in: Proc. Int. CAiSE'97 Workshop "Engineering Federated Database Systems (EFDBS'97)", Barcelona, pp.57-68.

Design and Architecture of the FDBS Prototype INFINITY

Theo Härder	Günter Sauter	Joachim Thomas
University of Kaiserslautern	Daimler-Benz AG, Research & Technology	IBM Toronto Laboratory
Dept. of Computer Science	Dept. CIM-Research (F3P)	1150 Eglinton Av East
67653 Kaiserslautern, Germany	89013 Ulm, Germany	North York, Ontario, Canada
haerder@informatik.uni-kl.de	guenter.sauter@dbag.ulm.DaimlerBenz.com	jthomas@torolab.vnet.ibm.com

Abstract

This paper focuses on the architecture of our prototype supporting the integration of heterogeneous data. The main characteristics of our system are its extended schema architecture and the generic translation approach based on a mapping language. At first, we introduce the schema architecture as well as the essential properties of our mapping language. One of our major contributions is the execution model which describes the system dynamics when data is derived from different sources and combined/converted to the specified application view. As the second essential contribution, the paper presents an optimization mechanism, called context management, to address the deficiencies resulting from an object-oriented interface on top of relational database systems.

Keywords: algebraic query processing, context management, federated database system, mapping language, middleware, schema integration, schema mapping, STEP

1. Motivation

The approach discussed in this paper resulted from research on a uniform product data model, which has been carried out at the CIM-Research Dept. of Daimler-Benz in Ulm/Germany in the past few years. For Daimler-Benz, just like for many other manufacturing companies, the increasing demand for flexibility and variety of product palettes calls for a uniform system environment permitting global consistency of product data as well as interoperability among all participating data-processing subsystems. For example, geometrical product data, supported by CAD systems, and its corresponding logical bill-of-material structure, administered by data management systems, must be maintained in a single environment in order to provide a quick overview of as well as fast access to all relevant data associated with certain product lines. Even the introduction of new and extended systems requires the interoperation with or, at least, the access to so-called legacy systems. In particular together with the high availability of information via the World Wide Web, the demand for integrating the data of multiple databases (DBs) is strongly increasing.

There are two general types of problems that impede interoperability defined as the capability of *heterogeneous* systems to cooperate. Firstly, the schemas of the DBs to be integrated might strongly differ (**structural heterogeneity** including incomplete coverage of data types and possibly different data models) and cannot be replaced by homogenous alternatives. One of the main characteristics of legacy systems is either the absence of a conceptual schema or its strong similarity to the internal schema. Most application programs explicitly require high-speed access to data which often implicitly calls for unnormalized schemas which are highly tuned for very specific access profiles. As a consequence, the structure of schemas differs with the applications and their access profiles. However, migrating to a new system generation that would allow to reimplement applications in a more uniform way and that would abstract as far as possible from details of the physical data representation is often highly uneconomical. Legacy systems are usually intertwined into the information-processing infrastructure being queried via hand-coded interfaces by numerous application programs and related systems. An atomic switch to powerful successors is therefore an expensive and delicate undertaking. Another argument additionally opting against this strategy is the relatively low frequency of accesses to those systems.

The second type of problem is **heterogeneity of semantics** which prevents the coupling of systems in a straightforward way. The DB design is biased by the needs of a particular application to optimize run-time performance. Analogously, integrity constraints are often embedded, distributed, and replicated within application programs thereby preventing a uniform, system-enforced control of the data semantics. As a result, at the level of the DB schema only a partial mapping of the application semantics is conceivable. Hence, capturing all these aspects of semantics cannot necessarily be conducted in an automatic way.

In Section 2, we present our schema architecture which is designed to address structural heterogeneity as well as heterogeneity of semantics. The mapping language which is briefly introduced in Section 3 is developed to bridge between the schemas of the various levels of this architecture. We give an overview of our overall system architecture in Section 4, and in Section 5 we detail some of its interesting aspects. Related work w.r.t. FDBSs and mapping languages is discussed in Section 6. Finally, the results are summed up in Section 7.

2. Schema Architecture of an FDBS

In general, aspects of **structural heterogeneity** and issues of schema integration are addressed by the architecture of federated database systems (FDBSs, [SL90]). The key idea is the translation of schemas written in heterogeneous data models into so-called component schemas written in a common data model. The latter schemas, respectively views on these schemas, are then integrated into the so-called federated schema. The schema transformation resp. integration is either specified explicitly in schema mapping languages or implicitly in query languages of the federation resp. access methods of an object-oriented federated schema.

There are several approaches coping with the **heterogeneity of semantics**. They are either based on automatic analysis of semantics or on human expertise. The latter proposals comprise reverse engineering methodologies and enrichment of the DB itself (see [HST97] for a discussion on this work). However, we made the experience that the basic assumption underlying any kind of automatism, i. e., having a schema with expressive names and a low degree of inter-relationships among the entities, is wishful thinking in most practical environments. Furthermore, the additionally enriched schema which is being extracted is most often not related in a formal way to the corresponding original DB. Proposals to enrich the databases themselves might be promising, but tend to make large DBs even larger. As a consequence, our approach relies on human guidance. We have chosen the international (ISO) standard for the exchange of product model data (STEP, [IS94a]) to address the problem of semantic conflicts. This standard defines a data model, called EXPRESS [IS94b], an access interface called SDAI [IS96a], and a set of standardized schemas representing various application domains. Among those schemas, an important one represents bill-of-material structures in the automotive industry. Currently, this schema contains about 300 entities, comprehensively described in the document [IS96b] so that a common and clear understanding of the schema can be anticipated.

