MEDWRAP: Consistent View Maintenance over Distributed Multi-Relation Sources*

Aparna S. Varde and Elke A. Rundensteiner

Department of Computer Science, Worcester Polytechnic Institute, Worcester, MA 01609
aparna@wpi.edu

Abstract. Data warehouses today extract information from several sources, each with multiple relations. Incremental View Maintenance (VM) of warehouses in such environments faces the problem of concurrency conflicts due to simultaneous relational updates occurring within and across these (semi-autonomous) sources. Existing VM algorithms only partially solve this issue. Some like ECA and CCA assume a single-source warehouse, while others like Strobe and SWEEP assume a multi-source environment with only one relation per source. However, in practice data sources have multiple relations in one schema. In this paper, we propose a solution called MEDWRAP that applies two-layered compensation. It resolves concurrency conflicts by using single-source compensation at each source wrapper and multi-source compensation at the mediator. We show that this achieves correct and consistent view maintenance. Not requiring intermediate views to be stored at the wrapper, MEDWRAP is space-efficient, a highly desirable feature, given the ever increasing size of modern warehouses.

1 Introduction

Data Warehouse Maintenance. A data warehouse is a materialized repository of integrated information based on the user’s needs [CD97]. The system that builds and maintains the warehouse is the Data Warehouse Management System (or DWMS) [Moh96]. Henceforth, we refer to the DWMS as the mediator. Due to the ever growing size of data warehouses, it is popular to maintain them incrementally [LSK01] when data updates occur. Several incremental View Maintenance (VM) algorithms like [ZGMHW95, AESY97] have been proposed. They compute the impact of a data update from one source on the warehouse content by sending maintenance queries [DZR99] to the other sources. Due to the autonomy of data sources, other updates may occur concurrently during the processing of a maintenance query and affect its result. This problem is called a concurrency conflict.

State-of-the-Art Solutions. Some VM algorithms like ECA [ZGMHW95] and CCA [Zhu99] solve this problem in a single data source. They utilize remote compensation queries to find out what joins with the conflicting tuple in each relation of that source. Other VM algorithms like Strobe [ZGMW96] and SWEEP [AESY97] allow relations to spread over multiple sources, restricting each data source to contain a single relation. [ZGMW96] handles data updates from different sources using remote compensation. [AESY97] adopts a local

* This work was supported in part by the NSF NYI grant #IRI 97-96264, the NSF CISE Instrumentation grant #IRIS 97-29878, and the NSF grant #IIS 9988776.
compensation technique that uses data contained in the update notifications received by the mediator to perform correction of maintenance errors. However the assumption that each data source contains only one relation is unrealistic since real-world data warehouses have sources with schemata composed of numerous relations. We need a solution to deal with concurrency conflicts in such environments.

**Proposed Approach.** In this paper, we propose the MEDWRAP solution that employs two compensation layers, one at the MEDiator of the system and another at the WRAPper of each data source. The wrapper effectively makes each source appear as if it were composed of one virtual relation without storing intermediate materialized views [GM96]. It compensates for concurrency conflicts within relations of that source. Thus a VM algorithm that works across multiple sources, assuming one relation per source, can now be used at the mediator. This compensates for concurrency conflicts occurring across sources. With this two-layered compensation however the same concurrent relational update in one source may cause a conflict in processing queries from both the wrapper and the mediator, posing an additional issue. This and other issues encountered in the design of MEDWRAP are discussed in the paper.

**Outline.** Motivating examples are outlined in Section 2. Section 3 presents the MEDWRAP approach and its architecture. Its design and algorithm appear in Section 4. Section 5 evaluates MEDWRAP and compares it with alternative solutions. Section 6 gives conclusions.

## 2 Motivating Examples

**Concurrency Conflict Problem.** Assume two data sources IS1 with relation R11 and IS2 with relations R21 and R22. The view at the mediator is V = R11 \\* (R21 \\* R22). If update \( \Delta R_{11} \) occurs at IS1, then an algorithm like [AESY97] would send the maintenance query \( \Delta R_{11} \times (R_{21} \times R_{22}) \) to source IS2. If meanwhile \( \Delta R_{21} \) had occurred concurrently causing a concurrency conflict, then the incorrect maintenance query result \( \Delta R_{11} \times ((R_{21} + \Delta R_{21}) \times R_{22}) \) would be returned. To compensate for the conflicting update \( \Delta R_{21} \), the mediator would need to subtract the value \( \Delta R_{11} \times (\Delta R_{21} \times R_{22}) \) from the incorrect query result. But the mediator cannot locally calculate this value since it does not have information about the extent of R22 in its update notifications. Hence a local compensation VM algorithm like [AESY97] would fail in this scenario.

