DyDa: Dynamic Data Warehouse Maintenance in a Fully Concurrent Environment ⋆

Xin Zhang and Elke A. Rundensteiner

xinz | rundenst@cs.wpi.edu
Department of Computer Science, Worcester Polytechnic Institute, Worcester, MA 01609

Abstract. Data warehouses (DW) are an emerging technology to support high-level decision making by gathering information from several distributed information sources (ISs) into one materialized repository. In dynamic environments such as the web, DWs must be maintained in order to stay up-to-date. Recent maintenance algorithms tackle this problem of DW management under *concurrent data updates* (DU), whereas the EVE system is the first to handle (*non-concurrent schema changes*) (SC) of ISs. However, the concurrency of schema changes by different ISs as well as the concurrency of interleaved SC and DU still remain unexplored problems. In this paper, we propose a solution framework called DyDa that successfully addresses both problems. The DyDa framework detects concurrent SCs by the broken query scheme and conflicting concurrent DUs by a local timestamp scheme. The two-layered architecture of the DyDa framework separates the concerns for concurrent DU and concurrent SC handling without imposing any restrictions on the autonomy nor on the concurrent execution of the ISs. This DyDa solution is currently being implemented within the EVE data warehousing system.

Keywords: Data Warehousing, View Maintenance, Data Updates and Schema Changes, Concurrency.

1 Introduction of Data Warehousing

Data warehouses (DW) are built by gathering information from several autonomous Information Sources (ISs) and integrating it into one virtual repository. Data warehousing [6] has importance for many applications in network environments, such as travel services, E-commerce, decision support systems, and other web related applications. An important consequence of the autonomy of sources is the fact that those sources may change without being controlled from a higher data integration layer.

⋆ This work was supported in part by several grants from NSF, namely, the NSF NYI grant #IRI 97–96264, the NSF CISE Instrumentation grant #IRIS 97–29878, and the NSF grant #IIS 97–32897. Dr. Rundensteiner would like to thank our industrial sponsors, in particular, IBM for the IBM partnership award, and GTE for partial support of Xin Zhang.

1.1 DW Management under Fully Concurrent Environment

Many sources, particularly WWW-based data sources, change not only their data, but also their schema or query interface without cooperating with users of their data. Even conventional databases typically undergo schema changes due to many reasons, including application requirement changes, corrections of schema design errors, upgrades, etc. Case studies both by [11] for health management applications and [5] for different business type of application reveal that schema changes are inevitable not only during the development of a project but also once a project has become operational. Even seemingly stable data sources that appear to be only undergoing data updates may indirectly cause schema changes to be propagated to the DW via these wrappers, e.g, SchemaSQL [3]. Therefore, any data warehouse has to take the possibility of IS schema changes into account.

Furthermore, since ISs are autonomous, schema changes can occur concurrently and can be mixed arbitrarily with data updates from other ISs. Hence, data warehouses built on top of those ISs have to meaningfully handle these updates if their views are to survive over a period of time.

There are three types of tasks related to DW management in distributed environments. The most heavily studied area is the incremental maintenance of materialized views under distributed ISs. Such view maintenance (VM) algorithms [12, 1, 16, 15] maintain the extent of the data warehouse whenever a data update (DU) occurs at the IS space. The second area called view synchronization (VS) [9, 10] is concerned with evolving the view definition itself whenever there is a schema change (SC) of one of the ISs that results in a view definition to become undefined. The third area, referred to as view adaptation (VA) [2, 6, 7], adapts the view extent incrementally after the view definition has been changed either directly by the user or indirectly by a view synchronization module.

Materialized view maintenance (VM) is the only area among those three that thus far has given attention to the problem of *concurrency of (data) updates* at ISs [12, 1, 16, 15]. Our recent work on Coop-SDCC [14] is the first to study the concurrency problem of both data updates and schema changes in such environments. The Coop-SDCC approach integrates existing algorithms designed to address the three DW management subproblems VM, VS and VA into one system by providing a protocol that all ISs must abide by and that as consequence enables them to correctly co-exist and collaborate. In that solution, the SDCC system *locks* the ISs to wait for the mediator to accomplish the maintenance of the data warehouse before admitting a SC to be executed at that IS and thus propagated up to the DW. This solution has however the limitation of requiring information sources to *cooperate* by first announcing an intended SC, then waiting for the DW to finish servicing any outstanding requests, before being permitted to execute the schema update at the IS.

