Data Integration Using DataPile Structure

David Bednárek, David Obdržálek, Jakub Yaghob, and Filip Zavoral

Department of Software Engineering
Faculty of Mathematics and Physics, Charles University Prague
{david.bednarek, david.obdrzalek, jakub.yaghob,
filip.zavoral}@mff.cuni.cz

Abstract. One of the areas of data integration covers systems that maintain coherence among a heterogeneous set of databases. Such a system repeatedly collects data from the local databases, synchronizes them, and pushes the updates back.
One of the key problems in this architecture is the conflict resolution. When data in a less relevant data source changes, it should not cause any data change in a store with higher relevancy.
To meet such requirements, we propose a DataPile structure with following main advantages: effective storage of historical versions of data, straightforward adaptation to global schema changes, separation of data conversion and replication logic, simple implementation of data relevance.
Key usage of such mechanisms is in projects with following traits or requirements: integration of heterogeneous data from sources with different reliability, data coherence of databases whose schema differs, data changes are performed on local databases and minimal load on the central database.

1 Introduction

The concept of data integration covers many different areas of application [3,13]. In this paper, we focus on one kind of applications characterized by the following requirements:

- Data warehousing: The data originated at the local data sources should be replicated into a central repository (data warehouse) in order to allow efficient analytical processing and querying the central system independently of local systems.
- Back-propagation: Any update which occurs in a local database (performed by its local application) should be distributed to other local databases for which this kind of data is relevant.
- History records: The central repository should maintain full history of all data stored therein.

Each one of the requirements forms a well-known problem having well-known solutions [2,8,9,10]; nevertheless, combining the requirements together introduces new, interesting problems, and disqualifies many of the traditional solutions. This paper presents a technique, called DataPile, which combines flexible storage technology

(built upon a standard relational database system) with system architecture that separates the replication mechanisms from the schema-matching and data-conversion logic. Since the approach is inspired by XML techniques rather than relational databases, its combination with modern XML-based technologies is straightforward. Nevertheless, the system is created over relational database system and direct integration with traditional database systems is also possible.

One of the most difficult problems in the area of data integration is handling of duplicate and inconsistent information. The key issue in this problem is entity identification, i.e. determining the correspondence between different records in different data sources [11, 14]. The reality requires that the system administrators understand the principles of the entity matching algorithm; thus, various difficult formalisms presented in the theory [7] are not applicable. Our approach uses a simplified entity matching system which allows the users to specify matching parameters that are easy to understand. Some researchers [6] advice that successful entity identification requires additional semantics information. Since this information cannot be generally given in advance, the integrated system should be able to defer decision to the user. The system should detect inconsistencies and either resolve them, or allow users to resolve them manually. The need for user-assisted conflict resolution induces a new class of problems: The repository should be able to store data before final resolution while their relationship to the real world entities is not consistent. Consequently, the system should be able to merge entities whenever the users discover that the entities describe the same real-world entity, and, conversely, to split an entity whenever the previous merge is found invalid. Under the presence of integrity constraints and history records, this requirement needs special attention.

The relationship between the global system and local database is usually expressed using the global-as-view and local-as-view approaches [5]. In our system, a mixture of these methods is used depending on the degree of integration required.

Maintenance of history records falls in the area of temporal databases and queries, where many successful solutions are known [1, 4, 12]. The theory usually distinguishes between the valid time, for which the data element is valid in the real world, and the transaction time, recording the moments when the data entry was inserted, updated, or deleted. In our approach, the central system automatically assigns and stores the transaction time, while the local systems are responsible for maintaining the valid time where appropriate. Queries based on transaction time are processed by special algorithms implemented in the central system; queries related to valid time are processed in the same manner as queries to normal attributes.

The rest of the paper is organized as follows: The second chapter describes the principles of the DataPile technology used to flexibly store structured data in a relational database system. The next chapter focuses on entity identification using data matching and relevance weighing. The fourth chapter shows the overall architecture of the integrated system. The fifth chapter presents an evaluation based on a commercial data-integration project where the DataPile approach was used.