**First Potential Solution.** For the above scenario, consider a wrapper at the source IS2 that could calculate the effect of any of its update first on its own relations, i.e. \((\Delta R_{21} \times R_{22})\) and send this computed update to the mediator, instead of just sending the pure one-relation update \( \Delta R_{21} \). However while the wrapper is doing the above computation, yet another concurrent update \( \Delta R_{22} \) may occur, causing a new conflict that cannot be corrected by the mediator. Thus this is not a feasible solution strategy to explore.

**Simplistic Approach.** Alternatively we propose a simple solution to our problem, henceforth called the simplistic approach. The mediator could treat
every relation of a source as a separate data source. For our example above, we would have the sources IS1, R1, IS2, R2, and IS2, R2. An algorithm like [AESY97] would then function as before, sending a separate maintenance query down to each and every now single-relation source, receiving its result, and compensating for concurrency conflicts at the mediator. However this is likely to pose tremendous overhead on the system. We would not take advantage of the query processing power of the DBMS engine of the data source in answering a multi-relation join query efficiently. In this paper we develop a practical solution that exploits the power of each data source for compensation to a maximal extent, achieving improved maintenance performance, without storage overhead.

3 The MEDWRAP Approach

MEDWRAP uses two-layered compensation. One VM algorithm is used by the mediator that maintains the materialized view at the data warehouse. The other VM algorithm is used at the wrapper dedicated to each source. The wrapper computes a source update for every relational update.

Definition 1: A source update \( \Delta IS \) corresponding to a relational update \( \Delta R \) in that source is the effective change of state in the source based on the view definition \( V \) at the data warehouse. The calculation of \( \Delta IS \) is done by using \( V \) at the IS to find out the effect of \( \Delta R \) on the portion of \( V \) relevant to this IS.

For example, consider IS1 with \( R_{11}[W, X] \) and IS2 with \( R_{21}[A, B], R_{22}[B, C] \), the view at the warehouse being \( V = \pi_W(R_{11}[W, X] \bowtie_X \pi_A(R_{21}[A, B] \bowtie R_{22}[B, C])) \). Rewriting this view definition in terms of sources rather than relations, we have \( V = \pi_W(IS_1) \bowtie_X IS_2 \). If in the single-relation source IS1, \( \Delta R_{11} \) occurs, then \( \Delta IS_1 = \pi_W(\Delta R_{11}) \), since attributes W and X are required in V. In the multi-relation source IS2, if \( \Delta R_{21} \) occurs, then \( \Delta IS_2 = \pi_A(\Delta R_{21} \bowtie R_{22}) \), since attribute A is required in V. Following the same logic as in [ZGMHW95] and related literature, duplicates are retained in the materialized view, by keeping a count of tuples. This enables deletions to be performed incrementally.

Note that the wrapper now effectively makes the source appear as one virtual relation, though without requiring any intermediate storage. Thus, we can employ maintenance strategies like [AESY97] that work across multiple sources, assuming a single relation per source, at the mediator. This is the core idea underlying MEDWRAP.

Similar to [DZR99, AESY97], we assume in MEDWRAP that (I) The communication between the wrapper and the mediator, and between the source and the wrapper is FIFO, and (II) Updates on base relations are inserts and deletes of tuples, a modify being represented as a delete followed by an insert.

In MEDWRAP, if we use an algorithm like [AESY97] at the mediator, then a Global Maintenance Query or GMQ is a query generated by the mediator in response to a source update, to find out what joins with this update in each of the other sources, based on the view definition at the data warehouse. This is essential to refresh the view based on the source update.
For example, in a system having $IS_1$ with $R_{11}[W,X]$, and $IS_2$ with $R_{21}[A,B]$ and $R_{22}[B,C]$, if the materialized view is $V = \pi_W(IS_1 \bowtie_{X=A} IS_2)$, then in response to source update $\Delta IS_1$, the mediator would send the global maintenance query $GMQ = \pi_W(\Delta IS_1 \bowtie_{X=A} IS_2)$ to source $IS_2$.

