Mediator Join Indices

Ling Ling Yan M. Tamer Özsu Ling Liu
Laboratory for Database Systems Research
Department of Computing Science
University of Alberta, Edmonton, Alberta, T6G 2H1
{ling, ozsu, lingliu}@cs.ualberta.ca

Abstract

A mediator join index (MJI) is proposed to speed up N-way inter-database joins by reducing the amount of data transfer during evaluation. A family of algorithms, the Query Scrubbing Algorithms (QSA), are developed to maintain MJI and to evaluate queries using MJI. QSA algorithms use query scrubbing to cope with update and query anomalies related to materialized views in the mediator context. Compared with existing algorithms, QSA algorithms incur less overhead in handling the anomalies and makes MJI a promising technique for efficient mediator query processing.

1. Introduction

In recent years, the need to access data in multiple data sources increased dramatically. This type of access is complicated by heterogeneities and autonomy of the data sources. The mediator architecture can be used to manage this complexity [7]. Mediators are used in many contexts. When using mediators to provide integrated views over multiple information sources and entertain mediator queries posed against these views, a major research issue is the performance of mediator query processing, that is, mediator query optimization. There are many established techniques for query optimization in distributed and centralized database systems. One important technique in this category is indexing. It has long been established that the proper use of indices can greatly speed up the processing of certain queries. In the mediator context, indexing would offer the same benefit but is rarely discussed. Indeed, indices in mediator context can not be maintained and used in the same way as in centralized systems; they pose new problems. In this paper, we investigate using *Mediator Join Index (MJI)* to speed up N-way inter-database joins in the mediator context.

1.1. What is MJI?

We start with a motivating example. Consider a mediator view relation

$$R_V = R_1 \bowtie_{f_1} R_2 \bowtie_{f_2} R_3$$

where R_1, R_2 and R_3 reside in different databases. Let K_1, K_2 and K_3 be the respective keys of the 3 relations. To process a query against R_V, the mediator must evaluate a 3-way inter-database join. If we materialize relation

$$IDX_{R_V} = \pi_{K_1,K_2,K_3}(R_V)$$

in the mediator, then at query evaluation time, the mediator *knows* which tuples contribute to the final join result and only retrieves these "useful" tuples from the respective sources. Hence the amount of data transferred is minimized. This idea is similar to join index [6] except that we use relational keys instead of surrogates.

Generally, a mediator join index (MJI) is a structure defined over an N-way join view

$$V = R_1 \bowtie_{f_1} R_2 \bowtie_{f_2} ... \bowtie_{f_{N-1}} R_N$$

where relations $R_1, ..., R_N$ reside in different data sources. Let $K_1, ..., K_N$ be the respective keys of these N relations. The MJI over V is defined as

$$MJI_V = \pi_{K_1,...,K_N}(V)$$

The maintenance of MJI and its usage in query evaluation are two important issues that pose new problems in the mediator context (Section 1.2). Our goal is to solve these problems and establish MJI as an indexing technique that supports fast N-way inter-database joins.

1.2. Maintaining and Using MJI

Given an N-way join view, its MJI will be maintained and used by the mediator. Both maintenance and usage of

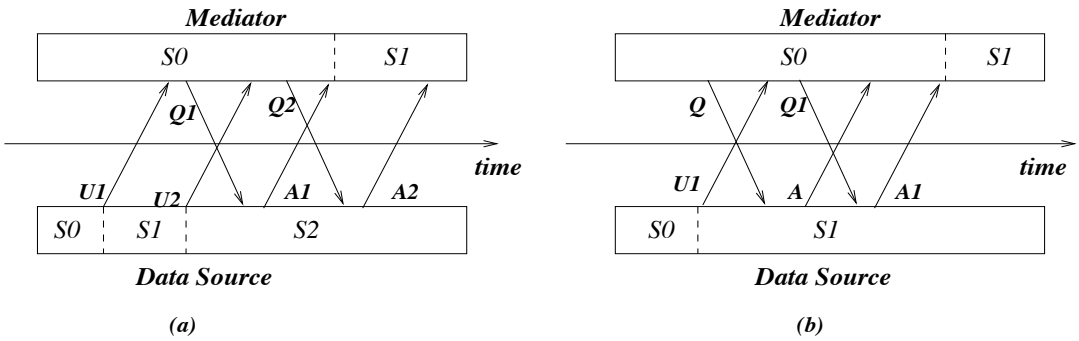

Figure 1. View Update Anomaly and Query Anomaly

MJI poses new problems. We review these problems in this subsection.