2 The DataPile

2.1 Terminology

We have used an own terminology, which is partly derived from the XML terminology. The first term is *entity*, which represents a type of the traditional database row. An entity consists of *attributes*, which are analogous to the traditional database columns. An *entity instance* is an instance of entity and directly equals to traditional database row contents. An *attribute value* is an instance of attribute and forms a value of one column in one row. A *metatable* is a conventional database table used by the DataPile to store schema information and other system data.

2.2 Data Verticalization

Usual information systems consist of some nontrivial number of conventional database tables; huge information systems have huge number of such tables. Moreover, the requirement for preserving all changes in data usually leads to the scheme, where changing one value of one column in one row causes inserting a new changed row (possibly very large) and updating the old row with some state changing column (e.g. validity termination timestamp). Another problem in conventional information systems is extensibility; adding some new columns or new tables may cause large application code rewriting.

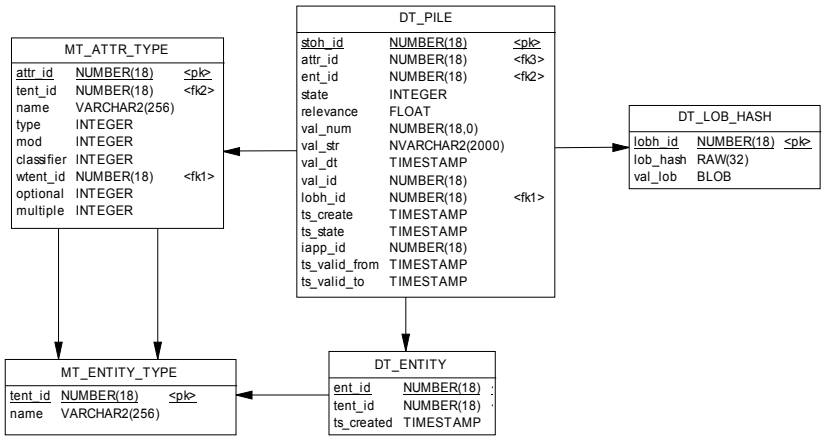

Figure 1. A sample schema of the DataPile-based system

All these problems are addressed by the proposed method of storing data in different way than in traditional approaches but using standard relational databases – the DataPile. All real applications data are stored in two relational tables: one less important table DT_LOB_HASH is dedicated for storing LOBs (for performance purposes), and the second one, the most important, DT_PILE stores data of all other datatypes. This particular table is called the Pile, because all data is stored in one table without any "well-formed" internal structure or hierarchy. Each row in the pile represents one attribute, whose value is/was valid during certain interval of transaction time.

The Fig. 1 represents slightly simplified schema of the heart of DataPile-based information system. Tables with prefix DT_ hold real data; all other tables (with prefix MT_) are metatables. The table DT_ENTITY holds valid "global" ID for an entity instance stored in the pile together with information about the entity in form of a reference to the metatable MT_ENTITY_TYPE which stores entities. Entities consist of attributes, and this is modeled by the metatable MT_ATTR_TYPE.

Real values are stored in columns val_xxx of the main table DT_PILE , where xxx represents logical type of the attribute (number, string, datetime, ID – foreign key). Besides the actual data, other additional data is stored in the DT_PILE table: Transaction time aspect of any attribute value is represented by two columns ts_valid_xxx. The type of given attribute value can be found by reference attr_id to the MT_ATTR_TYPE. The ent_id value compounds all attribute values into one entity instance. Other columns not mentioned here serve the system for proper implementation of the functionality needed.

Such a structure easily avoids all the problems mentioned at the beginning of this paper: The number of relational tables used does not grow with an expansion of an information system; it is constant regardless on how huge the system is. Data changes are preserved with minimal overhead – one attribute value change is represented by inserting a new value into the pile – one new row is inserted into DT_PILE table not touching the rest of attribute values related to the same entity instance. Extensibility of the system is reached by the possibility to insert some new rows into metatables and therefore the possibility of defining new entities, attributes, or both.