The main idea of our schema architecture is to have not only a common data model before integrating the local heterogeneous schemas, but also a common schema structure which is based on the STEP standard. Analogously to [SL90], the first step in our process of schema integration is the translation of schemas written in heterogeneous data models into "data-model homogenized" schemas (component schemas) written in the EXPRESS data model. Many approaches addressing the problems of heterogeneous databases assume only minor conflicts among the schemas to be integrated, e. g., renaming of attributes and entities. Often, an implicit harmonization of the heterogeneous schemas is proposed. That is, the resolution of conflicts caused by different structures of the schemas (e. g., isomorphic entity correspondences) and the integration of those schemas into the federated schema is combined in one step. In contrast, we split this two-phase process and turn the implicit resolution of conflicts into an explicit additional schema level which is called “schema-structure homogenized”. That is, each local database provides an interface to the FDBS based on a schema written in the common data model of the federation and having a common structure. Obviously, the application domain may differ strongly, but the way in which identical domains are represented is then harmonized. If the application domain is already captured by the STEP standard, the corresponding schema is able to build the basis of the schema-structure homogenized layer. For example, [IS96b] can be employed in the automotive industry to define a common schema structure. This idea is shown in the following figure.

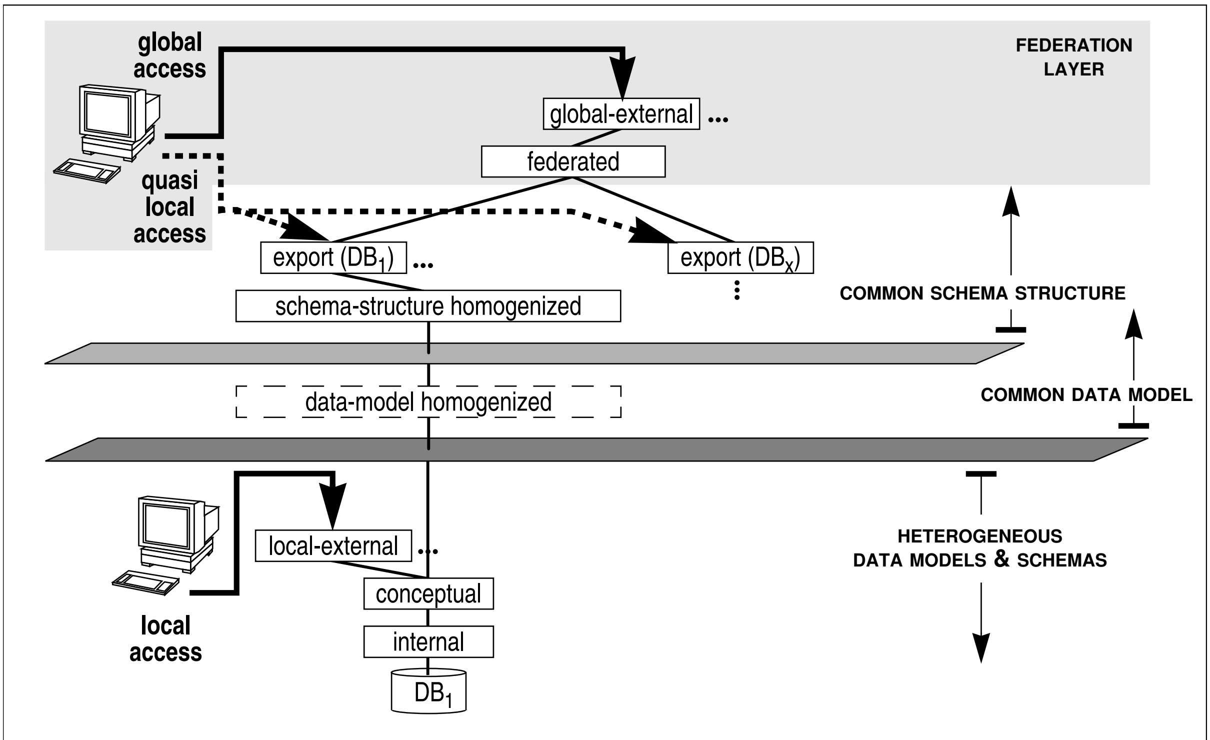

Figure 1: Schema architecture of our approach

One of the major benefits of our extended schema architecture is its portability of global application programs (vertical portability). Since export schemas, the federated schema, and global-external schemas all have the same structure and are all written in the same data model, the corresponding data can be accessed by the same queries. For the same reasons, local application programs which access the data according to the schema-structure homogenized interface can be migrated to other local databases without changing the queries (horizontal portability).

As stated before, a prime advantage of our approach is to have not only a common data model as a basis for schema integration, but also a common schema structure with given semantics defined in ISO documents. This is particularly helpful when integrating databases containing complex bill-of-material structures of different companies. Actually, the integration is simply the union of schemas, all written in the common data model and all having the same well-defined structure of an ISO standard.

The two-step translation of heterogeneous constructs (occurring at the data model level and the schema level) from local conceptual schemas into structure-homogenized schemas can be combined. That is, it is not distinguished between mapping heterogeneous data models to the common data model and mapping heterogeneous schema structures to the common schema structure. In this case, the data-model homogenized schema is only “virtual“. This procedure is sufficient if the conceptual schema, the structure-homogenized schema, and their inter-relationship are well-known. However, the two-step translation is advantageous when integrating legacy systems with an unclear data representation or a missing conceptual schema.

Each step of the translation process has to be specified in a mapping or view definition language. The language constructs have to define how the gap is to be bridged between (pre-existing) database schemas and the federated schema or a schema at a higher level of the schema architecture. We developed such a language which is called BRIITY (mapping language bridging heterogeneity).

3. The Mapping Language BRIITY

The key characteristics of our mapping language are its

- support of the integration of multiple schemas written in heterogeneous data models,
- power w. r. t. the number of mapping conflicts solved,
- descriptiveness, that is, declarative mapping specifications,
- immunity from technological changes, i. e., independence from platform characteristics, and
- support of user-defined update statements having the same expressiveness as retrieval statements.