MJI is a materialized view in the mediator. Materialized views are not a new idea; they have long been used as a means for improving the performance of certain queries. Traditionally, materialized views are maintained incrementally [1]. To propagate an update U_1 on R_1 into a materialized view $V = v(R_1, ..., R_N)$, a *view maintenance query (VMQ)* due to U_1, $VMQ(U_1)$, is performed, such that

$$v(R_1 + U_1, R_2, ..., R_N) = v(R_1, ..., R_N) + VMQ(U_1)$$

The source updates (e.g. U_1), the VMQ of these updates (e.g. $VMQ(U_1)$), and the view updates are performed atomically in a single transaction. In mediator context, incremental maintenance of materialized views involves a new problem as illustrated in Figure 1(a). VMQ $Q1$ is evaluated in state $S2$, while it *should be* evaluated in $S1$. The interleaving of source updates and VMQs causes *view update anomalies* [8]. Using the traditional view maintenance algorithm in the mediator context may result in invalid view states. Algorithms have been developed to cope with these anomalies in data warehousing environment [8, 9]. The relationship between these algorithms and our algorithms will become clear in Section 1.3.

When processing a view query using MJI, we decompose the query based on the current content of MJI, send the subqueries to relevant sources, and assemble the query answer using the subquery results. This simplistic strategy works incorrectly in the scenario shown in Figure 1(b). The query is decomposed based on MJI state corresponding to source state $S0$. However, subquery Q sent to the source is evaluated in state $S1$ while it *should be* evaluated in state $S0$. When such a case occurs, we say there are *query anomalies*. Analysis and example of query anomalies are given in Section 4.

1.3. Scope and Contributions

Algorithms for maintaining and evaluating queries with MJI must cope with both update anomalies and query anomalies. In this paper, we present such algorithms. These algorithms use *query scrubbing* and are hence called the *Query Scrubbing Algorithms*, or the QSA family. The general idea of query scrubbing is the following. We notice that anomalies happen only when we query a source, either for the purpose of view maintenance, or for processing view queries. We further observe that ultimately, the anomalies arise when these queries are processed in a source state that is "newer" than the "old" source state based on which the materialized information held at the mediator is derived. Assume the mediator is aware of all the difference between the new state and the old source state. To eliminate anomalies, all the mediator has to do is to make modifications to the query results to *pretend* that the query is evaluated based on the "old" source state. This is what we refer to as query scrubbing.

We further illustrate the idea of query scrubbing using Figure 1. In Figure 1(a), a QSA algorithm for view maintenance will do the following: upon receiving answer $A1$ and knowing that $A1$ is affected by $U2$ (since $U2$ notification arrived before $A1$), it "scrubs" $A1$ with $U2$, that is, it modifies $A1$ so that it appears as if $U2$ did not happen before $Q1$ was evaluated. This means that we design a procedure $VMQscrub(A1, U2)$, which "scrubs" $A1$ and returns the scrubbed $A1$ as the result. This scrubbing must satisfy the following condition:

$$V^{U1} = V + VMQscrub(A1, U2)$$

where V^{U1} is the updated state of V with $U1$ being the only source update. The "scrubbed" $A1$ is merged with the view content.

In Figure 1(b), a QSA algorithm for view query processing will do the following: upon receiving answer A and knowing that A is affected by $U1$, it scrubs A using $U1$ before returning it as the answer to Q. That is, we design a

procedure $querySCrub(A, U1)$, which "scrubs" A and returns the scrubbed A as the result. This scrubbing must satisfy the following condition:

$$Q(S0) = querySCrub(A, U1)$$

where $Q(S0)$ is the result of Q evaluated in state $S0$.

A Strobe family algorithm for maintaining materialized mediator views, the C-Strobe [9], uses query scrubbing. While QSA algorithms perform scrubbing entirely in the mediator, without further querying the data sources, C-Strobe does so by issuing source queries. These source queries may need to be scrubbed too, incurring more source queries. C-Strobe does not terminate until all such queries are cleared. QSA algorithms incur less overhead in query scrubbing and they keep the view content "fresher", as shown in Section 3. A MJI-indexed view is indeed a *hybrid view*, or a partially materialized view. A reversed Eager Compensation Algorithm (ECA) [8] is briefly suggested in [4] to overcome query anomalies but no algorithmic details are provided. The query processing algorithm we give solves this problem in the special case of MJI.

The rest of this paper is organized as follows. Section 2 contains definitions, notions and assumptions. Section 3 presents MJI maintenance algorithms. Section 4 discusses query processing using MJI. Section 5 discusses the limitations of using MJI and gives a performance perspective. Section 6 contains conclusions and future work.

2. Mediator Join Index

2.1. Join Indices and Mediator Join Indices

Join indices were studied in [6]. Assume that each tuple of a relation is identified by a unique *surrogate* that never changes. Denote the surrogate of tuple i in R as r_i and the surrogate of tuple j in S as s_j. Denote the tuple identified by a surrogate c as $tuple(c)$. The join index for join $R \bowtie_f S$ is defined as

$$JI = \{[r_i, s_j] \mid f(tuple(r_i), tuple(s_j)) = true\}$$

To adapt this definition into the mediator context, we use the relational keys instead of surrogates.