From the above described layout we can see that this data structure fulfils two requirements put on the information system as a whole: easy extensibility of the data scheme and full information about data changes on the timeline.

3 Data Matching and Weighing

The requirement on data unification is solved by two algorithms: data matching and data weighing.

3.1 Data Matching

Let us show an example, which represents usual situation we meet while processing the same data in different applications. Let application A1 have a record about a per-

son with the name “Jana”, surname “Teskova” with some personal identification number “806010/7000” and an address “Mother’s home No. 10”. The same information is stored in the application A2 as well. After Jana Teskova got married, she took her husband’s surname (as it is quite usual over here). So her surname changes to “Stanclova”. She also moved to live with her new husband on the address “New home 20”. Our person notifies about her marriage and the accompanying changes only the office using application A1, and does not notify other office with application A2 - at first she might not even know A2 does not share the data with A1 as they both are used to keep data about people in one organization, and at second she may expect that A1 and A2 are integrated together, so changes in A1 are automatically redistributed to A2 as well (but this is a so called distribution problem, which is discussed later). So the result is A1 and A2 store different data about one entity instance. What happens when we try to merge data from A1 and A2 into a common data storage?

As our example shows, nearly all attributes have changed. But some of them are constant, especially personal identification number, which should by truly unique in our country. The association of words “should be” unfortunately means that cases exist, when different persons have the same personal identification number. On the other side, these cases are rare. Having two personal records with the same personal identification number means they belong in fact to a single person with probability of roughly 0,999999.

In this example, other attributes have changed, but a combination of some attributes can have significant meaning: e.g. name and surname together form a whole name. Even name and surname aren’t commonly unique in a state, equality of such attributes means some nontrivial probability these two records describe a single person.

This example leads us to attribute classification. Every attribute is assigned one of these classes: determinant, relevant, uninteresting.

- Determinant – identifies an entity instance with very high probability (e.g. personal identification number, passport number etc.).
- Relevant – significant attribute, which helps identify unambiguously equality of entities (e.g. attribute types “name” and “surname” for entity type “person”).
- Uninteresting – has no impact on entity matching.

Following algorithm describes entity matching for two entity instances (one is already stored in the database, the second one is a newly integrated/created entity):

1. All determinant and relevant attribute values are equal – quite clear match with very high probability.
2. A nonempty subset of determinant and nonempty subset of relevant attribute values are equal, remaining determinant and relevant attribute values have no counterpart in the complimentary entity instance – very good match with quite high probability yet (example: let us extend our example with another attribute “passport number”. The first entity instance has attributes “personal identification number” and “passport number” filled. The second entity instance has only “personal identification number” filled and “passport number” is missing.).

3. A nonempty subset of determinant attribute values is equal, remaining determinant attribute values has no counterpart in the complimentary entity instance, but some nonempty subset of relevant attribute values differ – this case seems to be clear as well, because the probability of match for determinant attribute values outweighs probability of different relevant attribute values, but some uncertainty remains as the probability of determinant attribute values is always <1.0. We must not loose any data and their history, so the system solves such a case by considering these two entity instances as different with notification to the system administrator. The administrator can investigate this case more precisely and can merge these two entities together using an administrator application. This case directly describes the situation during entity matching from our first example – the personal identification number as a determinant attribute value is the same, but surname as a subset of relevant attribute values differs.
4. A nonempty subset of determinant attribute values differs, remaining determinant attribute values with counterpart are equal, some nonempty subset of relevant attribute values is equal, remaining relevant attribute values have no counterpart – this case usually arises out of misspelling one determinant attribute value. This case is solved as above – entity instances are considered to be different and the system administrator is notified.
5. All other cases – input entities are different entity instances with very high probability.

