A Practical Approach to Access Heterogeneous and Distributed Databases

Fernando de Ferreira Rezende, Ulrich Hermsen, Georgiane de Sá Oliveira, Renata Costa Guedes Pereira, and Jochen Rütschlin

DaimlerChrysler AG – Research and Technology – Dept. FT3/EK – P.O.Box 2360
89081 Ulm – Germany
jochen.ruetchlin@daimlerchrysler.com

Abstract. A common problem within most large corporations nowadays is the diversity of database systems that are employed by their many departments in the development of a product. Usually, the total corporate data resource is characterized by multi-vendor database servers which, unfortunately, have no ability to relate data from heterogeneous data sources. In this paper, we present a database access interface which allows users to formulate SQL2 queries in a homogeneous way against a federation of heterogeneous databases. The database heterogeneity is not only completely hidden from the user, but what the user really perceives is a global database schema which can be queried as though all data reside in a single local database when, in fact, most of the data are distributed over heterogeneous, autonomous, and remote data sources. Further, the users can navigate through the database complex and compare, join, and relate information via a single graphic interface.

1 Introduction

Large corporations are often penalized by the problem of diversity of software systems. The many departments involved in the development of a product usually use different software tools; of course, each of them employs its own and familiar tools which help them to solve their problems, or their part of the complete product development work, in some way. Particularly in the field of the *engine development*, such tools vary mainly among simulation, CAD, testbenchs, and calculation tools. On the other hand, this diversity problem, or in other words the heterogeneity problem, is reflected in the many different sources of data used by the engineers to collect the necessary data for the product development as well. Usually, but by no means in all the cases, these data are managed by *Database Management Systems* (DBMSs) and stored in *databases* (DBs). Hence, the data are distributed along departmental and functional lines, leading to fragmented data resources. Such data distribution contributes to the emergence of the so-called *islands of information*. Thus, the data are lastly organized and managed by a mix of different DB systems (DBSs) from different software vendors. At both levels of this software diversity – the tools and DBSs levels – there exists the problem of communication between the software. The tools usually do not understand or are able to communicate with each other. And further, the autonomous DBSs have no ability to relate data from the heterogeneous data sources within the organization. The keyword to solve the heterogeneity problem at all levels is *integration* – tool as well as DB integration.

In the project MEntAs (*Engine Development Assistant*) – an innovation project at DaimlerChrysler Research and Technology – we have to cope with both sides of this

M. Jarke, A. Oberweis (Eds.): CAiSE'99, LNCS 1626, pp. 317-332, 1999.

heterogeneity problem. On one side, the several software tools employed by the engineers must be integrated; in the sense that the data produced by one tool can be automatically consumed by the next tools used in the engine development process. This data production and consumption among the tools generates a kind of data- and workflow, which is then managed and controlled by a *Workflow Management System*. On the other side, the several DBs must be integrated as well. In this paper, we will particularly give emphasis to this side of the software integration in the project MEntAs. We show how we have solved the DB heterogeneity problem in MEntAs in a very user-friendly way. We leave the discussion of our panacea to the problem of software tool integration in MEntAs to a later opportunity.

The problem that the engineers mainly have with the DB heterogeneity is to collect the information in the many data sources and, additionally, to correlate and to compare the data. The DBs are usually encapsulated by the applications, so that the engineer cannot have direct access to the data stored in them. Hence, they must use the many different interfaces provided by such applications to each DB. Furthermore, such interfaces are limited to a number of pre-defined queries that can be executed against a single DB. Since the process of designing and developing a new engine concept is extremely creative, the engineers often feel themselves limited in their creativity because such interfaces are not satisfactory in providing data from different data sources, in comparing and joining these data.

Our solution to this problem in MEntAs is based upon a *DB Middleware System* [1]. On top of a DB middleware system, we have designed and implemented a DB access interface which allows engineers to formulate queries in a homogeneous way against a federation of heterogeneous DBs. The queries follow the ISO standard SQL2 (*Structured Query Language* [2, 3]) for DB manipulation. By means of a *Graphic User Interface* (GUI), the engineers are friendly guided in the process of creating their own queries which may even traverse the boundaries of a DBS and involve the DB complex integrated in the federation. On processing the queries, the DB heterogeneity is not only completely hidden from the users. But what they really perceive is a global DB schema which can be queried as though all data reside in a single local DB when, in fact, most of the data are distributed over heterogeneous and remote data sources. Further, the engineers can navigate through the DB complex and compare, join, and relate information via a single, homogeneous graphic interface. By such a means, the right data are put available to the engineer much more comfortably and, most importantly, much faster than in usual developing environments.

This paper is organized as follows. In Sect. 2, we sketch the architecture of our DB access interface and explain its components separately. In Sect. 3, we deliberate on the functionality of the interface, showing how it looks like and how the engineer interacts with it. Finally, we conclude the paper in Sect. 4.

2 The Client/Server Architecture of the DB Access Interface

For the DB access interface in MEntAs, we have chosen a client/server architectural approach. This decision was mainly taken due to three reasons: scalability, parallelism, and multi-tier characteristic. Client/server architectures can scale up very well by simply adding to them new hardware power or software components whenever necessary. In addition, client/server computing has a natural parallelism: many clients submit many independent requests to the server that can be processed in

parallel, and furthermore, parallelism directly means better performance and so faster results. Finally, client/server architectures have the important feature of being multi-tier; they can be integrated to and take part in other architectures, new components can be added or dropped out, the server can play the role of a client, and the other way around, a client can be a server, etc.