DEFINITION **2.1** [Mediator Join Index (MJI)]
Let $R_1, ..., R_N$ be relations that reside on (possibly) different databases. Let $K_1, ..., K_N$ be the keys of relations $R_1, ..., R_N$, respectively. Let V be a mediator view defined as

$$V = R_1 \bowtie_{f_1} R_2 \bowtie_{f_2} ... \bowtie_{f_{N-1}} R_N$$

The mediator join index (MJI) for V is a relation $MJI(K_1, ..., K_N)$ defined by the following:

$$MJI = \pi_{K_1,...,K_N}(R_1 \bowtie_{f_1} R_2 \bowtie_{f_2} ... \bowtie_{f_{N-1}} R_N)$$

□

2.2. Consistency, Freshness and Correct Query Processing with MJI

We adopt the definitions of consistency and freshness in [4]. Let

$$MJI = \pi_{K_1,...,K_N}(R_1 \bowtie_{f_1} R_2 \bowtie_{f_2} ... \bowtie_{f_{N-1}} R_N)$$

MJI is *consistent* with $R_1, ..., R_N$ if (1) At any time t, MJI is based on states of $R_1, .., R_N$ at times $t_1, ..., t_N$, respectively, where t_i's are in the past of t. That is, MJI is *valid* and does not forecast the future; and (2) The more recent MJI state corresponds to more recent states of $R_1, ..., R_N$. That is, MJI does not "flash back". The *guaranteed freshness*, $< T_1, ..., T_N >$, is such that, at any time t, if MJI corresponds to R_i state at time t_i $(1 \leq i \leq N)$, then $t - t_i \leq T_i$. Given a query Q against a MJI-indexed view relation, the processing of Q is *correct* if the MJI state based on which query decomposition is performed is consistent with the source states based on which subqueries are evaluated.

2.3. Notions and Assumptions

We assume that update notifications issued by data sources are in the form of

$$insert(DB_i, R_i, t)$$

or

$$delete(DB_i, R_i, t)$$

where $1 \leq i \leq N$. For MJI maintenance, deletions are propagated without querying the sources since MJI includes keys of all source relations. For insertions, *View Maintenance Queries (VMQs)* are generated as:

$$VMQ(insert(DB_i, R_i, t)) = \pi_{K_1,...,K_N}(R_1 \bowtie_{f_1} ... \bowtie_{f_{i-1}} \{t\} \bowtie_{f_i} R_{i+1} \bowtie ... \bowtie_{f_{N-1}} R_N) \quad (1)$$

We make the following assumptions about data sources:

1. Each DB_i $(1 \leq i \leq N)$ provides *immediate notifications* to the mediator. That is, DB_i sends a notification to the mediator immediately after an update is successfully committed.
2. Mediator receives messages from $DB_i (1 \leq i \leq N)$ in the same order as they are sent.

In Section 5, problems related to these assumptions are discussed. In what follows, we use

$$DEL_{cond}(S, p)$$

to denote an operation that deletes all S tuples that satisfy predicate p.

3. Mediator Join Index Maintenance

Once created, a MJI is maintained by propagating changes in the source relations into the index content incrementally. In this section, we present the MJI maintenance algorithm, QSA-MJI. This algorithm uses query scrubbing, introduced in Section 1.3. We maintain a *Pending Update Queue*, PUQ, that contains all source updates to be propagated into the index. Updates in PUQ are propagated into the index in a FIFO order. For any $U \in PUQ$, if U is a deletion, it is propagated without querying the sources; if U is an insertion, a VMQ is generated as in formula (1). This VMQ is evaluated by function $VMQevaluate$. The answer returned by $VMQevaluate$ is merged into the materialized view immediately unless some updates in *front* of U in PUQ are still waiting for answers to their VMQs to be computed. Given VMQ $Q = \pi_{K_1,...,K_N}(R_1 \bowtie_{f_1} ... \bowtie_{f_{i-1}} \{t\} \bowtie_{f_i} R_i \bowtie ... \bowtie_{f_{N-1}} R_N)$, $VMQevaluate$ identifies a permutation of $\{1, ..., N\}, Z = \{i, j_1, ..., j_{N-1}\}$ and a set of predicates $\{f'_1, ..., f'_{N-1}\}$, such that $R_1 \bowtie_{f_1} ... \bowtie_{f_{i-1}} \{t\} \bowtie_{f_i} R_i \bowtie ... \bowtie_{f_{N-1}} R_N = \{t\} \bowtie_{f'_1} R_{j_1} \bowtie_{f'_2} ... \bowtie_{f'_{N-1}} R_{j_{N-1}}$. The right hand side of his equation is computed in $N-1$ loops. In the first loop, R_{j_1} is joined with $\{t\}$. Subsequent loops join the result of the previous loop with the next relation in sequence Z, identified by function $next_source$ which returns the next relation to be joined as well as the join predicate to be used. $next_source$ returns nil when Z is exhausted. Each join is computed by issuing a query to the source where the next relation resides. As shown in Figure 1(a), the result of this query is potentially "dirty" and must be scrubbed. Query scrubbing is performed by function $VMQscrub$, which eliminates the effect of further updates in the data source from the query result.