3.2 Data Weighing

Let us show another example: An employee record is usually kept in different applications in different departments, e.g. human resources department, payroll/accountants department, library, etc. Some applications and departments themselves emphasize some entities and usually some subset of attributes from entities used, e.g. staff department knows with high probability that given person has a certain name, surname, home address, etc., whereas payroll department knows with high probability his/her account number, etc.

It should be beneficial for data integration to have possibility measure somehow the probability, that an application has entity instances (or more precisely on individual attribute values) filled with correct values. Therefore, every attribute value in the DataPile keeps a number which measures probability this given value is correct. This probability is stored in the DT_PILE column “relevance”.

During processing of incoming data all incoming attribute values are somehow evaluated (this will be explained later) and the computed relevance is compared to current relevance of attribute value stored in the DataPile. If the new value has greater or equal computed relevance than current value has, the new value “wins”, becomes the current value, and the old value is marked as archive. Otherwise (when the relevance is lower than current relevance) the new value is stored as well, but only as a remark saying the application has ineffectually tried to change this attribute value.

But there is a problem: when an unimportant application with low relevance keeps correct data (replicated from the central repository) and wants to change some attribute values (because a user has made some changes), these changes will be always ignored. This problem is solved by "approving" the data. Every application must confirm to central repository it agrees with current data replicated from central repository to this application. This confirmation is stored in the DataPile and the system knows the given application has accepted the current attribute value. When such an application (which approved a current attribute value) changes the value, the rule about weighing relevance is ignored and the attribute value is changed.

For example, a large company has usually some branch offices, where department branches can be located as well. Such branches usually show different credibility, which should be reflected by the relevance computation as well.

For computing relevancy of an attribute value following equation is therefore used:

$$R_a = R_{ap} \cdot R_{iap} \cdot R_{et} \cdot R_{ec} \cdot R_{at} \tag{1}$$

where R_a means attribute value relevancy, R_{ap} is static application relevancy (e.g. application used by staff department), R_{iap} is static instance of application relevancy (e.g. branch of department), R_{et} is a static entity relevancy, R_{ec} is a computed entity instance relevancy, and R_{at} is a static attribute relevancy. All these static values are stored in metatables as floating-point values. R_{ec} represents computed relevancy for given entity instance and its value is computed as follows:

$$R_{ec} = \prod_i R_{ac}(i) \tag{2}$$

where $R_{ac}(i)$ is either a value stored in metatables (when attribute i from given entity has some particular value stored in metatables as well), or it has value of 1. This computed relevancy supports changing of the relevancy based on the presence of selected attributes in the entity.

4 Implementation

4.1 The DataPile Architecture

The whole DataPile-based system is in general shown on the following picture.