**Definition 2:** We define a source concurrency conflict as a source update that occurs during the processing of a global maintenance query, affecting the query result returned to the mediator.

In the above case, if $\Delta IS_2$ occurs during the processing of this GMQ, then it is a source concurrency conflict.

If an algorithm like [Zhu99] is used at the wrapper, then a Local Maintenance Query or LMQ is a query generated by the wrapper in response to a relational update in the source, to find out what joins with this update in the other relations of that source, based on the projection of the view definition of the mediator at this source. This is essential to generate a source update from a relational update.

In the example of the GMQ above, if $IS_2$ has $R_{21}[\phi, \phi]$ and $R_{22}[\phi, \phi]$, \footnote{\(\phi\) denotes null or empty} and $\Delta R_{21} = +[4,2]$ occurs, then $IS_2$ wrapper sends $LMQ = \pi_A(\Delta R_{21} \bowtie R_{22})$ to $IS_2$. It projects only relevant attribute $A$, since this is needed in computing $IS_1 \bowtie IS_2$ at the mediator.

**Definition 3:** We define a relational concurrency conflict as a relational update that occurs within a particular source, while a local or global maintenance query is being processed in that source, affecting the query result returned to the wrapper or mediator.

For example, if $\Delta R_{22} = +[2,5]$, occurs while the above LMQ is being processed, $IS_2$ returns an incorrect LMQ $=$ $+[4]$. $\Delta R_{22}$ is therefore a relational concurrency conflict. Note that this same $\Delta R_{22}$ can also affect the global maintenance query result or GMQR of a GMQ that is being answered by $IS_2$.

### 3.1 The Wrapper of MEDWRAP

We employ ideas from existing single-source VM algorithms for the MEDWRAP wrapper. VM algorithms were initially designed to maintain a view. Hence if we were to use an algorithm like [Zhu99] at the wrapper $W$, then it would generate its own local maintenance queries directed towards $IS_4$ in response to relational updates. However, it is not designed to answer global maintenance queries coming from outside the wrapper, in this case from the mediator. Hence we need additional query processing functionality in the wrapper. The simple method of accepting a global maintenance query from the mediator, directing it to the source, and sending results back to the mediator faces the same problem of relational concurrency conflicts that cannot be solved by the mediator.

We saw in the example of Definition 3 that the same relational update of a given data source could cause a conflict in both local update processing and global query processing in that respective wrapper. Hence, we now propose to employ one common processor for both query and update processing at each
data source. This is the **LProcessor** (Fig.1). More importantly, one common queue called **LQueue** is used for both global queries from the mediator and local update handling requests from the **IS**. If relational updates enter **LQueue** after a query and before its result, then we know that they could cause an incorrect query result. Hence they are identified as *relational concurrency conflicts*. **Fig.1** has the architecture of the MEDWRAP wrapper. **Fig.2** has its terminology.

![Fig.1. The MEDWRAP wrapper](image)

**Fig. 1.** The MEDWRAP wrapper

<table>
<thead>
<tr>
<th>Terminology</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>GMQ</td>
<td>Global Maintenance Query</td>
</tr>
<tr>
<td>LMQQ</td>
<td>Local Maintenance Query Request</td>
</tr>
<tr>
<td>CQR</td>
<td>Compensating Query Request</td>
</tr>
<tr>
<td>CQRR</td>
<td>Compensating Query Result</td>
</tr>
<tr>
<td>UIS</td>
<td>Source Update</td>
</tr>
<tr>
<td>LQueue</td>
<td>Queue in the Wrapper</td>
</tr>
<tr>
<td>LProcessor</td>
<td>Processor in the Wrapper</td>
</tr>
</tbody>
</table>

Following the example from Section 3, we demonstrate the working of the wrapper. The relational update $\Delta R_{21} = [+4, 2]$ is sent from the source to the wrapper and collected in **LQueue**. The **LProcessor** picks up the first message from the queue and starts processing. For example, if this is $\Delta R_{21}$, it sends $LMQ = \pi_A(\Delta R_{21} \bowtie R_{22})$ to $IS_2$. The source computes LMQR and returns it to **LQueue**. If concurrent update $\Delta R_{22}$ enters **LQueue** before the LMQR, it is identified as a *relational concurrency conflict*. **LProcessor** sends a *compensating query* in response to each conflict. A **Compensating Query or CQ** is a query generated by the wrapper in response to a relational concurrency conflict during local or global query processing, to find out what joins with the conflicting update in the other relations of that source, based on the projected view definition.