1.2 Our Approach

In this paper, we overcome the limitation of this previous solution. We now propose the Dynamic Data warehouse (DyDa) framework that can handle fully concurrent SCs and DUs without putting any restriction on the timing of when an SC is allowed to take place nor requiring any cooperation from the ISs.

Since we release the restriction of cooperative ISs, during the process of managing a data warehouse, query requests that are sent down to different ISs [1, 15], new updates (DU or SC) could occur concurrently at some of the ISs. We call such updates *concurrent updates*. A concurrent *DU* will result in an *incorrect query result* returned by an IS [14, 8], whereas as a concurrent SC results in a *broken query* that cannot be processed by the ISs, i.e., an error message.

In this work, we provide the first complete solution called DyDa with:

– We characterize the problem of maintenance under concurrent SCs, which we call the broken-query problem.
– We devise a strategy for the detection of concurrent SCs based on the broken query concept, that identifies the causes of the conflict.
– We analyze selected VM, VS and VA algorithms from the literature to determine if and how they are affected by concurrent SCs, and thus elicit their properties required to function correctly in this dynamic environment.
– We introduce the overall solution framework DyDa that adapts a two-layered architecture that separates the concerns for concurrent DU and concurrent SC handling without imposing any restrictions on the fully concurrent execution of the ISs.

2 Problem Definition

Background on Schema and View Definition Changes. We have to first understand how DW management algorithms, i.e., view maintenance (VM), view synchronization (VS), and view adaptation (VA), work together (see the Figure 1). The VS algorithm will update the view definitions affected by schema changes (SC) at the IS. The output of VS will be a set of view definition changes (VDCs), which are used to update the affected view definitions. Then the VA algorithm will adapt the extent of the rewritten views based on the set of VDCs. At the same time, VM will continuously update the view extent based on data updates (DU) that independently may occur at different ISs.

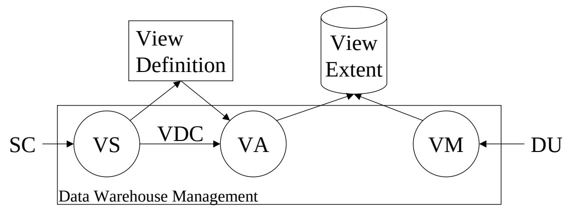

Fig. 1. Relationship between VM, VS and VA Algorithms

Definition of the Maintenance-Concurrent Problem. While the concept of maintenance concurrency has previously been defined for data updates only [12], we now extend it to incorporate schema changes. In the following we assume the views are defined in SPJ queries and the connection between each IS and DW is FIFO.

Figure 2 defines the main notations we use. A sequential number n, unique for each update, will be generated by DyDa whenever the IS update message reaches the system.

Definition 1. *Let $X(n)[j]$ and $Y(m)[i]$ denote either DUs or SCs on IS[j] and IS[i] respectively. We say that the update X(n)[j] is* **maintenance-concurrent** *(in short* **concurrent**) *with the update Y(m)[i], denoted X(n)[j] $\vdash$ Y(m)[i], iff: i) $m < n$, and ii) X(n)[j] is received at the DW* **before** *the answer QR(m)[j] of update Y(m)[i] is received at DW.*

Notation	Meaning
IS[i]	Information source with subscript i.
X(n)[i]	X is update (SC or DU) from IS[i] at sequence number n.
Q(n)	Query used to handle update X(n)[i].
Q(n)[i]	Sub-query of Q(n) sent to IS[i].
QR(n)[i]	Query result of Q(n)[i].
QR(n)	Query result of Q(n) after re-assembly of all QR(n)[i] for all i.

Fig. 2. Notations and Their Meanings.

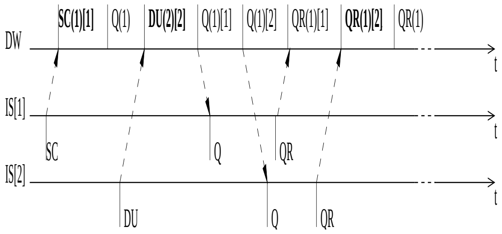

Fig. 3. Time Line for a Maintenance Concurrent Data Update.

