \bowtie S)_I$ is added to the materialized join view, V_{mat}, and $R0$ and $S0$ are updated. *MatView* is not designed to handle deletions from R and S, ie., R_D and S_D are assumed to be empty.

The following data is stored at the warehouse: A materialized join view, V_{mat}, and two sets of non-joining tuples, $R0$ and $S0$.

As indicated by Equations (3)-(5) the propagation of R_I into $(R \bowtie S)_I$ can be processed as the union of five joins as defined in Equation (6).

$$R^{new} = ((R0^{old} \cup R1^{old}) \setminus R_D) \cup R_I \tag{4}$$

$$S^{new} = ((S0^{old} \cup S1^{old}) \setminus S_D) \cup S_I \tag{5}$$

$$\begin{aligned} (R \bowtie S)_I = \; & (R_I \bowtie (S0^{old} \setminus S_D) \cup \\ & (R_I \bowtie (S1^{old} \setminus S_D)) \cup \end{aligned}$$

$V_{ptr}^{11.15.95}$	
$R.tid$	$S.tid$
r_1	s_1
r_2	s_2

$V_{ptr}^{11.17.95}$	
$R.tid$	$S.tid$
r_1	s_1
r_2	s_2
r_3	s_5
r_5	s_3

Figure 4: Join ViewCaches

$$((R0^{old} \setminus R_D) \bowtie S_I) \cup$$
$$((R1^{old} \setminus R_D) \bowtie S_I) \cup$$
$$(R_I \bowtie S_I) \tag{6}$$

The joins in Steps 1-5 are based on Equation (6). Since $R1$ and $S1$ per definition are contained in V_{mat} the joins $R_I \bowtie S1$ and $R1 \bowtie S_I$ are replaced by the joins $R_I \bowtie V_{mat}$ and $V_{mat} \bowtie S_I$, respectively.

MatView. Materialized Join View.
Establish the following duplicate free tuple sets during Steps 1-5:
JP: Tuple identifier pairs for new joining tuple pairs.
NRI: Tuple identifiers for non-joining R_I tuples.
NSI: Tuple identifiers for non-joining R_I tuples.
$JR0$: Tuple identifiers for joining $R0$ tuples.
$JS0$: Tuple identifiers for joining $S0$ tuples.
1. Compute $R_I \bowtie S_I$.
2. Compute $V_{mat} \bowtie S_I$.
3. Compute $R_I \bowtie V_{mat}$.
4. Compute $R0 \bowtie S_I$.
5. Compute $R_I \bowtie S0$.
6. Insert JP into V_{mat}.
7. Insert NRI into $R0$.
8. Insert NSI into $S0$.
9. Delete $JR0$ from $R0$.
10. Delete $JS0$ from $S0$.

The variable JP is used for the construction of $(R \bowtie S)_I$. The variables NRI, NSI, $JR0$, and $JS0$ are used to construct the modifications that must be made to the sets of non-joining tuples, $R0$ and $S0$.

3.2 Materialized OuterJoin View

The algorithm *MatOutView* uses R_I and S_I to generate $(R \bowtie S)_I$. After removal of duplicates $(R \bowtie S)_I$ is added to the materialized outer join view, V_{matout}. *MatOutView* is not designed to handle deletions from R and S, ie., R_D and S_D are assumed to be empty.

The following data is stored at the warehouse: A materialized outer join view, V_{matout}.

The joins in Steps 1-3 are based on Equation (6). Since $R0$, $R1$, $S0$, and $S1$ are contained in V_{matout} the joins $R_I \bowtie S0$, $R_I \bowtie S1$, $R0 \bowtie S_I$, and $R1 \bowtie S_I$ are replaced by the joins $R_I \bowtie V_{matout}$ and $V_{matout} \bowtie S_I$.

MatOutView. Materialized OuterJoin View.
Establish the following duplicate free tuple sets during Steps 1-3:
- JP: Tuple identifier pairs for new joining tuple pairs.
- NRI: Tuple identifiers for non-joining R_I tuples.
- NSI: Tuple identifiers for non-joining R_I tuples.
- $JR0$: Tuple identifiers for joining $R0$ tuples.
- $JS0$: Tuple identifiers for joining $S0$ tuples.