Algorithm-1 QSA-MJI

At $DB_i(1 = 1...N)$:

- Upon committing update U: notify the mediator of U.
- Upon receiving query Q: evaluate Q based on the current state and return answer to mediator.

At Mediator:

- Upon receiving an update U_i:
  1. Add U_i to the end of PUQ.
  2. If U_i is an insertion, let $Q = VMQ(U_i)$. Call $A = VMQevaluate(V, U, Q)$.
  3. If U_i is the first element in PUQ, call $ViewUpdate$; otherwise do nothing. ∎

Function $VMQevaluate(V, U, Q)$
Input:

V: the view being indexed.
$V = R_1 \bowtie_{f_1} ... \bowtie_{f_{i-1}} \{t\} \bowtie_{f_i} R_i \bowtie ... \bowtie_{f_{N-1}} R_N$.

U: an insert notification from DB_i,
$U = insert(DB_i, R_i, t)$.

Q: $Q = VMQ(U) = \pi_{K_1,...,K_N}(R_1 \bowtie_{f_1} ... \bowtie_{f_{i-1}} \{t\} \bowtie_{f_i} R_{i+1} \bowtie ... \bowtie_{f_{N-1}} R_N)$.

Output: A: a correct answer to Q.

1. $A = \{t\}$. $EVAL = \{R_i\}$. $< source, pred >= next_source(V, EVAL)$.
2. **WHILE** ($source \neq nil$) **DO**
   (a) $Q_s = A \bowtie_{pred} R_{source}$.
   Send Q_s to DB_{source} and wait for the answer.
   (b) When answer to Q_s, A_s, arrives at the mediator, do the following:
      i. Set UUQ = {all DB_{source} updates that are behind U in PUQ}.
      ii. $A = VMQscrub(DB_{source}, Q_s, A_s, UUQ)$.
   (c) $EVAL = EVAL \cup \{R_{source}\}$.
   $< source, pred > = next_source(V, EVAL)$.
3. **RETURN** $\pi_{K_1,...,K_N}(A)$. ∎

Function $VMQscrub(DB_i, Q_s, A_s, UUQ)$
Input:

DB_i: the relevant data source.

Q_s: $Q_s = A \bowtie_f R_i$, where A is some intermediate result as in $VMQevaluate$ 2 (a).

A_s: a "dirty" answer to Q_s.

UUQ: $UUQ = U_1, ..., U_M$, $U_k(k = 1, M)$ is $insert(DB_i, R_i, t)$ or $delete(DB_i, R_i, t)$.

Output: A "clean" (correct) answer to Q, that is, the effect of updates in UUQ removed from A_s.

1. **FOR** $k = M, 1$ **DO**
   If $U_k = insert(DB_i, R_i, t_x)$, then
   $DEL_{cond}(A_s, A_s[K_i] = t_x[K_i])$.
   If $U_k = delete(DB_i, R_i, t_d)$, then
   $A_s = A_s \cup (A \bowtie_f \{t_d\})$.
2. **RETURN** A_s. ∎

Procedure $ViewUpdate$

WHILE (true) **DO**

1. If PUQ is empty, **RETURN**; otherwise, let U be the first element in PUQ.
2. If $U = delete(DB_i, R_i, t_d)$, then $DEL_{cond}(MJI, MJI[K_i] = t_d[K_i])$.
3. If $U = insert(DB_i, R_i, t_i)$ and the answer to $VMQ(U)$, A, has been received and scrubbed by the mediator, $MJI = MJI \cup A$.
4. If the mediator is still computing the answer for $VMQ(U)$, **RETURN**.
5. Remove U from PUQ. ■

Remarks. QSA-MJI can be adapted to work in a "batch" mode, where updates that cancel each other are not propagated or used for query scrubbing. This adaptation does not raise new issues. In QSA-MJI, the number of source queries needed to propagate an insertion into a N-way MJI is at most $N-1$. Scrubbing does not incur any source queries. C-Strobe [9] uses a larger number of queries because it issues source queries to scrub (compensate) these $N-1$ queries. These compensation queries must be compensated for if the source is further updated, and so on. C-Strobe does not terminate until all the compensation queries are cleared. Besides incurring less scrubbing/compensation overhead, QSA-MJI always terminates faster than C-Strobe and hence keeps the MJI fresher.