Figure 3 illustrates the concept of a **maintenance-concurrent** update defined in Definition 1 with a time line. Messages only get time stamps assigned upon their arrival at the DW layer. That means that the **maintenance-concurrent** update is defined with respect to the data warehouse layer instead of the IS layer. Assume we have one DW and two IS[1] and IS[2]. First, there is a schema change SC at IS[1]. Then, there is a data update DU at IS[2]. From the figure, we can see that the SC is received by the DW before the DU, but DU occurs at IS[2] before the adaptation query Q(1)[2] of SC arrives at IS[2], that is, DU occurs before the query result QR(1)[2] arrives at the DW. So, here the DU is maintenance concurrent with SC by Definition 1.

There are four types of **maintenance-concurrent** updates listed in Table 1 in the order of the easiest to the hardest in terms of handling them.

Definition 2. *A query is called a* **broken query** *if it cannot be processed by an IS because the schema of the IS expected by the query is not consistent with the actual schema of the IS encountered during query processing.*

Type	Meaning A **maintenance-concurrent** DU happened	Denoted By
I	when handling DU	DU_h – DU_{mac}
II	when handling SC	SC_h – DU_{mac}
III	when handling DU	DU_h – SC_{mac}
IV	when handling SC	SC_h – SC_{mac}

DU_h (or SC_h) denotes the DU (or SC) that is currently being handling by the DyDa system. DU_{mac} (or SC_{mac}) denotes the DU (or SC) that is a **maintenance-concurrent** DU (or SC).

Table 1. Four Types of Maintenance-Concurrent Updates.

We distinguish between three types of broken queries based on the type of SC causing the problem as well as the characteristics of the IS space available for the system to deal with this problem.

- **Type 1: Broken Queries caused by a RenameSC.** In this case the data is still there but only the interface of this IS relation has changed. We can use a name mapping strategy to get to the desired data.
- **Type 2: Broken Query caused by a DropSC with Replacement.** While the data has removed from the IS, we are able to find the required information from an alternate source that holds duplicate data.
- **Type 3: Broken Query caused by DropSC without Replacement.** The data has really been dropped from the IS, and the system is not able to identify an alternate source for the data.

AddSC will not result in broken queries as they do not interfere with the interpretation of any existing query. A broken query as defined in Definition 2 will be returned by the respective IS as an empty query result with a "broken query" error message.

3 The DyDa Framework

3.1 Overall Solution Architecture

The DyDa framework is divided into three spaces: DW space, middle space, and IS space as described in Figure 4, which depicts the modules of our proposed framework and the data flow between them. The DW space houses the extent of the DW. It receives queries from the middle space bundled together with the data to update the DW. The IS space is composed of ISs and their corresponding wrappers. Wrappers are responsible for translating queries, returning query results, and sending notifications of DUs and SCs of the ISs to the middle space.

The middle space is the integrator of the DyDa framework. It can be divided into two subspaces. The higher-level subspace is called the **DW management subspace**. All the DW management algorithms, like VS, VA and VM, are located in this subspace. The lower-level subspace is called the **query engine**

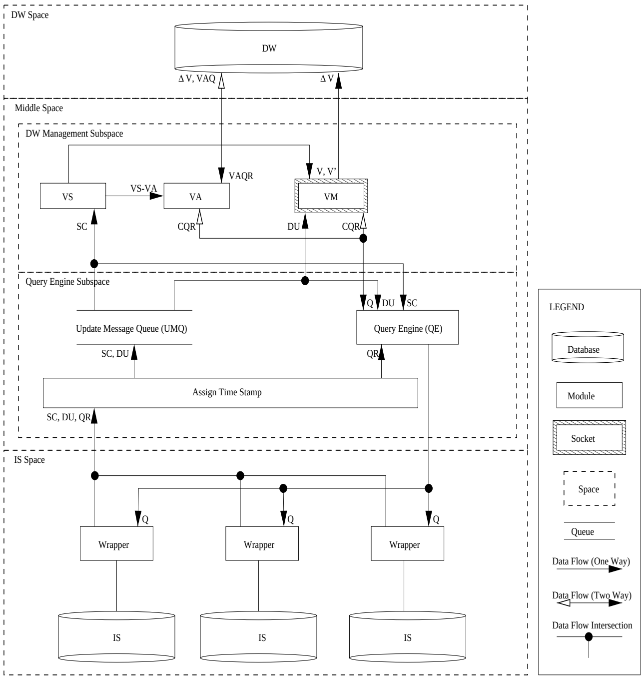

Fig. 4. Architecture of DyDa Framework

Meaning of Symbols Used in Figure 4.