1. Compute $R_I \bowtie S_I$.
2. Compute $V_{matout} \bowtie S_I$.
3. Compute $R_I \bowtie V_{matout}$.
4. Insert JP into V_{mat}.
5. Insert $(NRI, -)$ into V_{mat}.
6. Insert $(-, NSI)$ into V_{mat}.
7. Delete $(JR0, -)$ from V_{mat}.
8. Delete $(-, JS0)$ from V_{mat}.

The variable JP is used for the construction of $(R \bowtie S)_I$. The variables NRI, NSI, $JR0$, and $JS0$ are used to construct the modifications that must be made to the sets of non-joining tuples in V_{matout}. The notation $(NRI, -)$ symbolizes the combination of NRI tuple with null values for the attributes of S.

4 Refreshment Algorithms for ViewCaches

In this section we present three incremental algorithms for the refreshment of data warehouse join views that are stored as ViewCaches. All three algorithms are based on data structures that store all data that is needed for view refreshment at the warehouse.

4.1 Join ViewCache

The algorithm *ViewCache* [Rou91] is an incremental algorithm that maintains and materializes a join ViewCache for a 2-way join. Figure 4 illustrates the contents of join ViewCaches. *ViewCache* is designed to handle both insertions to and deletions from R and S.

The following data is stored at the warehouse: A join ViewCache, $(R \bowtie S)_{ptr}$, and copies of the join operands, R and S.

ViewCache. Join ViewCache.
1. Compute $R \bowtie S_I$.
2. Compute $R_I \bowtie S$.
3. FOR each $(R \bowtie S)_{ptr}$ page, p, DO
  3.1. Delete $(R_D, -)$ and $(-, S_D)$.
  3.2. Insert joining pairs from Steps 1.1 and 2.1. Avoid duplicates.
  3.3. Rewrite and, optionally, materialize p.

ViewCache works as follows. In Step 1 R^{new} is joined with S_I and in Step 2 S^{new} is joined with R_I. Thus, *ViewCache* uses R^{new} and S^{new} to implement the joins in Equation (3). The tuple identifier pairs that represent the joining tuple pairs are saved and used for the cache update in Step 3 where the ViewCache is scanned. For each cache page R_D and S_D are used to identify the tuple pairs that must be deleted from the current cache page (Step 3.1). In Step 3.2 the relevant subset of the saved tuple identifier pairs from Steps 1-2 are inserted to the current cache page (Step 3.2). In Step 3.3 the updated current cache page is rewritten as a ViewCache page and/or as a materialized view one.

The dominant factor of the cost savings obtained by the ViewCache storage organization that guarantees spatial clustering of the pointers and its incremental refresh is attributed to its *one pass algorithm* which performs steps 1,2 and 3 simultaneously.

4.2 Partially Materialized Join ViewCache

The incremental algorithm *PartMatJoin* is based on partially materialized views for $R0$ and $S0$ and a partially materialized view, $V_{ptr(a)}$. *PartMatJoin* is not designed to handle deletions from R and S, ie., R_D and S_D are assumed to be empty.

Figure 5 illustrates the partially materialized views that correspond to Figure 3. The partially materialized view $(R \bowtie S)_{ptr(a)}$ is illustrated in Figure 6.