Theorem-1. *Assume that a MJI is consistent with the relations involved in it at time t_0. Then algorithm QSA-MJI ensures the same consistency at any time $t > t_0$.* ■

Proof: Obviously, based on the immediate update notification assumption, $ViewUpdate$ mandates a correct sequence (FIFO) in propagating updates into the index. What left to be shown is that each update is propagated into the index without violating consistency. Given a consistent state of MJI at time t_0, corresponding to source states at times $< t_0^1, ..., t_0^N >$, consider the first update received at the mediator after t_0, U_j from DB_j, at time t_1. Since DB_j provides immediate notification, we know that the time at which U_j occurred, t_1^j, satisfies $t_0^j < t_1^j$. Now we prove that after propagating U_j using QSA-MJI, MJI is consistent with the states of the sources at times $< t_0^1, ..., t_0^{j-1}, t_1^j, t_0^{j+1}, ..., t_0^N >$. If $U_j = delete(DB_j, R_j, t_d)$, then we need to remove all tuples in MJI with $MJI[K_j] = t_d[K_j]$. Obviously QSA-MJI does this. If $U_j = insert(DB_j, R_j, t_x)$, a VMQ $Q = \pi_{K_1,...,K_N}(R_1 \bowtie_{f_1} ... \bowtie_{f_{j-1}} \{t_x\} \bowtie_{f_j} R_{j+1} \bowtie ... \bowtie_{f_{N-1}} R_N)$ must be evaluated. $VMQevaluate$ evaluate this query by sending queries to each data source(except DB_j). Consider data source $DB_i (i \neq j)$. R_i might have been changed since time t_0^i but these changes are unwanted since we want to make sure that the query is evaluated based on the R_i state at time t_0^i. To eliminate the effect of these changes on the subquery sent to DB_i, we perform $VMQscrub$. So now the proof boils down to establishing the following:

Given $Q_s = A \bowtie_f R_i$, and $UUQ = U_1, ..., U_M$, a queue of updates to R_i that are behind U in PUQ, where U is the update such that Q_s is a subquery of $VMQ(U)$. Let $Answer(Q, R, r)$ denote the query result of query Q with the state of relation R be r. Let r_i' be the state of R_i before the sequence of updates $U_1, ..., U_M$ happened. Let r_i be the state of R_i based on which Q_s is evaluated, that is, $A_s = Answer(Q_s, R_i, r_i)$. $VMQscrub(DB_i, Q_s, A_s, UUQ)$ satisfies the following:

$$Answer(Q_s, R_i, r_i') = VMQscrub(DB_i, Q_s, Answer(Q_s, R_i, r_i), UUQ)$$

That is, after A_s is scrubbed, it is as if updates in UUQ did not happen before Q_s is evaluated.

$VMQscrub$ "reverse" the effect of $U_1, ..., U_M$ on A_s. It does so by removing the effect of U_M, then of U_{M-1}, ..., and so on. We construct the proof by induction on M:

$\underline{M = 1.}$ If $U_1 = delete(DB_i, R_i, t_d)$, when Q_s is evaluated, the state of relation R_i, $r_i = r_i' - \{t_d\}$, where r_i' is the state of relation R_i before U_1 happened. Considering that U_1 is a successful deletion, we have: $r_i' = r_i \cup \{t_d\}$. For $Q_s = A \bowtie_f R_i$, $Answer(DB_i, Q_s, R_i, r_i') = Answer(Q_s, R_i, r_i) \cup (A \bowtie_f \{t_d\})$. $VMQscrub$ does this and is correct. If $U_1 = insert(DB_i, R_i, t_y)$, Q_s is evaluated in the state of relation R_i, $r_i = r_i' \cup \{t_y\}$, where r_i' is the state of relation R_i before U_1 happened. Considering that U_1 is a successful insertion, we have: $r_i' = r_i - \{t_y\}$. For $Q_s = A \bowtie_f R_i$, $Answer(Q_s, R_i, r_i') = DEL_{cond}(A_s, A_s[K_i] = t_y[K_i])$. $VMQscrub$ does this and is correct.

$\underline{M = k+1.}$ Assume that $VMQscrub$ is correct for $M = k$ and consider $M = k+1$. Let the state of relation R_i based on which Q_s is evaluated be r_i. Let the state of R_i before $U_1, ..., U_{k+1}$ happened be r_i'. Let the state of R_i *before* U_{k+1} happened be r_i''. Using the same process as in the case of $M = 1$, we can prove that after the first round of the **FOR** statement, we obtain $A_s = Answer(Q_s, R_i, r_i'')$, that is, the effect of update U_{k+1} is eliminated from A_s. The rest of $VMQscrub$ achieves

$$VMQscrub(DB_i, Q_s, Answer(Q_s, R_i, r_i''), \{U_1, ..., U_k\})$$

Applying the induction assumption to the above expression, we have:

$$Answer(Q_s, R_i, r_i') = VMQscrub(DB_i, Q_s, Answer(Q_s, R_i, r_i''), \{U_1, ..., U_k\})$$

Hence:

$$Answer(Q_s, R_i, r_i') = VMQscrub(DB_i, Q_s, Answer(Q_s, R_i, r_i), \{U_1, ..., U_{k+1}\}). \blacksquare$$