Symbols	Meaning
V	View definition affected by either Schema Change (SC) or Data Update (DU).
V'	Evolved view definition of affected view V.
DW	Data warehouse.
ΔV	Incremental view extent of data warehouse, i.e., set of tuples to be inserted into or removed from the extent of view V.
CQR	Query result that is returned by QE.
VS-VA	All information VA module requires from VS module for view adaptation. It includes: V, V', Meta-knowledge, Synchronization-mapping.
VAQ	View Adaptation Query.
VAQR	View Adaptation Query Result.

subspace. This subspace is composed of the Query Engine (QE) and its supporting modules, namely, the Update Message Queue (UMQ) module and Assign Time Stamp module. The UMQ module captures all updates received, and the Assign Time Stamp module is used to assign a time stamp to each incoming message, i.e., SC, DU and query result, based on their arrival. The two subspaces effectively correspond to two different levels of concurrency control. The key idea is that **maintenance-concurrent** DUs will be handled locally by the QE module at the lower-level of the middle space, so that DW management algorithms at the upper-level, such as VS, VM and VA, are shielded from and will never be aware of any **maintenance-concurrent** DUs. While our associated technical report [13] gives more details on the dynamicity and protocol of the system, we now discuss the justification of the two-layered concurrency control in Section 4.

4 Two Levels of Concurrency Control of DyDa

While the detection of concurrent DUs is based on the local timestamp assigned to DUs upon their arrival at the DW [12], the detection of concurrent SCs is based on identifying when and why a submitted query is broken and hence returned unanswered.

In the DyDa framework in order to separate out the handling of concurrent data updates and schema changes, there are two levels of concurrency control corresponding to the two sub-spaces of the *middle* space. At the lower level of concurrency control (the *query engine subspace*), the concurrent DUs as well as the concurrent SCs of the type **RenameSC** will be handled by the QE. The *DW management subspace* supports the higher level of concurrency control. The management algorithms (e.g., VM, VS and VA) at that subspace cooperate with each other in order to handle schema changes of the type **DropSC**. **AddSC** SCs do not render view definitions undefined and hence do not affect VM, VS nor VA. Thus they do not need to be handled by DyDa.

4.1 Low Level Concurrency Control at QE Subspace

DyDa solves the type *I* and *II* **maintenance-concurrent** problems (see Table 1) at the QE level by the local correction strategy and part of type *III* and *IV* **maintenance-concurrent** problems (see Table 1) by a local naming mapping strategy.

Using Local Correction to Handle Maintenance-Concurrent DUs. All queries from the DW modules down to the IS space will first go through the QE module (Figure 4). This includes incremental view maintenance queries, view adaptation queries, or view recomputation queries. Given that all three query types are "extent-related queries", the query engine will use the local correction (LC) algorithm described in [14] to successfully fix all side effects of concurrent DUs on these queries before passing the corresponding query results up to the next layer of the system. This results in the concurrent DUs to be transparent to all modules in *DW management* subspace. By using the local correction algorithm, there is no possibility to be faced with an infinite wait due to the recursive correction of queries, as is a problem in the Strobe solution [16]. Details of the local correction scheme we adopt for DyDa and its proof of correctness are beyond the scope (and space limitations) of this paper, but can be found in [12].

Using Name Mapping to Handle Maintenance-Concurrent RenameSCs In addition to the above service, features based on a temporary name mapping table have been added to the query engine of DyDa in order to handle concurrent SCs of type **RenameSC**. Whenever there is a query result received, the QE will check whether there is a concurrent renameSC received in the UMQ. If there is no concurrent renameSC, QE will directly return the result. If there is one concurrent renameSC that makes the query broken, QE will rename the query based on the renameSC and resent it again until succeed and keep the renaming transparent to the higher layer. More comprehensive discussion of handling renameSC at QE can be found in [13].