In this section, we highlight the general structure of our language by referring to the mapping specification of Example 1.

```
 1: BEGIN
 2:   MAPPED_SCHEMAS
 3:       ts := target_schema <- rel_db:= rel_db@rel_dbs@localhost;
 4:   END_MAPPED_SCHEMAS;
 5:   INCLUDE
 6:       LIB /usr/users/sauter/libstring.a;
 7:       INC string.h;
 8:   END_INCLUDE;
 9:   TYPE_MAPPING
10:       MAP  ts.DM <- rel_db.US$;
11:            ts.DM <- 0.67 * rel_db.US$;
12:            rel_db.US$ <- 1.5 * ts.DM;
13:       END_MAP;
14:   END_TYPE_MAPPING;
15:   ENTITY_MAPPING
16:     MAP Department <- _pers:= rel_db.PERS;
17:       ON_RETRIEVE ...
18:       ON_UPDATE ...
19:       ON_INSERT ...
20:       ON_DELETE ...
21:     END_MAP;
22:   END_ENTITY_MAPPING;
23: END.
```

Example 1: General structure of a mapping specification[1]

A mapping specification starts with basic definitions that lay the foundations for the subsequent mapping rules, i. e., the names of the source(s) and the target schema involved (cf., line 2-4).

Data types and functions defined in some programming language that are related to the mapping process itself and that, for this reason, cannot be attributed to S or T (to avoid introducing dependencies in those schemas or to violate their autonomy), can be imported with the help of the `INCLUDE` section (cf., line 9-14).

1. Without loss of generality, we will disregard giving syntax definitions of our language because of space limitations. Interested readers may refer to [Sa96a].

For a better structuring of our mapping specifications, we separate the mappings of type-level constructs from those of the entity level. Types (built-in or user-defined ones) are handled in the `TYPE_MAPPING` section. Simple mappings without conversion functions consist only of the type mapping header as, e. g., line 10, which maps a *varchar* in S into a *string* in T. Apart from a type mapping header, more complicated mappings also possess a type mapping body (line 11-13) that permits to refer to arbitrary mathematical expressions and/or (included) functions. Thus, bi-directional conversions between data types can be defined, i. e., from S to T (line 11), or vice versa (line 12).

The essential part of such a specification is the `ENTITY_MAPPING` section which relates target entities and their attributes on one side to source constructs on the other side. Like the type mapping section, the one responsible for entity mapping consists of a header (line 16), relating one target entity to one or more source entities, and a body (line 17-21) with the detailed definition of the mapping itself. The body is further subdivided into an `ON_RETRIEVE`, an `ON_UPDATE`, an `ON_INSERT`, and an `ON_DELETE` clause. In the `ON_RETRIEVE` clause, the user can define how retrieve operations on target attributes of the corresponding entity should be translated to DB accesses. The propagation of update operations according to the modifications of target attributes can be specified in the `ON_UPDATE` clause. Operations to be executed in S after the creation resp. deletion of a target instance can be defined in the `ON_INSERT` resp. `ON_DELETE` clause.

The last section of a mapping specification (omitted in Ex. 1) provides means to declare additional integrity constraints which may be applied when contradictory, incorrect, missing, or obsolete data occur.

4. System Architecture of INFINITY

So far, we have briefly introduced the main ideas of our schema architecture and mapping language which embody the key concepts enabling the integration of heterogeneous DBSs. To demonstrate the feasibility of our approach, we have developed an FDBS prototype called INFINITY which realizes our schema architecture and language. Its overall system architecture is illustrated in Figure 2.