Example 3.1 [QSA-MJI case study] Let

$$V = R_1 \bowtie_{R_1.W=R_2.W} (R_2) \bowtie_{R_2.Y=R_3.Y} (R_3)$$

be a join view. The 3 relations and the MJI on V are shown in Table 1. We omit the join conditions when they are obvious. At time t_0, the MJI is consistent with R_1, R_2 and R_3 (Table 1). Consider the following scenario:

t_1: Mediator receives notification

$$U_1 = insert(DB_3, R_3, [32, 102])$$

from DB_3. We have $PUQ = \{U_1\}$ and

$$Q_1 = VMQ(U_1) = R_1 \bowtie R_2 \bowtie \{[32, 102]\}$$

Mediator calls $VMQevaluate(V, U_1, Q_1)$. First, query

$$Q_2 = R_2 \bowtie \{[32, 102]\}$$

is sent to DB_2.

t_2: DB_1 makes two updates

$$U_2 = insert(DB_1, R_1, [11, 1])$$

$$U_3 = (insert(DB_2, R_2, [12, 2])$$

and notifies the mediator of both updates. Processing of these updates are omitted from this example. $PUQ = \{U_1, U_2, U_3\}$.

t_3: Q_2 is returned by DB_2 as

$$A_2 = \{[22, 2, 32, 102]\}$$

At the mediator, since there is no DB_2 update in PUQ, no scrubbing. $VMQ((V, U_1, Q_1)$ continues to send query

$$Q_3 = A_2 \bowtie R_1 = \{[22, 2, 32, 102]\} \bowtie R_1$$

to DB_1.

t_4: Answer to Q_3 is returned by DB_1 as

$$A_3 = \{[12, 2, 22, 32, 102]\}$$

At the mediator, this answer must be scrubbed with updates U_2 and U_3 from DB_1, result being ϕ. If we use the traditional view maintenance algorithm, A_3 would be merged into MJI as it is, resulting in a MJI state $\{[12, 22, 32]\}$. This state is invalid because the state of R_1 that contains key 12 must also contain key 11, and the state of R_3 that contains key 32 must also contain key 31. Hence, a valid MJI state that contains [12,22,32] must also contain [11,21,31].

□

4. Query Processing Using MJI

Similar to join index, MJI is designed to speed up specific types of queries. Without loss of generality, we consider the following query in the rest of this section:

$$Q = \sigma_{p_1}(R_1) \bowtie_{f_1} ... \bowtie_{f_{N-1}} \sigma_{p_N}(R_N)$$

A simple strategy to evaluating this query using MJI consists of two steps. First, *query decomposition*. Let

$$Q_i = \sigma_{p_i}(R_i) \bowtie \pi_{K_i}(MJI)(1 \leq i \leq N)$$

Send Q_i to DB_i. Second, *answer assembly*. Let the answer to Q_i be A_i $(1 \leq i \leq N)$, compute the answer to query Q as

$$A = A_1 \bowtie_{f_1} ... \bowtie_{f_{N-1}} A_N$$

This strategy is not correct as demonstrated by the following example.

Example 4.1 Let

$$V = R_1 \bowtie_{R_1.W=R_2.W} R_2$$

be a join view. The 2 relations and the MJI on V are shown in Table 2. We omit the join conditions when they are obvious. We also omit the propagation of source updates U_1 and U_2. Assume that at time t_0, we have the states shown in Table 2, and process the query

$$Q = R_1 \bowtie R_2$$

Consider the following sequence of events:

t_0: Send query

$$Q_1 = R_1 \bowtie \pi_{K_1}(MJI)$$

to DB_1. $\pi_{K_1}(MJI) = \{[11], [12]\}$. Send query

$$Q_2 = R_2 \bowtie \pi_{K_2}(MJI)$$

to DB_2. $\pi_{K_2}(MJI) = \{[21], [22]\}$.

	$R_1(K_1, W)$	$R_2(K_2, W, Y)$	$R_3(K_3, Y)$	$MJI(K_1, K_2, K_3)$
t_0	ϕ	$[21, 1, 101], [22, 2, 102]$	$[31, 101]$	ϕ
t_1	ϕ	$[21, 1, 101], [22, 2, 102]$	$[31, 101], [32, 102]$	ϕ
t_2	$[11, 1], [12, 2]$	$[21, 1, 101], [22, 2, 102]$	$[31, 101], [32, 102]$	ϕ

Table 1. Example 3.1

	$R_1(K_1, X_1, W)$	$R_2(K_2, X_2, W)$	$MJI(K_1, K_2)$
t_0	$[11, x_{11}, 1], [12, x_{12}, 2]$	$[21, x_{21}, 1], [22, x_{22}, 2], [23, x_{23}, 3]$	$[11, 21], [12, 22]$
t_1	$[11, x_{11}, 1], [12, x_{12}, 2], [13, x_{13}, 3],$	$[21, x_{21}, 1], [22, x_{22}, 2], [23, x_{23}, 3]$	$[11, 21], [12, 22]$
t_2	$[11, x_{11}, 1], [13, x_{13}, 3],$	$[21, x_{21}, 1], [22, x_{22}, 2], [23, x_{23}, 3],$	$[11, 21], [12, 22]$