For example, here **LProcessor** would send $CQ_{22} = -\pi_A(\Delta R_{22} \bowtie R_{22})$ to $IS_2$. This returns a compensating query result $CQR_{22} = -\pi_A([-4, 2] \bowtie [2, 5]) = [-1]$. **LProcessor** then uses all query results to build a $\Delta IS$ corresponding to the $\Delta R$. Thus, $\Delta IS_2 = LMQR + CQR_{22} = [+4] - [4] = \emptyset$. This is the correct $\Delta IS_2$ in response to $\Delta R_{21} = [+4, 2]$, given that $R_{22}$ was empty when $\Delta R_{21}$ occurred. Note that since $\Delta R_{22}$ occurred after $\Delta R_{21}$, the LMQ for $\Delta R_{22}$ would now be answered using the new value of $R_{21}$ i.e. $[4, 2]$. Thus, the later updates incorporate the effect of those previously reported to the mediator.

Similarly, in global maintenance query processing, if there are relational concurrency conflicts while a GMQ is being answered by the source to give GMQR, this compensating query strategy can be used to get the correct GMQR. For details, refer to the additional example in Figure 6. The **LProcessor** sends either the $\Delta IS$ or the GMQR to the mediator.
**Theorem 1:** If the wrapper calculates $\Delta IS$ for each $\Delta R$, and GMQR for each GMQ using a compensation technique to remove the effect of concurrent $\Delta Rs$, then it propagates a consistent state of all the relations in that source at all times. (For proof refer to [VR02]).

**Theorem 2:** If the wrapper propagates a consistent state of all its relations, then each $\Delta IS$ and GMQR arriving from the source will have the effect of all the $\Delta Rs$ previously reported to the mediator from that source. (Proof in [VR02]).

### 3.2 The MEDWRAP System

The MEDWRAP system is shown in Fig.3 with its notations in Fig.4. The mediator maintains a materialized view $V$ at the data warehouse. A global processor $GProcessor$ performs the computations described below, and a global queue $GQueue$ stores messages coming from source wrappers.

**Fig. 3.** The MEDWRAP System

**Fig. 4.** Notations in MEDWRAP

On receiving a $\Delta IS$ from a source, the mediator sends a GMQ to each source to find out what joins with this update in the other sources. The GMQR arriving from each source is stored in $GQueue$. If any other source update arrives in $GQueue$ after a particular GMQ and before its GMQR, then the mediator identifies this as a *source concurrency conflict*. MEDWRAP uses the content of the update notifications in $GQueue$ to find out how this conflicting source update joins with the other sources, and performs compensation locally, using an algorithm like [AESY97]. Note that this now works, since we have overcome the problem of *relational concurrency conflicts* due to the compensation-based wrapper at each source. The GMQR for that source generated after compensation is consistent with respect to the current source state. Likewise, the mediator uses the values of GMQRs from each source to build a final view refresh $\Delta V$. It uses this $\Delta V$ to update the materialized view. (Detailed example in Figure 6.)

**Theorem 3:** If the wrapper sends each $\Delta IS$ and GMQR to the mediator in the order that the later ones incorporate the effects of all the previously reported $\Delta Rs$, then the mediator can correctly maintain the data warehouse. (For proof refer to [VR02]).
Theorem 4: $\Delta V$ computed by the mediator from $\Delta IS$ in MEDWRAP is identical to $\Delta V$ calculated directly from $\Delta R$ in the simplistic approach explained in Section 2. (Proof in [VR02].)

Lemma 1: The materialized view at the data warehouse will be maintained completely consistent [ZGMW97] with each source state, if the VM algorithms used at the wrapper and the mediator in MEDWRAP individually achieve complete consistency [ZGMW97]. If either algorithm achieves only strong consistency [ZGMW97], then MEDWRAP does VM with strong consistency.