$R0^{11.15.95}_{ptr(a)}$

$R.tid$	$R.a$
r_3	3
r_4	4

$R0^{11.17.95}_{ptr(a)}$

$R.tid$	$R.a$
r_4	4
r_6	7

$S0^{11.15.95}_{ptr(a)}$

$S.tid$	$S.a$
s_3	5
s_4	6

$S0^{11.17.95}_{ptr(a)}$

$S.tid$	$S.a$
s_4	6
s_6	8

Figure 5: Partially materialized views for non-joining tuples

$V^{11.15.95}_{ptr(a)}$

$R.tid$	$S.tid$	a
r_1	s_1	1
r_2	s_2	2

$V^{11.17.95}_{ptr(a)}$

$R.tid$	$S.tid$	a
r_1	s_1	1
r_2	s_2	2
r_3	s_5	3
r_5	s_3	5

Figure 6: Partially materialized join views

The following data is stored at the warehouse: A partially materialized join view, $V_{ptr(a)}$, two partially materialized sets of non-joining tuples, $R0_{ptr(a)}$ and $S0_{ptr(a)}$, and copies of the join operands, R and S.

Since *PartMatJoin* stores and maintains V as the partially materialized view $V_{ptr(a)}$ the joins $R_I \bowtie (S1^{old} \setminus S_D)$ and $(R1^{old} \setminus R_D) \bowtie S_I$ can be performed directly on the partially materialized join ViewCache. The joins $R_I \bowtie (S0^{old} \setminus S_D)$, $(R0^{old} \setminus R_D) \bowtie S_I$ and $R_I \bowtie S_I$ are performed during Steps 1-3.

PartMatJoin. Partially Materialized Join ViewCache.

Establish the following duplicate free tuple sets during Steps 1-5:

- JP: Tuple identifier pairs for new joining tuple pairs.
- NRI: Tuple identifiers for non-joining R_I tuples.
- NSI: Tuple identifiers for non-joining R_I tuples.
- $JR0$: Tuple identifiers for joining $R0$ tuples.
- $JS0$: Tuple identifiers for joining $S0$ tuples.

1. Compute $R_I \bowtie S_I$.
2. Compute $(R0^{old}_{ptr(a)} \setminus R_D) \bowtie S_I$.
3. Compute $R_I \bowtie (S0^{old}_{ptr(a)} \setminus S_D)$.
4. FOR each $(R \bowtie S)_{ptr(a)}$ page, p, DO
  4.1. Compute $R_I \bowtie p$.
  4.2. Compute $p \bowtie S_I$.
  4.3. Insert JP into p.
  4.4. Rewrite and materialize p.
5. FOR each $R0^{old}_{ptr(a)}$ page, p, DO
  5.1. Delete $JR0$ from p.
  5.2. Insert NRI into p.
  5.3. Rewrite p.
6. FOR each $S0^{old}_{ptr(a)}$ page, p, DO
  6.1. Delete $JS0$ from p.
  6.2. Insert NSI into p.
  6.3. Rewrite p.

PartMatJoin works as follows. In Steps 1-4 the 5 joins from Equation (6) are performed. The following five variables are maintained. JP (Joining Pairs) contains the set of new joining tuple pairs whose tuple identifiers and join attribute must be added to the join cache. NRI (Non-joining R_I tuples) contain the subset of R_I that does not join with any S tuples. The tuple identifier and join attribute of these tuples must be added to $R0_{ptr(a)}$. NSI (Non-joining S_I tuples) contain the subset of S_I that does not join with any R tuples. The tuple identifier and join attribute of these tuples must be added to $S0_{ptr(a)}$. JR0 (Joining $R0$ tuples) contain the subset of $R0^{old}$ that joins with at least one S tuple. The tuple identifier and join attribute of these tuples must be removed $R0_{ptr(a)}$. JS0 (Joining $S0$ tuples) contain the subset of $S0^{old}$ that joins with at least one R tuple. The tuple identifier and join attribute of these tuples must be removed $S0_{ptr(a)}$. In Steps 5-6 $R0$ and $S0$ are updated.

4.3 Partially Materialized Operand ViewCaches

The algorithm PartMatOp [Bæk95] is an incremental algorithm that is identical to ViewCache with the following exception. *ViewCache* is designed to handle both insertions to and deletions from R and S.

The joins in Steps 1-2 are based on partially materialized views for R and S rather than on R and S. $R_{ptr(a)}$ contains two attributes, tid and $R.a$, and $S_{ptr(a)}$ contains two attributes, tid and $S.a$, as illustrated in Figure 7.