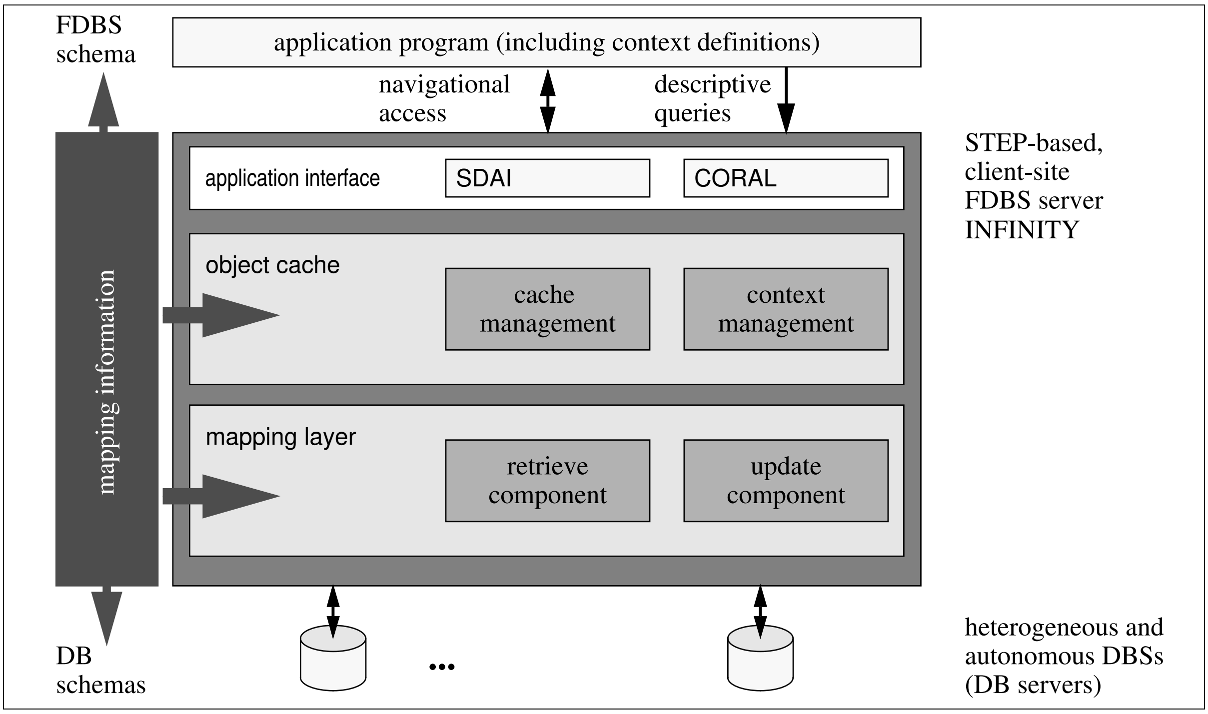

Figure 2: Overall system architecture of our prototype

The top layer provides to the users an SDAI interface which is implemented in C++. In addition, the query interface CORAL supports the application programmer to define pre-fetching statements which are called contexts (see below). Data retrieved from the sources (relational, object-oriented and STEP DBs[2]) is stored in an object cache which is illustrated as the intermediate layer of the INFINITY architecture. The translation of user queries into DB accesses is performed in the mapping layer beneath the cache using the mapping rules defined in BRIITY.

5. Execution Model

The goal of the execution model is to describe the system dynamics when data is derived from different sources and combined/converted to the specified application view. The main task of this mapping process is achieved by the mapping layer (cf., Figure 2). This layer is designed to accept descriptive as well as navigational queries at its interface towards a client cache (the object cache in this architecture). We have chosen an algebraic query translation approach because of its applicability and suitability for relational as well as object-oriented query processing and optimization. It allows for the use of well-known query processing techniques from relational database management systems [Mi95, Th96].

The BRIITY mapping rules, shown in Example 1, are stored in arbitrary ASCII files. They are processed by a parser thereby translating the query into a so-called mapping graph of a given algebra (cf., Figure 3 ❶). Currently, the complete mapping specification is parsed at compile time, translated, and stored in a pool of algebra graphs. The power of the mapping language resp. its individual mapping rules as well as the query language for the target schema determine the expressive power of the algebra needed. Thus, we have chosen an algebra similar to NF^2 [AFS89]. The leaves of the mapping graph are DB access operations to the integrated DBSs (e. g., `ACCESS PERS` in Figure 3) whereas the internal nodes are formed by algebra operators. Data from different DBs are integrated by join operators checking also for inconsistent and conflicting data. For example, instances of different sources having the same logical identity (ID) and different attribute values are refused resp. corrected according to user-defined operations. The root of the operator graph is a specific operator to create target instances (e. g., `MAP Department` in Figure 3).

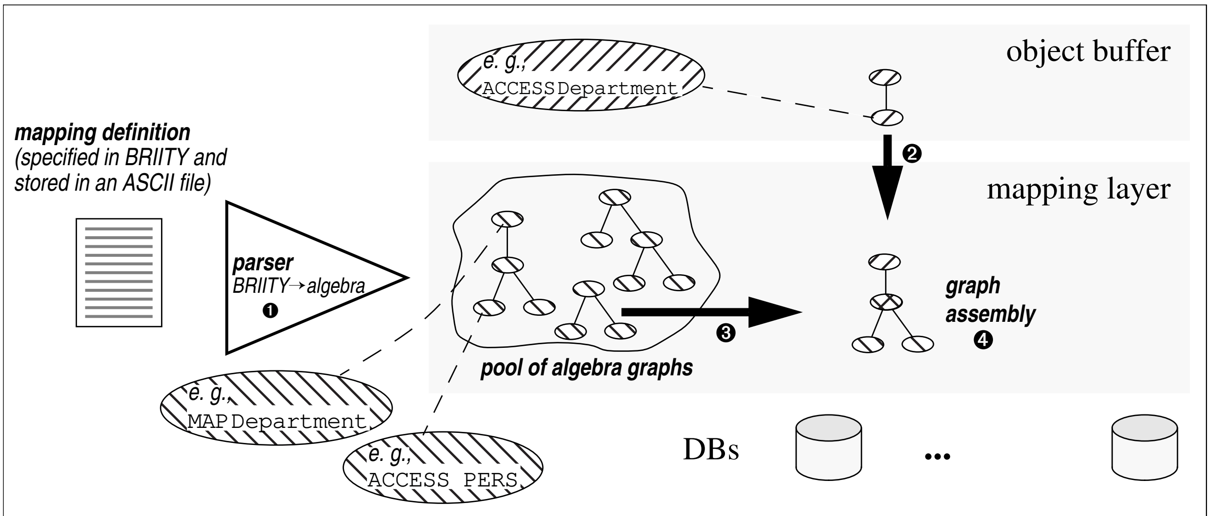

Figure 3: Query processing

Loading data into the object buffer is requested by sending a set-oriented query to the mapping layer. This layer transforms the query into an algebra graph, the so-called query graph (cf., Figure 3 ❷). To

2. We are using the term STEP DB to denote a DBS having an SDAI interface (and data described by an EXPRESS schema).

complete the translation process, the corresponding mapping graph is selected from the pool of algebra graphs (cf., Figure 3 ❸), i. e., the graph with the resp. root operator. Both the query and the mapping graphs are then assembled by removing the root operator of the mapping graph and the access operator of the query graph (cf., Figure 3 ❹). This is necessary in order to make the pool graphs independent of the application scenarios in which they are to be employed. The resulting graph builds the basis for further optimizations and for the generation of executable code to be sent to the DBs. The retrieved data is kept in (nested) relations within the mapping layer and processed according to the operators of the algebra graph.

IDs assigned to target instances and the corresponding IDs of source constructs are maintained in a separate table for IDs. Thus, a one-to-one relationship between target instances and corresponding source instances can be established. BRIITY supports the definition of target IDs resp. the derivation of target ID values from source values. Furthermore, the relationship between target and source IDs can be used in BRIITY to specify the propagation of updates according to the target schema to the integrated DBs. Thus, some view-update problems can be solved[3].