If more than one SC is found in the UMQ, and at least one of them is not RenameSC (i.e., at least one DropSC exists), then the QE cannot handle the broken query anymore. It will report an error message to the *data warehouse maintenance* subspace. If the modified query based on the RenameSC breaks again due to another RenameSC in the UMQ, the QE module will modify the query and resubmit it until either the query succeeds, or a DropSC is encountered in the UMQ. In the highly unlikely case that the ISs were to continuously issue new RenameSCs, then the QE would have to endure a long processing time.

4.2 High Level Concurrency Control at DW Management Subspace

Finally, the VS, VM and VA modules in the *DW management subspace* finally need to handle the concurrent DropSC problem. VS has the least effect of the concurrent DropSCs, since it will not send queries to QE as shown below. VM module is affected by the concurrent DropSCs, but doesn't have the capability to handle them as shown below. VA module is the module designated to handling the concurrent DropSCs as shown in [13].

The VS Module and Maintenance-Concurrent SCs. From Definition 1, we know that VS will never have any trouble with concurrent SCs, because it will not send any queries down to the IS space. However, the VS module needs to provide information to the VA module to help VA to adapt the view extent under such concurrent SCs (Figure 4). From the state diagram (Figure 1) in Section 2, we can see that for every SC, the VS module will be called to generate a new version of the view definition to account for this SC.

From the point of view of VS, all SCs happen sequentially. If two SCs come to the *middle* space at the same time, they will be assigned a random handling order.[1] In a distributed environment as we are assuming, there is no convenient way to determine when the two SCs happened at the IS space relative to each other (unless they both come from the same IS). Plus they may indeed have truly occured at the same time. The issue of which of the SCs happened first for two different SCs from two autonomous ISs does not affect the correctness of the VS.

As further detailed in [13] after we apply *VS*, the *VA* module will know how each view definition evolved and thus is capable of generating the corresponding view adaptation queries for it. If *VS* drops a view as being no longer salvageable, i.e., empty extent, then *VA* doesn't need to adapt it.

The VM Module and Maintenance-Concurrent SCs. If VM encounters a concurrent SC, it recognizes that by the fact that it will receive a broken query passed up from the QE module. Because the VM algorithms in the literature haven't considered thus far how to handle broken queries, we put the responsibility of this handling on the VA module that needs to be redesigned anyway.

[1] If two SC changes come to the middle space of DyDa framework at same time, we will pick a preferred order of handling the SCs based on a quality-cost model described in [4]. The decision of the handling order of a set of SCs is out of the scope of this paper.

Thus in our system existing VM algorithms from the literature are kept intact. Hence the VM algorithm simply stops (abnormally) and sends the DU that it is currently handling to the VA module. That DU will then be handled later by the VA algorithm when it is adapting the extent of the view as discussed in detail in [13]. Given that we isolate the VM module from the handling of **maintenance-concurrent** SCs, all the VM algorithms, e.g., PSWEEP [12], SWEEP [1], Strobe [16], and ECA [15] could be used in DyDa.

4.3 Effect of Maintenance-Concurrent Updates on Existing DW Management Algorithms

In the DyDa framework, we decide to let the query engine (QE) module as explained in Section 3 fix the problem of any **maintenance-concurrent** DU before the query results reach the DW management modules. So, the **maintenance-concurrent DUs** have no effect on the three modules VA, VM and VS. However, the three modules have a different degree of awareness of **maintenance-concurrent SCs**, namely:

- There is no concept of **maintenance-concurrent** SCs for the VS module, because the VS module never sends any query down to the IS space.
- While the VM module will send queries down to the IS space for view maintenance, it assumes the view definition that it is maintaining will not be changed. So if a maintenance query is broken by a **maintenance-concurrent** SC, the VM module has to be reset by the DyDa system so to work with the newly updated view definition that has been generated by VS to take care of that **maintenance-concurrent** SC. VM itself however can function unchanged in our environment.
- The VA module also sends down queries to the IS space to adapt the view extent. If a view adaptation query is broken by a **maintenance-concurrent** SC, then VA needs to handle this in its adaptation process. Hence, a completely new VA algorithm has been designed for DyDa [13].