Table 2. Example 4.1

t_1: DB_1 performs

$$U_1 = insert(DB_1, R_1, [13, x_{13}, 3])$$

and notifies the mediator.

t_2: DB_1 performs

$$U_2 = delete(DB_1, R_1, [12, x_{12}, 2])$$

and notifies the mediator.

t_3: Q_1 arrives at DB_1 and is evaluated to

$$A_1 = \{[11, x_{11}, 1]\}$$

A_1 is sent to the the mediator.

t_4: Q_2 arrives at DB_2 and is evaluated to

$$A_2 = \{[21, x21, 1], [22, x_{22}, 2]\}$$

A_2 is sent to the mediator.

t_5: A_1 is received at the mediator.

t_6: A_2 is received at the mediator and the query answer to Q is computed as

$$A_1 \bowtie_W A_2 = \{[11, x_{11}, 1, 21, x_{21}]\}$$

This answer is incorrect. While the state of R_2 has never been changed, a valid state of R_1 that includes tuple $[11, x_{11}, 1]$ must contain either tuple $[12, x_{12}, 2]$ or $[13, x_{13}, 3]$. Hence a valid query answer that includes tuple $[11, x_{11}, 1, 21, x_{21}]$ must include either $[12, x_{12}, 2, 22, x_{22}]$ or $[13, x_{13}, 3, 23, x_{23}]$.

□

The above anomaly is due to the inconsistency between the data retrieved from the data sources by subqueries and the content of the MJI based on which query decomposition is done. We use a QSA algorithm to solve this problem. When processing query

$$Q = \sigma_{p_1}(R_1) \bowtie_{f_1} ... \bowtie_{f_{N-1}} \sigma_{p_N}(R_N)$$

we decompose it into sub-queries

$$Q_i = \sigma_{p_i}(R_i) \bowtie \pi_{K_i}(MJI) \quad (1 \leq i \leq N)$$

and send them to DB_i's. When answer to Q_i, A_i, arrives from DB_i, we scrub A_i to eliminate the effect of all the updates made by DB_i since Q_i was sent.

Algorithm-2 QUERY-MJI-QSA(V, MJI, Q)

Input:

V: a view. $V = R_1 \bowtie_{f_1} R_2 \bowtie_{f_2} ... \bowtie_{f_{N-1}} R_N$.

MJI: the MJI on V.

Q: $Q = \sigma_{p_1}(R_1) \bowtie_{f_1} ... \bowtie_{f_{N-1}} \sigma_{p_N}(R_N)$.

Output: A: A valid answer to Q.

- Lock MJI for read until query Q is completed.
- **FOR** $i = 1, N$ **DO**

  Let $Q_i = \sigma_{p_i}(R_i) \bowtie \pi_{K_i}(MJI)$.
  Send Q_i to DB_i.
- Upon receiving answer A_i to Q_i $(1 \leq i \leq N)$, let UUQ_i be the queue of all DB_i updates received by the mediator after Q_i is sent. Call $querySCrub(DB_i, Q_i, A_i, UUQ_i)$.

- Once all A_i's ($i = 1, N$) are received and scrubbed by the mediator, assemble the answer to Q, A, as $A = A_1 \bowtie_{f_1} A_2 .... \bowtie_{f_{N-1}} A_N$. ∎

Function $querySCrub(DB_i, Q_i, A_i, UUQ_i)$
Input:

A_i: answer to Q_i, returned by DB_i.

DB_i: data source involved in Q_i.

Q_i: the subquery sent to DB_i, $Q_i = \sigma_{p_i}(R_i \bowtie MJI[K_i])$.

UUQ_i: the unwanted update queue for Q_i, $UUQ_i = U_1, ..., U_M$.

Output: Scrubbed A_i with effect of updates in UUQ_i eliminated.

1. **FOR** $k = M$ to 1 **DO:**
   (a) If $U_k = insert(DB_i, R_i, t_x)$, $DEL_{cond}(A_i, A_i[K_i] = t_x[K_i])$.
   (b) If $U_k = delete(DB_i, R_i, t_d)$ and $t_d[K_i] \in \pi_{K_i}(MJI)$, apply predicate p_i to tuple t_d. If it is true, $A_i = A_i \cup \{t_d\}$.
2. **RETURN** A_i. ∎

Theorem-2. Algorithm QUERY-MJI-QSA processes queries correctly (ref. Section 2.2). ∎

This theorem can be established in similar fashion as Theorem-1.