In the following discussion, we want to address the problem of navigational resp. object-oriented interfaces on top of relational DBSs. As described before, the object cache provides a navigational interface to applications. In general, DB accesses are initiated by object references causing object faults. Assume, the entity 'Employee' has an attribute 'Job' and the user wants to select all programmers. In this case, an iterator has to be created for the entity 'Employee' followed by loading the object identities (OIDs) into the object cache. In the worst case scenario, this operation has to be translated into DB accesses for the selection of all employees and the projection of the relevant information to generate the target OID. If supported, only the qualifying instances have to be retrieved to create the correct OIDs. Then, each instance is checked for the value of the attribute 'Job', i. e., in the worst case, for each instance the corresponding DB select statement has to be executed. Obviously, such an exhaustive selection can be very time-consuming. Therefore, we introduced the concept of contexts to optimize DB access. The user is allowed to specify context statements in the language CORAL (very similar to SQL3) within the application program labeled as comments and attached to aggregates resp. iterators. For example, the context defined in Figure 4 (cf., box at the top right side of the figure) is attached to the iterator `dept_iterator`.

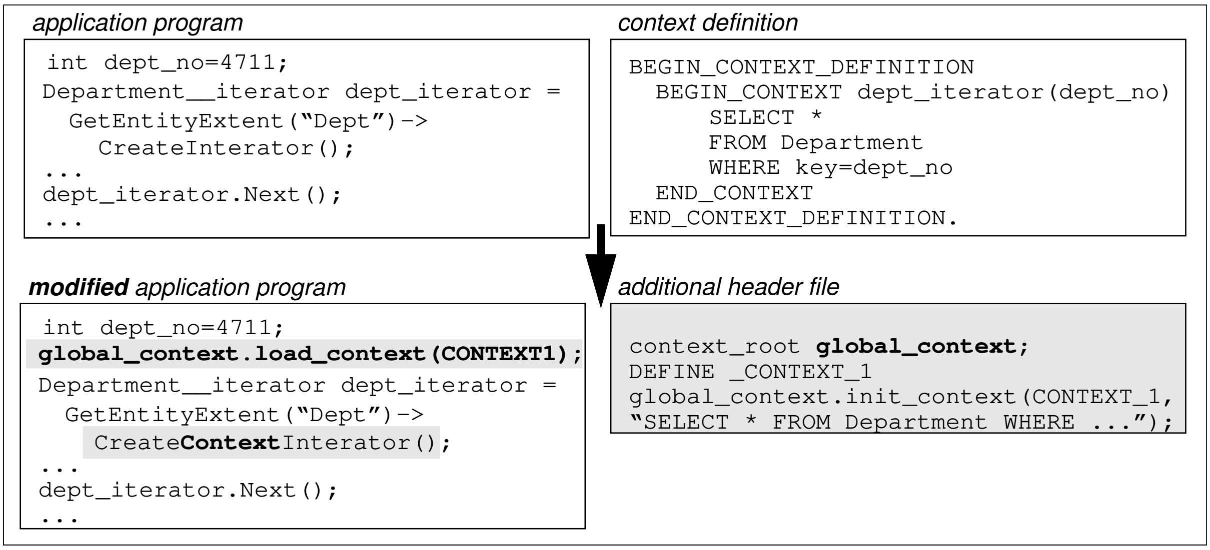

Figure 4: Context definition

3. Here, a comparison of view-update problems and solutions given by the language can not be performed.

A pre-processor evaluates the context statements and modifies the application program so that a query can be exploited to prefetch the relevant data asynchronously to the regular processing of the application program (cf., Figure 4 grey-shaded boxes). That is, before the entity extent is iterated, the relevant instances can be retrieved and fetched in a single statement. Variables used in the application program can also be referenced in the context definition (cf., Figure 4 `dept_no`). It should be emphasized that applications without contexts are supported by our object cache, too, and that applications including contexts can refer to an object cache not supporting the concept of contexts as well. Thus, we have optimized the system with respect to the

- frequency of invoking the mapping component and communication frequency between the FDBS and the integrated DBSs: one access instead of n+1 accesses (one for the select of all OIDs and n for the retrieval of each instance).
- number of the translation processes within the mapping component: the translation processes are very similar in both cases, because of the strong similarity of the requests of the object cache. That is, less and only slightly more complex (one additional select predicate) translation processes return the precise resp. relevant results.
- amount of data to be sent from DBSs to the FDBS (programmers instead of all employees with their OIDs, see example above).
- response time for the first qualifying instance, because the instances can be prefetched and false drops are not transferred to the application.
- response time for all other qualifying instances as well, because only "hits" are transferred to the application, and therefore.
- main memory utilization as well.
- frequency of data replacement, because less data has to be cached and information about the cache contents is available by predicates, i. e., set-oriented descriptions.

We are concluding the description of our execution model with an example (see [ST96] for more details). In Figure 5, a target schema written in either an object-oriented data model or an EXPRESS data model has to be mapped to a relational source schema.