5 Conclusions

In this paper, we first identify the broken query problem of DW management under concurrent DUs and SCs. Then, we propose the DyDa solution framework that solves the problem using a two layered architecture as presented in the paper. To our knowledge, our work is the first to address the data warehouse maintenance problem under fully concurrent DUs and SCs of ISs. DyDa overcomes the limitation of the only previous approach by dropping its restrictive assumption of cooperation of the ISs [14] We are in the process of implementing this DyDa data warehouse solution In the future, we plan to run experimental studies to assess performance aspects of this solution.

Acknowledgments. We would like to thank all DSRG members, in particular, A. Koeller and Y. Li for helping to implement the EVE system.

References

[1] D. Agrawal, A. E. Abbadi, A. Singh, and T. Yurek. Efficient View Maintenance at Data Warehouses. In *Proceedings of SIGMOD*, pages 417–427, 1997.

[2] A. Gupta, I. Mumick, and K. Ross. Adapting Materialized Views after Redefinition. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 211–222, 1995.

[3] L. V. S. Lakshmanan, F. Sadri, and I. N. Subramanian. SchemaSQL — A Language for Interoperability in Relational Multi-database Systems. In T. M. Vijayaraman et al., editors, *Proceedings of the twenty-second international Conference on Very Large Data Bases*, pages 239–250, Mumbai, India, Sept. 1996.

[4] A. J. Lee, A. Koeller, A. Nica, and E. A. Rundensteiner. Non-Equivalent Query Rewritings. In *International Database Conference*, Hong Kong, July 1999.

[5] S. Marche. Measuring the Stability of Data Models. *European Journal of Information Systems*, 2(1):37–47, 1993.

[6] M. Mohania and G. Dong. Algorithms for Adapting Materialized Views in Data Warehouses. *International Symposium on Cooperative Database Systems for Advanced Applications*, December 1996.

[7] A. Nica and E. A. Rundensteiner. View Maintenance after View Synchronization. In *International Database Engineering and Applications Symposium (IDEAS'99)*, August, Montreal, Canada 1999.

[8] D. Quass and J. Widom. On-Line Warehouse View Maintenance. In *Proceedings of SIGMOD*, pages 393–400, 1997.

[9] E. A. Rundensteiner, A. Koeller, and X. Zhang. Maintaining data warehouses over changing information sources. *Communications of the ACM*, June 2000.

[10] E. A. Rundensteiner, A. Koeller, X. Zhang, A. Lee, A. Nica, A. VanWyk, and Y. Li. Evolvable View Environment. In *Proceedings of SIGMOD'99 Demo Session*, pages 553–555, May/June 1999.

[11] D. Sjoberg. Quantifying Schema Evolution. *Information and Software Technology*, 35(1):35–54, January 1993.

[12] X. Zhang, L. Ding, and E. A. Rundensteiner. PSWEEP: Parallel View Maintenance Under Concurrent Data Updates of Distributed Sources. Technical Report WPI-CS-TR-99-14, Worcester Polytechnic Institute, Dept. of Computer Science, May 1999.

[13] X. Zhang and E. A. Rundensteiner. DyDa: Dynamic Data Warehouse Maintenance in a Fully Concurrent Environment. Technical Report WPI-CS-TR-99-20, Worcester Polytechnic Institute, Dept. of Computer Science, July 1999.

[14] X. Zhang and E. A. Rundensteiner. The SDCC Framework for Integrating Existing Algorithms for Diverse Data Warehouse Maintenance Tasks. In *International Database Engineering and Application Symposium*, pages 206–214, Montreal, Canada, August, 1999.

[15] Y. Zhuge, H. García-Molina, J. Hammer, and J. Widom. View Maintenance in a Warehousing Environment. In *Proceedings of SIGMOD*, pages 316–327, May 1995.

[16] Y. Zhuge, H. García-Molina, and J. L. Wiener. The Strobe Algorithms for Multi-Source Warehouse Consistency. In *International Conference on Parallel and Distributed Information Systems*, pages 146–157, December 1996.