4 Design of MEDWRAP

Fig.5 gives the MEDWRAP algorithm, with a detailed example in Fig.6. MEDWRAP is designed using techniques similar to [AESY97, Zhu99]. Process $S$-up in the source appends a $\Delta R$ from the source to $L$Queue in the wrapper. $L$-Processor gets the first message from the $L$Queue. The $Wrap$-up function in $L$Processor generates an LMQ if the message is a $\Delta R$, and sends it to S-que in the source. S-que can process LMQs and GMQs. It computes the query result and appends it to $L$Queue. LProcessor gets the LMQ. Its function $Wrap$-ans collects the LMQs in case of local update processing and GMQRs in case of global query processing. In each case, it sends CQs if there are relational concurrency conflicts. It uses all query results to build the $\Delta IS$ or the GMQR, and appends it to $G$Queue in the mediator. GProcessor gets the first message from the $G$Queue. If the message is a $\Delta IS$, then it sends a GMQ to S-que in each source, one at a time. If the message is a GMQR, it locally compensates for source conflicts, if any. It finally computes the view change $\Delta V$ corresponding to $\Delta IS$ using all GMQRs after compensation. It then uses this $\Delta V$ to update the materialized view V at the warehouse.

$L$Queue and $G$Queue do implicit time-stamping because these queues store messages in their order of arrival, thus order of maintenance handling, and detect concurrency conflicts by the method outlined above. Thus MEDWRAP does not require explicit clocks as in [DZR99], or versions of transactions/tuples as in [CCR, KM99].

5 Comparative Evaluation

We compare MEDWRAP with two other approaches. One, the materialized Multi Relation Encapsulation Wrapper (MRE) [DZR99] that stores a source view (ISV) at each wrapper, analogous to a materialized view (MV) at the mediator. Two, the simplistic approach of Section 2 that treats every relation as a separate source if there are multiple relations per source. MEDWRAP, MRE and the simplistic approach can all make use of an algorithm like [AESY97] at the mediator. MEDWRAP is inspired by [DZR99], earlier work at WPI.

Space Cost Model. MEDWRAP and the simplistic approach store no views at wrappers. MRE stores one view (ISV) at each wrapper.
AT SOURCE

1. PROCESS R-update
   LOOP
   EXECUTE (ΔRq0);
   /* Rq0 is relational view of ΔR */
   APPEND (ΔRq0) TO LQuery;
   FOREVER

2. PROCESS Sq(q);  
   LET ΔR = Current semantic update;
   /* Q is a SQM, GMQ or CQ */
   LET Qr := Q[ΔR];
   /* every result as per current source state */
   IF (Q is a CQ) THEN WrapCQ(Qr);
   ELSE APPEND (q,ΔR) TO LQuery;

AT WRAPPER

1. LQuery:
   LET o = φ;  /* query initially empty */
   LET msg =trigger SQUpdate, Q message in LQuery ;
   /* ΔR is a relational update */
   LET distComplete(ΔR) = φ /* no conflict assumed */
   IF (ΔR, LQuery = After Q AND BEFORE QR) 
   THEN distComplete(ΔR) = true /* conflict occurs */

2. PROCESS LProcessor:
   LOOP
   GET (msg) FROM LQuery, loc;
   IF (msg = ΔR) THEN DO
   TempView := ΔR;
   FOR EACH (E in LMQR) DO
   /* re-apply T in DB session */
   GMQr := TempView, V ΔR;
   q :=GMQr, Q send GMQr to Sq of ΔR */
   ENDIF, /* if good δ */
   ELSE /* if message is δ */
   IF (distComplete(ΔR = true) THEN DO
   /* makes source concurrency-conflicts by local compensation */
   GMQr := GMQr, ΔR := ΔR, V TempView;
   FOREVER
   ENDIF, /* if compensation */
   IF (distComplete(ΔR = true) THEN DO
   /* makes source concurrency-conflicts by local compensation */
   GMQr := GMQr, ΔR := ΔR, V TempView;
   FOREVER
   ENDIF, /* if compensation */
   /* V := V - δ */
   ELSE /* if message is GMQr */
   ENDIF, /* if compensation */
   Else /* if message is GMQr */
   ENDIF, /* if compensation */

Fig. 5. The MEDWRAP algorithm

Space Cost:

<table>
<thead>
<tr>
<th>Approach</th>
<th>Total Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEDWRAP</td>
<td>MV + 0 = MV</td>
</tr>
<tr>
<td>MRE</td>
<td>MV + N = ISV</td>
</tr>
<tr>
<td>Simpistic</td>
<td>MV + 0 = MV</td>
</tr>
</tbody>
</table>

Time Cost Analysis. When a relational update occurs, the time taken by MEDWRAP to generate a corresponding view update is the sum of the following times: wrapper of one source generating ΔIS for ΔR (compensating for relational concurrency conflicts if any), wrapper sending ΔIS to mediator, mediator in response sending a GMQ each to the other (N-1) wrappers, wrappers processing to give (N-1) GMQRs (compensating for relational concurrency conflicts if any), wrappers sending (N-1) GMQRs to mediator, and mediator processing after each GMQR with or without source conflicts.