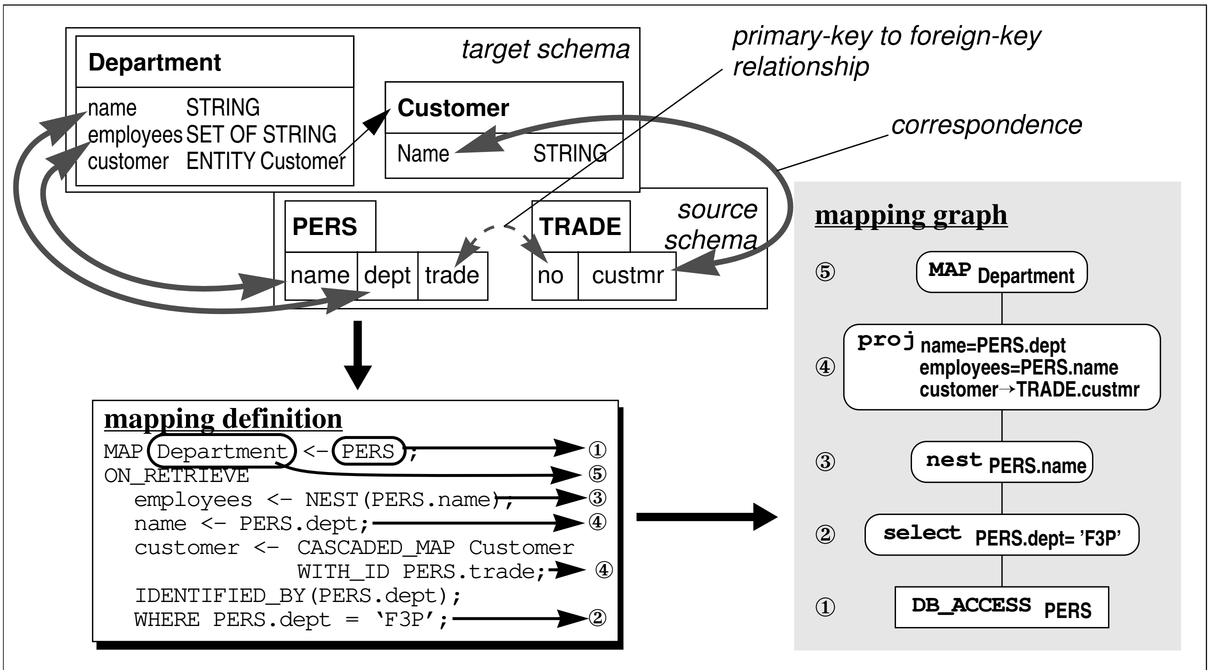

Figure 5: Mapping description and transformation

In this example, we focus on the mapping of the entity 'Department' to its corresponding relational model constructs, i. e., to the relation 'PERS'. The information of all persons working in the same department has to be nested in one target instance of this department. This kind of correspondence is specified in BRIITY by the NEST operation. The IDENTIFIED_BY clause relates target IDs to source IDs. If source instances have to be nested in a straightforward way, this clause is also used for grouping operations. Retrieving data of the complete complex entity 'Department' requires the mapping of the referenced entity 'Customer' in a *cascaded* way. This is defined by means of the CASCADED_MAP statement. The selection of the corresponding source information to be mapped to the referenced entity (an instance of 'Customer' in the example above) is defined by the WITH_ID clause. In the next step, this mapping definition is translated into a mapping graph. Due to space limitations, this process cannot be described in further detail.

As described before (cf., Figure 3), user requests resp. context queries according to the target schema are translated into algebra graphs and are assembled with the corresponding mapping graph. The resulting graph is used in the next step to send a query (queries) to the source DBS(s) and to derive the requested data. This is briefly sketched in the following figure where the result of the assembly of algebra and mapping graphs is shown on the left side. The corresponding steps of the mapping process are sketched on the right side of Figure 6.

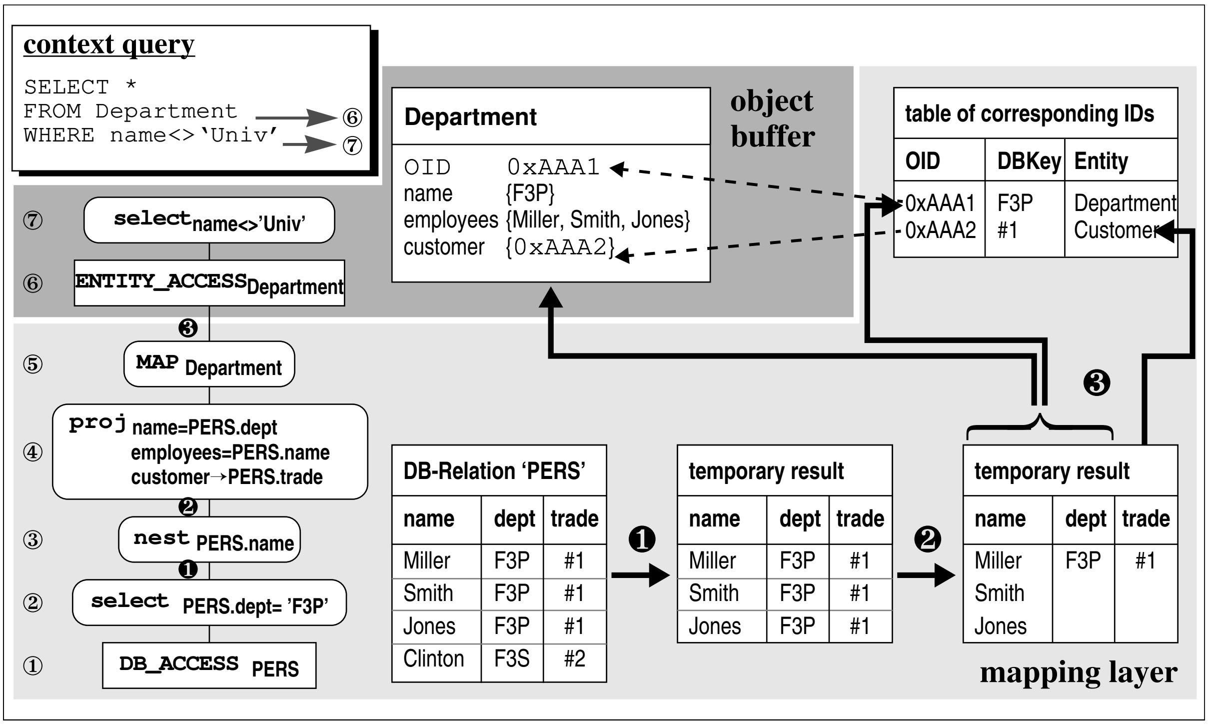

Figure 6: Execution process of a user request (cf., Figure 5)

The algebra operators ⑥ and ⑦ (cf., Figure 6) are generated from the context query and are assembled with the algebra operators of the mapping graph (①-⑤). The data is retrieved from the DBSs, evaluated according to the operators of the given algebra graph, and translated into a (nested) relation representing target instances to be built (cf., Figure 6 ❶ & ❷). OIDs are assigned when instances are created in the object cache and related to the source IDs according to the IDENTIFIED_BY statement of the mapping rule (cf., Figure 6 ❸). Hence, if the referenced instance (of the entity 'Customer') is already created, the correct target OID can be selected from the table of corresponding IDs using the WITH_ID clause of the IDENTIFIED_BY statement.