The tuple identifiers in the partially materialized views are used for the purpose of updating the view pointer cache. The attributes, $R.a$ and $S.a$, are used by the joins that implement Equation (3).

$R^{11.15.95}_{ptr(a)}$

$R.tid$	$R.a$
r_1	1
r_2	2
r_3	3
r_4	4

$R^{11.17.95}_{ptr(a)}$

$R.tid$	$R.a$
r_1	1
r_2	2
r_3	3
r_4	4
r_5	5
r_6	7

$S^{11.15.95}_{ptr(a)}$

$S.tid$	$S.a$
s_1	1
s_2	2
s_3	5
s_4	6

$S^{11.17.95}_{ptr(a)}$

$S.tid$	$S.a$
s_1	1
s_2	2
s_3	5
s_4	6
s_5	3
s_6	8

Figure 7: Partially materialized views for base relations

The following data is stored at the warehouse: A binary view pointer cache, $(R \bowtie S)_{ptr}$, two partially materialized views, $R_{ptr(a)}$ and $S_{ptr(a)}$, and copies of the join operands, R and S.

PartMatOp. **Partially Materialized Operand ViewCaches.**

1. FOR each $R^{old}_{ptr(a)}$ page, p, DO
- **1.1.** Delete R_D.
- **1.2.** Insert R_I.
- **1.3.** Rewrite p.
- **1.4.** Compute $p \bowtie S_I$.

2. FOR each $S^{old}_{ptr(a)}$ page, p, DO
- **2.1.** Delete S_D.
- **2.2.** Insert S_I.
- **2.3.** Rewrite p.
- **2.4.** Compute $R_I \bowtie p$.

3. FOR each $(R \bowtie S)_{ptr}$ page, p, DO
- **3.1.** Delete $(R_D, -)$ and $(-, S_D)$.
- **3.2.** Insert joining pairs from Steps 1.4 and 2.4. Avoid duplicates.
- **3.3.** Rewrite p.
- **3.4.** Materialize p.

PartMatOp works as follows. In Step 1 $R_{ptr(a)}$ us updated and $R^{new}_{ptr(a)}$ is joined with S_I. In Step 2 $S_{ptr(a)}$ is updated and $S^{new}_{ptr(a)}$ is joined with R_I. Thus, PartMatOp uses $R^{new}_{ptr(a)}$ and $S^{new}_{ptr(a)}$ to implement the joins in Equation (3). The tuple identifier pairs that represent the joining tuple pairs are saved and used for the cache update in Step 3 where the join pointer cache is scanned. For each cache page R_D and S_D are used to identify the tuple pairs that must be deleted from the current cache page (Step 3.1). In Step 3.2 the relevant subset of the saved tuple identifier pairs from Steps 1-2 are inserted to the current cache page (Step 3.2). In Step 3.3 the updated current cache page is rewritten and materialized.

In the example the view pointer cache points to base relations. For complex joins where the operands are not base relations the pointer cache will point to partially materialized views for the operands.

5 Experiments

In this section we describe a limited experiment testing the incremental join algorithms described above. All the experiments were run in the ADMS prototype [RES93]. The ADMS engine and its optimizer have been developed to take advantage of cached views [CR94].

We ran the experiments to measure the I/O cost of the five algorithms described in section 2: the two basic categories of fully Materialized Views, the ViewCache Pointer based one, and two Partially Materialized and partially ViewCaches.

1. Materialized Join View: the Warehouse stores the tuples of the result and, in separate relations, the tuples which do not appear in the Join View. These are necessary for discovering tuple joins that were not joining before, but they may join with newly inserted tuples on the other relation.
2. Materialized OuterJoin View: the Warehouse stores the outer join. This view does not need any additional data as the not joining tuples are stored in the OuterJoin view.
3. Join ViewCache: Only ViewCache pointers to the underlying relations which, like all pointer based views, are also stored in the warehouse.
4. Partially Materialized Join ViewCache: the warehouse stores the ViewCache augmented with all the joining values of the underlying tuples next to the pointers.
5. Partially Materialized Operand ViewCache: the ROWID and the joining attribute values, and a ViewCache pointer view.