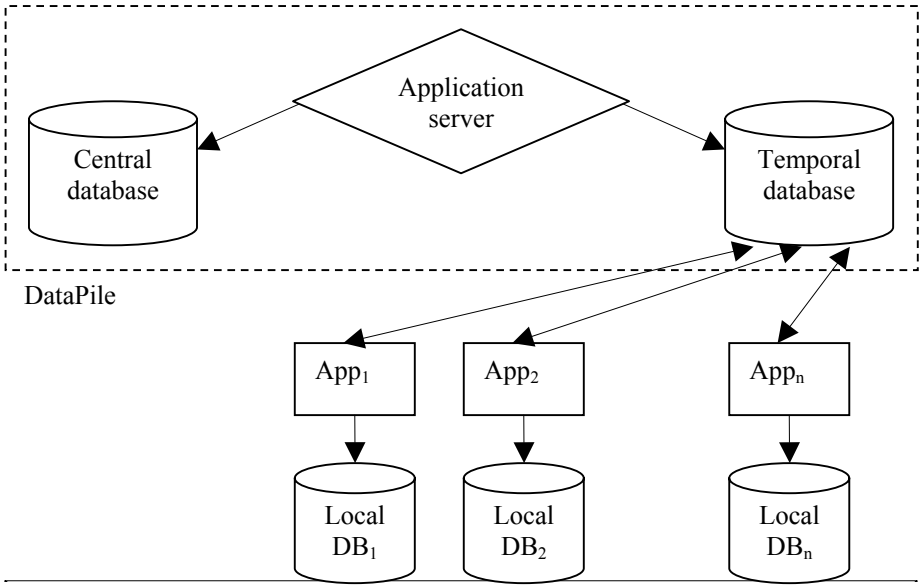

Figure 2. The DataPile architecture

Every application (marked in the picture as $App_{1..n}$) has its own local database (Local $DB_{1..n}$). The DataPile machinery is constituted by the Trinity of Central database, Temporal database, and Application server: the Central DB contains the DataPile as data structure for collected data storage, the Temporal DB serves only as communication medium between the DataPile machinery and applications (data is revealed here only during replication and immediately deleted when replication ends). The Application server, which gives life to the whole system, is discussed in following section.

4.2 Application Server

The whole DataPile architecture utilizes the request/reply paradigm. The application server behaves to the rest of world passively; it waits for requests inserted into the temporal DB, fulfills them using central DB data, and writes a reply back into the temporal DB.

Application server is by intention implemented so that it doesn't understand any data semantic. Everything is controlled by the content of metatables and nothing is hardcoded. The application server is primarily responsible for replication, computing algorithms like all the above mentioned data matching and weighing, and can handle other requests too (e.g. perform specialized queries upon the data stored in the DataPile).

4.3 Replication channel

The communication channel between an application and the DataPile machinery is not as simple as it may appear on the first look. In reality there are inserted two filters: export and import filter. They are traditional adapters, which are responsible for adapting different database schemas used by the DataPile machinery and local database. A watchful reader may note that the direction from application to the DataPile is marked as "export", opposite direction as "import"; the marking is taken from the application point of view.

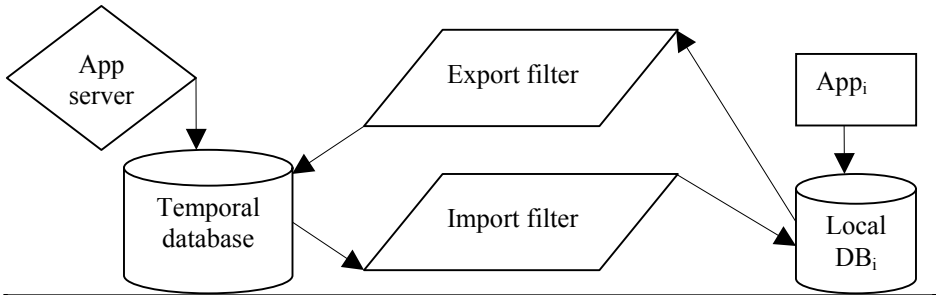

Figure 3. Communication between the DataPile machinery and an application

4.4 Cache

New applications may advantageously use collected data in central repository. Unfortunately the DataPile structure is not very well suited for direct access (e.g. searching is not very effective). Almost all applications in fact need to know only current attribute value, ignoring all history stored in the DataPile. To support such applications, the application server actively builds and maintains caches, where only current attribute values are stored and these caches are presented to applications in the form of traditional relational tables. The applications can easily search and use all other RDBMS functions on the caches.

5 Evaluation and Conclusions

The architecture described in this paper brings an alternative to traditional techniques. It brings several advantages but also some disadvantages.

All the concepts described in this paper were used in a real project – design and development of an information system based on data replication and synchronization of data coming from bigger number of different data sources (approx. 30 local information systems and 20 other applications producing 30 millions entries per year for 60 000 users, in this project). The project lasts from the fall 2003 and now is in the phase of finishing the pilot phase and starting roll-out.

Basic and commonly usable advantage of the DataPile structure is its maintainability, easy extensibility and ability to keep the track of the whole data history. All current applications used at all branches remain preserved and functional without any change according to a strongly desired requirement. The central data repository integrates data from all data sources including all their history and sources of their changes. This enables recovering of any historical snapshot of any data. All data changes are redistributed to all other applications that contain these data, even if the source and destination schemas are different. The global schema changes affect neither data in the central repository nor local applications.