Summing up, BRIITY's execution model is based on algebraic query processing. This, in turn, allows for the application of well-known relational optimization techniques. Additionally, it provides concepts for the management of contexts. This distinguishing feature substantially minimizes communication as well as processing overhead.

6. Related Work

To facilitate the characterization of our approach, we briefly classify the related work. We distinguish between two different types of FDBSs according to having implicitly vs. explicitly specified the mapping between source and target schemas.

The so-called **wrapper-based approaches** are intended to address the growing demand of integrated access to a variety of data repositories ranging from relational over non-relational DBs up to non-database sources including spreadsheets, text-processing documents, electronic mail, images, etc. While providing a uniform interface (API) to the new applications, the underlying data is encapsulated by realizing data access via standardized access interfaces. Since the native query support, the sources provide, is so different in expressive power (e. g., from simple file scans to join operations on complex objects or media-specific search facilities), it would be impractical to perform repository accesses through a single standard interface. Hence, this approach necessarily has to exploit the query and access capabilities of the participating data systems. Encapsulation, however, has to be achieved by the user by writing wrappers for every type of data source to be included (existing wrappers could be made available in a special library). The use of wrappers tries to encapsulate the various heterogeneous data and therefore masks many problems of structural and semantic heterogeneity (or shifts them to the wrapper writers). In wrapper-based approaches, the user query specifies the selection of the view data thereby associating the relationship between the target and the participating source schemas and converting isomorphic structures (for example, renaming). Hence, this approach restricts the flexibility of source assignment and burdens the user with the complexities of selecting and converting the required data[4]. Moreover, update of data sources, although possible in principle, is often not considered.

In contrast, **schema mapping approaches** allow for a separate view definition „independent“ from the source schemas. As a consequence, they require an explicit mapping specification, i. e., rules describing the derivation of target data from source data (and vice versa). Obviously, such an approach can provide more flexibility as far as renaming, source assignment, multi-source correspondences, etc. is concerned. Moreover, by delegating the mapping task to DB experts the complexities of the source schemas can be kept transparent for the user. As motivated above, these languages have to bridge between potentially poorly defined schemas and, as a consequence, have to make explicit and to document hidden semantics. Furthermore, explicit specification allows for better understanding and control of the interrelationship between target and source schemas as well as the data correspondences among source schemas.

Many papers have been published on schema mapping [AR90, Ba95, CL94, Ha95, KC95, KDN90, Ke91, KFMRN96, KLK91, PGU96, SPD92, SST94, TC94], but a satisfactory approach was not proposed so far. Although **logic-based views** as proposed in [KLK91] are specified in a declarative way, it is controversial whether or not a mapping specification based on (first-order) logic is intuitively understandable. Furthermore, only minor conflicts are solved, such as different naming, heterogeneous attribute correspondences, integration of multiple relational schemas, and update capabilities. A fundamental drawback to be found in all logic-based approaches is the fact that they do not consider the propagation of update operations on views. **Procedural languages** [KFMRN96, Ke91] are very powerful

4. The source schemas we referenced have about 200 entities. An 'item' was represented by about 20 relations, each of them having 50 attributes on the average.

and may include explicit data type mappings. However, the larger the schemas to be mapped, the more unreadable and unclear the complete mapping specification will be. Furthermore, the procedural description restricts the execution of the mapping, in particular the creation of target instances which is mostly determined in a specific ordering. Obviously, this strategy might cause problems in applications having large schemas. **Declarative language approaches** are easier to understand, more flexible to use, and more suitable to be optimized at run-time. However, all these approaches do not support the explicit specification of update operations, i. e., the well-known view-update problems cannot be solved by those languages. Consequently, updating the essential parts of integrated views, which are defined by joining source elements, is restricted. Most of these approaches do not support nest/unnest operations, nor target object inter-relationships. To the best of our knowledge, integrity constraints to be evaluated during the mapping, e. g., to address the problem of conflicting/missing source data, are not within the scope of related languages.

Execution models of such declarative languages realize a more generic kind of middleware technology because of the distinction resp. separation between the description model, i. e. the view / mapping language, and its execution. However, current systems allow only limited variability of the data sources, e. g., they can access relational DBs of different vendors as well as non-relational data sources such as formatted files. The views made available for the application are based on traditional view languages and are more or less relational with SQL as an API. For example, the product DataJoiner [IBM95] which currently supports single-source update is indicative for these approaches.

7. Summary

In this paper, we have presented our approach to the integration of heterogeneous DBSs, i. e., the coupling of DBSs embodying different data models and schema structures to represent the same or overlapping information. We have sketched the underlying schema architecture and the expressiveness of our mapping language BRIITY. Compared to related work, our mapping approach is more powerful and flexible w.r.t. the solution of mapping conflicts such as renaming, multi-source correspondences, (un-) nesting of attributes, etc. Our main proposal is a kind of execution model for the mapping specification. It describes and controls the dynamic process of deriving target data from the participating sources thereby providing the combination/conversion of data as required by the application. We have also briefly sketched our optimization techniques introduced by the concept of contexts, i. e., user-defined descriptive statements to make prefetching possible.