5.1 Experiment Design

We used a variation of the Wisconsin Benchmark relations [BT94]. Two of them, R and T, have 10,000 records each while a third one, S, contains 24,000. In each of R and T, 20% of the tuples do not join with any tuples of S (part of $R0$ and $T0$). The join views that were created and stored in the warehouse are:

1. *key-key join* where the joining attribute was a key on both relations, which produce a tuple count of 10,000.
2. *key-foreign key join* where the joining attribute was a key on the first and foreign key on the second, which results in a tuple count of 24,000.
3. *non key-non key join* where neither of the joining attributes were keys in their respective relations, and produces 48,000 tuples.

After the creation, 10% insertions were applied to each of the base relations and the incremental algorithms were performed. The I/O needed to perform the algorithms were obtained from the ADMS buffer manager statistics of page faults.

We did not test the algorithms under deletions. The reason is that, materialized view based methods are very different than pointer based ones. The first category requires sophisticated preprocessing of the logs and non-negligible I/O – comparable to duplicate elimination – while pointer based methods incur no cost for deletions when done in the same pass with the processing of insertions [Rou91].

5.2 Storage Overhead

First, we provide storage statistics for each of the view categories. Table 1 indicates the amount of storage in excess of the total storage cost required for the base relations. Note that in these sizes we assume that for the Materialized Join and OuterJoin Views, the warehouse does not store a copy of the base relations but, for the pointer based categories, it does.

The table shows that materialized views have an overhead ranging from 20-324% and this is caused by the multiplicity factor of the joining tuples, with the worst case being the non-key to non-key join. On the other hand, the pointer based methods incur overhead that ranges between 2-12%. This significant difference in storage overhead is also the main contributing factor for similar difference in I/O performance discussed in the next subsection.

	Key-Key	Key-Foreign Key	Non Key-Non Key
Materialized Join View	1.22	1.64	3.24
Materialized OuterJoin View	1.20	1.58	3.18
Join ViewCache	1.02	1.03	1.06
Partial Mater. Join ViewCache	1.04	1.06	1.11
Partial Mater. Operand ViewCaches	1.07	1.04	1.12

Table 1: Storage Overhead

5.3 Performance of Algorithms

In each of the described algorithms, we applied the best, to our knowledge, method. Most of the incremental algorithms are based on two-way hash joins but, for the materialized view category, we used a 3-way hash based join algorithm which hashes the insertion logs of the two relations, and then scans the materialized join view or outer join view once. Then duplicate elimination of the resulting tuples was done afterwards. For the pointer-based algorithms, duplicate elimination is done on the fly using hash bit vectors on the ROWIDs of the tuples.

	Key-Key	Key-Foreign Key	Non Key-Non Key
Materialized Join View	7436	13360	45492
Materialized OuterJoin View	6457	12473	44491
Join ViewCache	2739	4704	4815
Partial Mater. Join ViewCache	671	1307	1714
Partial Mater. Operand ViewCaches	582	893	1093

Table 2: Comparison of I/O of View Incremental Maintenance Algorithms

Table 2 shows the total I/O count for all methods. From the materialized based ones the materialized outer join is slightly better than the regular join view. This slim advantage is due to the one-pass of the algorithm on a single relation as opposed to separate passes on the small joins $R_I \bowtie S_I$, $R0 \bowtie S_I$, and $R_I \bowtie S0$. However, dramatic performance difference between the materialized and the pointer based methods was observed. Apparently, the materialized view based methods incur high I/O volume due to their mere size. Although some improvement using better techniques and indexing may be possible, it is doubtful that such an improvements can reduce the I/O to a point to compete with the pointer based techniques. Amongst the pointer based ones, the straight pointer ViewCache spends more than 95% of its I/O retrieving from the underlying base relations for obtaining the joining predicate attribute values. This lead us to the last two pointer based algorithms which keep these values in a easier, more dense, and a lot less I/O intensive disk cache.