During the development of the project, we have discovered several disadvantages of our approach:

Efficiency, especially during export and matching, is low. During the initial export of one certain local system, about 500 000 entries had to be processed. This took more than 24 hours. This time complexity is caused by a relatively complex matching algorithm. Fortunately, this time complexity is not very important in everyday life because number of data changes is smaller in magnitude in comparison to the initial migration data volume.

The second disadvantage is the fact, that the structure of the central repository makes constructing direct queries difficult. Therefore the concept of caches was introduced and all the queries to non-historical data are performed on the caches instead of the DataPile itself.

The project showed that the DataPile approach is suitable for certain class of large applications, where data warehousing is coupled with maintaining consistency of local databases. In this class of applications, the drawbacks mentioned above are outweighed by integration of data warehousing features with the support for data replication, synchronization, and cleaning using back-propagation.

References

1. Bruckner, R., List, B., Schiefer, J., Tjoa, J. A.: Modeling Temporal Consistency in Data Warehouses, In 12th International Workshop on Database and Expert Systems Applications (DEXA'01), IEEE Computer Society Press, Munich, Germany (2001) 901-905
2. Chawathe, S., Garcia-Molina, H., Hammer, J., Ireland, K., Papakonstantinou, Y., Ullman, J., Widom,J.: The TSIMMIS Project: Integration of Heterogeneous Information Sources. In Proc. of IPSJ Conference (1994) 7-18
3. Ibrahim, I. K., Schwinger, W.: Data Integration in Digital Libraries: Approaches and Challenges, Software Competence Center Hagenberg, Austria (2001)
4. Jensen, C. S., Snodgrass, R. T.: Temporal Data Management. IEEE TKDE (1999) 11(1): 36-45
5. Lenzerini, M.: Data integration: A theoretical perspective. In Proc. of the 21st ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems (PODS 2002) (2002) 233-246
6. Lim, E.P., Srivastava, J., Prabhakar, S., Richardson, J.: Entity identification in database integration, in Proceedings Ninth International Conference on Data Engineering, Vienna, Austria, April 19--23, 1993, IEEE Computer Society Press, Washington, DC (1993) 294-301

7. Mecella, M., Scannapieco, M., Virgillito, A., Baldoni, R., Catarci, T., Batini, C.: Managing Data Quality in Cooperative Information Systems, Proceedings of the 10th International Conference on Cooperative Information Systems, Irvine, CA (2002)
8. Mostéfaoui, A., Raynal, M., Roy, M., Agrawal, D., el Abbadi, A.: The Lord Of The Rings: Efficient Maintenance Of Views At Dataware Houses. Publication interne No. 1441, IRISA, Rennes, France (2002)
9. Nica, A., Lee, A. J., Rundensteiner, E. A.: The CVS Algorithm for View Synchronization in Evolvable Large-Scale Information Systems. In Proceedings of International Conference on Extending Database Technology (EDBT'98), Valencia, Spain (1998) 359-373
10. Rundensteiner, E. A., Koeller, A., Zhang, X.: Maintaining Data Warehouses Over Changing Information Sources, Communications of the ACM, Vol. 43, No.6 (2000)
11. Schallehn, E., Sattler, K., Saake, G.: Extensible and similarity-based grouping for data integration. In 8th Int. Conf. on Data Engineering (ICDE), San Jose, CA (2002)
12. Torp, K., Jensen, C. S., Snodgrass, R. T.: Stratum Approaches to Temporal DBMS Implementation. In Proceedings of IDEAS, Cardiff, Wales (1998) 4-13
13. Widom, J.: Research Problems in Data Warehousing. In Proceedings of the 4th Int'l Conference on Information and Knowledge Management (CIKM) (1995)
14. Yan, T. W., Garcia-Molina, H.: Duplicate removal in information dissemination. In Proceedings of VLDB-95, September 1995. Information Systems, Irvine, CA (2002)