On similar lines, we analyze MRE and the simplistic approach. The delays in processing are wrapper delay T_w, mediator delay T_m, I/O delay T_I and CPU delay T_c. Ignoring T_c, since it is negligible compared to the other delays, we tabulate results.

Time Cost:

<table>
<thead>
<tr>
<th>Approach</th>
<th>Time per Relational Update</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEDWRAP</td>
<td>2N + 6X + 1 + Tw + [2N - 1] * T_m + (N + 2)X + T_i</td>
</tr>
<tr>
<td>MRE</td>
<td>2N + 3X + 1 + Tw + [2N - 1] * T_m + (N + 1)X + T_i</td>
</tr>
<tr>
<td>Simpistic</td>
<td>2N + 1 + Tw + (2Z - 1) + Tw + 2Z + T_i</td>
</tr>
</tbody>
</table>
Assume IS1, IS2, IS3 as tabulated, and the materialized view (MV) at the mediator: V = πW(IS1 ∩ M1W, IS2 ∩ M2W, IS3). Thus IS1 = ππ(R1), IS2 = πW(R2 ∩ R2 ∩ R2), IS3 = ππ(R0 ∩ R0). Initially V = φ, ΔV = φ. Given a ΔR the following steps calculate ΔV:
1. ΔR1 = [5, 4], occurs in IS1. IS1 has only one relation, so ΔIS1 = ππ(ΔR1) = [4].
2. IS1 wrapper sends ΔIS1 = [4] to GQueue.
4. Assume LQueue of IS2 has messages in the order: ΔR21, GMQR2, ΔR22, GMQR2. i. At IS2: S-Queue calculates GMQR2 := [1, 4], based on current source state.
ii. S-Queue appends GMQR2 := [1, 4] to LQueue and GProcessor gets it.
iii. Wrapars in LProcessor detects relational concurrency conflicts ΔR22, ΔR23 since they are in LQueue after GMQR2 and before GMQR2. Wrapars send CQR0 := πR2 ∩ R2 ∩ R2 to S-Queue.
iv. S-Queue receives CQR0 := [1, 4] and sends it to Wrapars.
v. Wrapars calculates GMQR2 := GMQR2 + all CQR0 = [1, 4] + [1, 4] = [2, 4]. So, GMQR2 := [2, 4]. Wrapars sends GMQR2 := [2, 4] to GQueue.
6. Similarly, IS2 wrapper independently calculates ΔIS2 := ππ(ΔR21 ∩ R2 ∩ R2) := ππ([4, 2] ∩ φ ∩ φ) := φ and sends it to GQueue. This is ΔIS2 due to ΔR21.
7. Assume GQueue of mediator has messages in the order: ΔIS1, GMQR2, ΔIS2, GMQR2. The GProcessor detects concurrency conflict ΔIS2 since it is in the queue before GMQR2. GProcessor applies compensation to get GMQR3 := GMQR2 − ΔIS2 ∧ TempView = φ.
8. GProcessor updates TempView := GMQR3 = φ.
9. ΔV is used to update the materialized view at the mediator as: V = V + ΔV = V + φ = V.

Fig. 6. Example for the working of MEDWRAP
provides a good space/time trade-off for the view maintenance of multi-relation multi-source data warehousing systems.

6 Conclusions

MEDWRAP provides a consistent solution to incremental view maintenance in distributed multi-source data warehousing environments with multiple relations per source. MEDWRAP offers the flexibility of semi-autonomous sources since sources do not participate in view maintenance beyond reporting updates and processing queries, the benefit of software re-use by adopting techniques from existing algorithms, and the advantage of storage efficiency since no intermediate materialized views are needed at wrappers. Implementation of MEDWRAP and its integration with the existing data warehousing system DyDa at WPI [CZC+01, LNR01] are ongoing to allow for experimentation.

References

[ZGMW97] Y. Zhuge, J.Wiener et. al. Multiple View Consistency for Data Warehousing. In IEEE Intl. Conf. on Data Eng-97, p.289-300.