6 Conclusion

In this paper we describe and compare view refreshment algorithms that are based on different combinations of materialized views, partially materialized views, and pointers. Three new algorithms were presented along with their necessary data structures. One of them, uses the outer join materialized view and can bethought of an extension of the materialized view. The other two, partially materialize views and extend the ViewCache pointer based approach. These last two, offer a middle of the road approach between totally materialized and pointer based views. Namely, pointers are augmented with the necessary data values to do the incremental refresh, without digging into the bulky base relations and/or materialized views. The advantages of these methods were shown in reduction to both storage and cost overhead.

We believe that there is a lot of potential in combining partially materialized data values and pointers in OO and Multimedia database applications. *Active control* data values, such as the joining attribute values, can be separated from the bulky and mostly *passive* data, such as images or audio clips, which are only retrieved when are needed.

References

[Bæk95] L. Bækgaard. The Predicate Indexed Incremental Join. R 95-2019, Department of Mathematics and Computer Science, Aalborg University, Denmark, 1995.

[BLT86] J.A. Blakeley, P.-Å. Larson, and F.Wm. Tompa. Efficiently Udating Materialized Views. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data, Washington, D.C.*, pages 61–71, 1986.

[BM90] J.A. Blakeley and N.L. Martin. Join Index, Materialized View, and Hybrid-Hash Join: A Performance Analysis. In *Proceedings of the Sixth International Conference on Data Engineering*, pages 256–263, February 1990.

[BM95a] L. Bækgaard and L. Mark. Incremental Computation of Nested Relational Query Expressions. *ACM Transactions on Database Systems*, 20(2):111–148, June 1995.

[BM95b] L. Bækgaard and L. Mark. Incremental Computation of Set Difference Views. *IEEE Transactions on Knowledge and Data Engineering*, 1995. In press.

[BM95c] L. Bækgaard and L. Mark. Incremental Computation of Time-Varying Query Expressions. *IEEE Transactions on Knowledge and Data Engineering*, 7(4):583–590, August 1995.

[BT94] D. Bitton and C. Turbyfill. A Retrospective on the Wisconsin Benchmark. In M. Stonebraker, editor, *Readings in Database Systems*, pages 422–441. Morgan kaufmann, 1994.

[CKPS95] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing Queries with Materialized Views. In *1995 Proceedings of the 11th International Conference on Data Engineering, Taipei, Taiwan*, 1995.

[CR94] C.M. Chen and N. Roussopoulos. The implementation and Performance Evaluation of the ADMS Query Optimizer: Integrating Query Result Caching and Matching. In *Procs. of the 4th Intl. Conf. on Extending Database Technology*, 1994.

[CSL+90] M. Carey, E. Shekita, G. Lapsis, B. Lindsay, and J. McPherson. An Incremental Join Attachment for Starburst. In *1990 Proceedings of the 16th International Conference on Very Large Databases, Brisbane, Australia*, 1990.

[CW91] S. Ceri and J. Widom. Deriving Production Rules for Incremental View Maintenance. In *1991 Proceedings of the 17th International Conference on Very Large Databases, Barcelona, Spain*, 1991.

[GL95] T. Griffin and L. Libkin. Incremental Maintenance of Views with Duplicates. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, USA*, 1995.

[GMR95] A. Gupta, I.S. Mumick, and K.A. Ross. Adapting Materialized Views after Redefinitions. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, USA*, pages 211–222, 1995.

[GMS93] A. Gupta, I.S. Mumick, and V.S. Subrahmanian. Maintaining Views Incrementally. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington D.C., USA*, 1993.

[Han87] E.N. Hanson. A Performance Analysis of View Materialization Strategies. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data, San Fransisco, California*, pages 440–453, 1987.

[Han92] E.N. Hanson. Rule Condition Testing and Action Execution in ARIAL. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA*, 1992.

[HCKW90] E.N. Hanson, M. Chaabouni, C.-H. Kim, and Y.-W. Wang. A Predicate Matching Algorithm for Database and Rule Systems. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, New Jersey*, 1990.