8. References

AFS89 S. Abiteboul, P.C. Fischer, H.-J. Schek (Eds.): “Nested Relations and Complex Objects in Databases“, Lecture Notes in Computer Science, Springer, 1989

AR90 R. Ahmed, A. Rafii: “Relational Schema Mapping and Query Translation in Pegasus”, Technical Report HPL-DTD-90-12, Hewlett-Packard Laboratories, Oct. 1990

Ba95 I. Bailey: “EXPRESS-M Reference Manual”, ISO Document of TC184/SC4/WG5 N243, Aug. 1995

BE96 O.A. Bukhres, A.K. Elmagarmid: “Object-Oriented Multidatabase Systems”, Prentice Hall, 1996

Bl97 J.A. Blakeley: “Universal Data Access with OLE DB”, Proc. IEEE Compcon’97, San Jose, IEEE Computer Society Press, Feb. 1997, pp. 2-7

CL94 J. Chomicki, W. Litwin: “Declarative Definition of Object-Oriented Multidatabase Mappings”, in: Distributed Object Management, Morgan Kaufmann Publishers, 1994

Ha95 M. Hardwick: “EXPRESS-V Language”, Technical Report, Rensselaer Polytechnic Institute, Laboratory for Industrial Information Infrastructure, 1995

HST97 T. Härder, G. Sauter, J. Thomas: “The Intrinsic Problems of Heterogeneity and an Approach to Their Solution”, submitted, 1997

IBM95 “DataJoiner: A Multidatabase Server - Version 1”, White Paper, IBM, 1995, http://www.software.ibm.com/data/pubs/papers/

IS94a ISO 10303 - Industrial automation systems and integration - Product data representation and exchange - Part 1: “Overview and fundamental principles”, Int. Standard, 1994

IS94b ISO 10303 - Industrial automation systems and integration - Product data representation and exchange - Part 11: “Description methods: The EXPRESS language reference manual”, Int. Standard, 1st edition, 1994

IS96a ISO 10303 - Industrial automation systems and integration - Product data representation and exchange - Part 22: “Standard Data Access Interface”, ISO Document TC184/SC4/WG7, 1996

IS96b ISO 10303 - Industrial automation systems and integration - Product data representation and exchange - Part 214: “Application Protocol: Core Data for Automotive Mechanical Design Processes“, ISO Document TC184/SC4/WG3, 1996

KC95 J.-L. Koh, A.L.P. Chen: “A Mapping Strategy for Querying Multiple Object Databases with a Global Object Schema”, Proc. IEEE Int. Workshop on Research Issues on Data Engineering - Distributed Data Management, 1995

KCGS95 W. Kim, I. Choi, S. Gala, M. Scheevel: “On Resolving Schematic Heterogeneity in Multidatabase Systems”, in: W. Kim (Ed.): ’Modern Database Systems - The Object Model, Interoperability, and Beyond’, Addison Wesley, 1995, pp. 521-550

KDN90 M. Kaul, K. Drostern, E. Neuhold: “ViewSystem: Integrating Heterogeneous Information Bases by Object-Oriented Views”, Proc. 6th Int. Conf. on Data Engineering, Los Angeles, CA, Feb. 1990, pp. 2-10

Ke91 W. Kent: “Solving Domain Mismatch and Schema Mismatch Problems with an Object-Oriented Database Programming Language”, Proc. 17th Int. Conf. on Very Large Data Bases, Barcelona, 1991, pp. 147-158

KFMRN96 W. Klas, P. Fankhauser, P. Muth, T.C. Rakow, E.J. Neuhold: “Database Integration Using the Open Object-Oriented Database System VODAK”, in: O. Bukhres, A.K. Elmagarmid (Eds.): “Object-oriented Multidatabase Systems: A Solution for Advanced Applications”, Prentice Hall, 1996

KLK91 R. Krishnamurhy, W. Litwin, W. Kent: “Language Features for Interoperability of Databases with Schematic Discrepancies”, Proc. ACM Int. Conf. on Management of Data, 1991, pp. 40-49

Mi95 B. Mitschang: “Anfrageverarbeitung in Datenbanksystemen”, Vieweg, 1995

PGU96 Y. Papakonstantinou, H. Garcia-Molina, J. Ullman: “MedMaker: A Mediation System Based on Declarative Specifications”, Proc. Int. Conf. on Data Engineering, 1996

Sa96a G. Sauter: „The Mapping Language BRIITY - Reference Manual“, Technical Report F3-96-007, Daimler-Benz AG, Research & Technology, 1996

Sa96b G. Sauter: „Impacts of Heterogeneity on Interoperability“, Technical Report F3-96-021, Daimler-Benz AG, Research & Technology, 1996

SK95 G. Sauter, W. Käfer: „EXPRESS as the Common Data Model in Federated Database Systems“, Proc. 5th Int. EXPRESS User Group Conf., Grenoble, 1995

SL90 A. Sheth, J.A. Larson: “Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases”, ACM Comp. Surveys 22:3, 1990, pp.183-236

SPD92 S. Spaccapietra, C. Parent, Y. Dupont: “Model Independent Assertions for Integration of Heterogeneous Schemas”, VLDB Journal, Vol. 1, 1992, pp. 81-126

SST94 M.H. Scholl, H.J. Schek, M. Tresch: “Object Algebra and Views for Multi-Objectbases”, in: Distributed Object Management, Morgan Kaufmann Publishers, 1994

ST96 G. Sauter, J. Thomas: „Heterogene Sichten zur Unterstützung von Interoperabilität“, Technical Report F3-96-034, Daimler-Benz AG, Research & Technology, Dec. 1996

TC94 P.S.M. Tsai, A.L.P. Chen: “Concept Hierarchies for Database Integration in a Multidatabase System”, Proc. 6th ACM Int. Conf. on Management of Data, 1994

Th96 J. Thomas: “An Approach to Query Processing in Advanced Database Systems”, infix, DISDBIS 16, 1996