Example 4.2 We apply the new algorithms to handle the scenario in Example 4.1. At t_5, when Q_1 arrives, the mediator knows that since Q_1 was sent, there has been two updates in DB_1, U_1 and U_2. Scrub A_1 with these two updates, we get $A_1 = \{[11, x_{11}, 1], [12, x_{12}, 2]\}$. At t_6, A_2 arrives but there has been no updates from DB_2, hence A_2 is not scrubbed. Assemble the query result as usual we get $A = A_1 \bowtie_W A_2 = \{[11, x_{11}, 1, 21, x_{21}], [12, x_{12}, 2, 22, x_{22}]\}$. This answer is valid. □

5. Limitations and Performance

5.1. Limitations of MJI

In general, the use of MJI is limited by the capabilities of the data sources involved in providing update notifications with certain qualities. So far, we have assumed that (1) data sources provide immediate update notification; and (2) messages from a given source are delivered to the mediator in the same order as they are sent. Queries sent to sources that do not satisfy (1) and (2) can not be scrubbed, as discussed in this paper, or compensated for, as in [8, 9], hence their validity can not be guaranteed. If a source provides unordered update notification, we can attach a sequence number to the messages from this source to the mediator. This way, the mediator can trace the source state changes. We say a source is *inactive* if it only provides periodical update notifications or no update notifications. When a mediator view involves such a data source, a simple solution is to hold it virtual; any degree of materialization must be *self-maintainable* [2]. For hybrid views involving inactive sources, one must be careful when querying these sources to bring the virtual and materialized portion together; the consistency between the two portions is difficult to guarantee.

5.2. Performance Perspectives

To establish the performance of MJI, we must emphasize that MJI is a technique in the mediator context, which is a significantly different environment from centralized or distributed databases. In this context, data sources are highly autonomous. Many of the traditional query optimization techniques do not apply directly. This is essentially due to the fact that the mediator query processor may not even have a working cost model. For instance, an important technique for optimizing distributed joins is to use semi-joins. In the mediator context, we don't always know the values for various parameters that are necessary for deciding whether a semi-join is beneficial. The general model of mediator query optimization is an open research problem [5]. Until such a model is clearly defined, comparing MJI with traditional query processing techniques may not make much sense.

Given a N-way join view, V, we compare the operational and space cost of three strategies for holding V in the mediator: (1) virtual; (2) fully materialize; and (3) MJI-indexed. For space, virtual view is the most cost-effective, MJI the second and fully materialized views are the worst. For view maintenance, virtual views require no maintenance. The operational cost of maintaining a fully materialized view and that of maintaining MJIs of views are comparable. For query processing, fully materialized view performs the best since the query is processed entirely at the mediator. If V is virtual, unless a cost model is available in the mediator for multi-source queries, there is no promise for the performance of queries. If V is MJI-indexed, the number of "useless" tuples retrieved from the data sources is minimized with the overhead of shipping the keys of useful tuples ($\pi_{K_i}(MJI)$'s) to respective sources and that of query scrubbing. However, this overhead may be less than that of shipping data among sites for performing a N-way inter-database join without brutal-force or using semi-join based

techniques.

As shown above, we expect MJI to outperform the virtual view approach in terms of query processing speed and to outperform the fully materialized view approach in terms of space consumption.

6. Conclusions and Future Work

In this paper, we describe the mediator join index (MJI) that support fast N-way inter-database joins. A new family of algorithms, the Query Scrubbing Algorithms (QSA), are given for maintaining and evaluating queries with MJI. The QSA algorithm for index maintenance improves over previous algorithms given in [9] on termination performance. The QSA algorithm for query processing handles query processing in partially materialized views at the algorithm level.

As future research, we plan to do a comprehensive performance study of MJI. To establish MJI as an effective means for speeding up inter-database joins, it must be compared with other strategies for performing such joins. Currently, the parameters that affect the performance of inter-database join methods are not yet clear. Identifying these parameters will also be a useful step towards establishing a mediator query cost model.

References

[1] J. A. Blakeley, P. A. Larson, and F. W. Tompa. Efficiently Updating Materialized Views. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, pages 61–71, Washington,D.C., June 1986.

[2] A. Gupta, H. Jagadish, and I. Mumick. Data Integration Using Self-Maintainable Views. In *The Fifth International Conference on Extending Database Technology*, pages 140–144, Avignon, France, Mar. 1996.

[3] E. N. Hanson. A Performance Analysis of View Materialization Strategies. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, pages 440–453, 1987.

[4] R. Hull and G. Zhou. A Framework for Supporting Data Integration Using the Materialized and Virtual Approaches. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, pages 481–492, Montreal, May 1996.

[5] H. J. Lu, B. C. Ooi, and C. H. Goh. On Global Multidatabase Query Optimization. *ACM SIGMOD Record*, 21(4):6–11, Dec. 1992.

[6] P. Valduriez. Join Indices. *ACM Transactions on Database Systems*, 12(2):219–246, June 1987.

[7] G. Wiederhold. Mediators in the Architecture of Future Information Systems. *IEEE Computer*, pages 38–49, Mar. 1992.

[8] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View Maintenance in a Warehousing Environment. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, pages 316–327, San Jose, California, May 1995.

[9] Y. Zhuge, H. Garcia-Molina, and J. Wiener. The Strobe Algorithms for Multi-Source Warehouse Consistency. In *The Proceedings of the Fourth International Conference on Parallel and Distributed Information Systems (PDIS)*, Dec. 1996.