[HK95] C. Hamon and A.M. Keller. Two-Level Caching of Composite Object Views of Relational Databases. In *1995 Proceedings of the 11th International Conference on Data Engineering, Taipei, Taiwan*, 1995.

[Inm93] W.H. Inmon. *Building the Data Warehouse*. John Wiley & Sons, Inc., 1993.

[JMR91] C.S. Jensen, L. Mark, and N. Roussopoulos. Incremental Implementation Model for Relational Databases with Transaction Time. *IEEE Transactions on Knowledge and Data Engineering*, 3(4):461–473, 1991.

[LHM+86] B. Lindsay, L. Haas, C. Mohan, H. Pirahesh, and P. Wilms. A Snapshot Differential Refresh Algorithm. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data, Washington, D.C.*, 1986.

[LMSS95] J.J. Lu, G. Moerkotte, J. Schue, and V.S. Subrahmanian. Efficient Maintenance of Materialized Mediated Views. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, USA*, pages 340–351, 1995.

[McK88] L.E. McKenzie. An Algebraic Language for Query and Update of Temporal Databases. TR 88-050, University of North Carolina at Chapel Hill, Department of Computer Science, USA, 1988.

[QW91] X. Qian and G. Wiederhold. Incremental Recomputation of Active Relational Expressions. *IEEE Transactions on Knowledge and Data Engineering*, 3(3):337–341, 1991.

[RCBB89] A. Rosenthal, S. Chakravarthe, B. Blaustein, and J. Blakeley. Situation Monitoring for Active Databases. In *1989 Proceedings of the 15th International Conference on Very Large Data Bases, Amsterdam, The Netherlands*, 1989.

[RES93] N. Roussopoulos, N. Economou, and A. Stamenas. ADMS: A Testbed for Incremental Access Methods. Technical Report UMIACS-TR-90-103, University of Maryland Institute for Advanced Computer Studies, 1993.

[RK91] N. Roussopoulos and H. Kang. A Pipeline N-way Join Algorithm Based on the 2-way Semijoin Program. *IEEE Transactions on Knowledge and Data Engineering*, 3(4):486–495, December 1991.

[Rou82a] N. Roussopoulos. The Logical Access Path Schema of a Database. *IEEE Transactions on Software Engineering*, SE-8(6):563–573, 1982.

[Rou82b] N. Roussopoulos. View Indexing in Relational Databases. *ACM Transactions on Database Systems*, 7(2):258–290, June 1982.

[Rou91] N. Roussopoulos. An Incremental Access Method for Viewcache: Concept, Algorithms, and Cost Analysis. *ACM Transactions on Database Systems*, 16(3):535–563, September 1991.

[SJGP90] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On Rules, Procedures, Caching and Views in Data Basa Systems. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, New Jersey*, pages 281–290, 1990.

[SL76] D.G. Severance and G.M. Lohman. Differential Files: Their Application to the Maintenance of Large Databases. *ACM Transactions on Database Systems*, 1(3):256–267, September 1976.

[SR87] K. Subieta and W. Rzeczkowski. Query Optimization by Stored Queries. In *1987 Proceedings of the13th International Conference on Very Large Databases, Brighton, Great Britain*, pages 369–380, 1987.

[Sto75] M. Stonebraker. Implementation of Integrity Constraints and Views by Query Modification. In *Proceedings of the 1975 SIGMOD Workshop on Management of Data, San Jose, California*, pages 65–78, 1975.

[Val87] P. Valduriez. Join Indices. *ACM Transactions on Database Systems*, 12(2):218–246, June 1987.

[WDSY91] O. Wolfson, H.M. Dewan, S.J. Stolfo, and Y. Yemini. Incremental Evaluation of Rules and its Relationship to Parallelism. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data, Denver, Colorado, USA*, 1991.

[ZGMHW95] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View Maintenance in a Warehousing Environment. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, USA*, pages 316–327, 1995.