Client/server architectures may be differentiated according to the way the data are transferred and the distribution of tasks is organized. The three most important forms are: *Page Server*, *Object Server*, and *Query Server* (the reader is referred to [4, 5, 6, 7] for more details on these approaches). We consider the client/server architecture of the MEntAs DB access interface as being a simple but very effective example of a query server approach (Fig. 1).

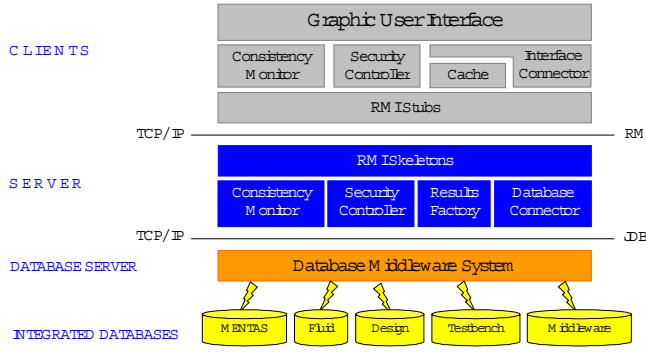

Fig. 1. Architecture of the MEntAs DB access interface.

Essentially, a query server works on the basis of contexts. A context normally comprises a set of complex objects and may be specified for example by means of MQL or SQL/XNF statements [8]. In the case of MEntAs, the contexts can be viewed as the SQL2 statements created by the own engineers via the GUI. The evaluation of such statements takes place in the server, where the objects are derived and stored in a transfer buffer. By means of operations such as projections, selections, and joins, the object views are tailored to the necessities of the users. In MEntAs, the essential functionality and power of the SQL2 language is put available to the engineer to create such operations. Hence, the volume of information to be transferred and fetched into the client cache is significantly reduced and optimized. Additionally, this approach is insensitive to cluster formations. Thus, ill-formed clusters do not affect the whole parallelism and concurrency of the system, because no objects are unnecessarily transferred to the client cache. Therefore, we hold the opinion that this approach mostly fulfils the necessities of MEntAs due to all its distinguishing qualities. This approach is a big research challenge, and up to now, it was followed only by prototype systems. Examples are: AIM-P [9], PRIMA [10], ADMS-EWS [11], XNF [8], Starburt's Coexistence Approach [12], Persistence [13], KRISYS [14], and PENGUIN [15].

Another important implementation decision we have taken during the design of the MEntAs DB access interface was the choice of an adequate programming language. Due to the broad range of hardware platforms and computer architectures employed by the mechanical engineers in the engine development (the heterogeneity at the level of hardware and software tools), we have committed for using *Java* as the

development environment and programming language, not only the client side but also the server side. This has shown later to be the right decision, since the development time was significantly reduced due to all facilities offered by Java, in comparison with for example C or C++, and its platform independence.

2.1 Integrated Databases

The *Integrated Databases* are the many data sources integrated in MEntAs. In the actual stage of the MEntAs development, these are all relational DBs [16].[1] Fig. 2 illustrates how the interconnection of heterogeneous DBs in MEntAs works. In order to construct a global DB schema for the engine development, we have analyzed together with the mechanical engineers the data models of each such DBs in order to identify the crucial data for MEntAs. In this process, we have recognized semantic differences, redundancies, synonyms, and homonyms. Thus, we could identify some common points, or better, overlapping objects, which could then be exploited for cross-DB join operations (DB navigation). After this analysis process, we have defined a set of SQL2 views for each DB reflecting the analyzed data model. Those views are created on top of the original DB by the corresponding department's DB administrator (mainly due to security reasons). On the other side, we tell the middleware system in our DB server that there are some views defined for it in a remote DB on some node in the network. This is done by means of nickname definitions in the middleware. By such a means, we bring the heterogeneous schemas into a global, virtual one, which contains just the data relevant for MEntAs.

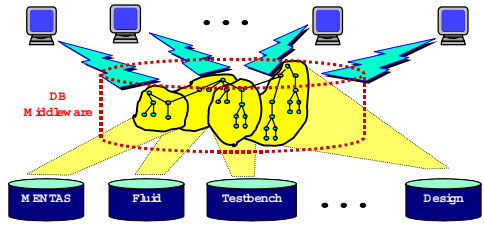

Fig. 2. The MEntAs approach towards integration of heterogeneous schemas in a global one.

2.2 Database Server

There are a lot of DB middleware systems commercially available in the market. The choice of the most appropriate one for MEntAs was based on a very detailed comparative analysis and performance evaluation. The complete results of our evaluation have been thoroughly reported in [17, 1, 18]. In particular, we have committed for using IBM's DataJoiner in MEntAs [19]. However, it is convenient to notice here that the client/server architecture of the MEntAs DB access interface is independent of the middleware system being used. This is so because the communication between the MEntAs server and the DB server, where the middleware is located on, is performed via JDBC (*Java Data Base Connectivity*), a Java standard for the access to relational DBs supporting the functionality and power of the SQL2

[1] We did not detect the use of object-oriented DBSs in any department involved in the engine development process. Nevertheless, the integration of DBSs other than relational, as for example hierarchical or network DBs, can be coped with by the DB middleware system either, but with some limitations.

standard. Therefore, MEntAs can apply any DB middleware system which offers a JDBC API without requiring great programming efforts for the modifications. Currently, practically all DB middleware systems provide a JDBC API.

2.3 Communication

2.3.1 Server/Database Server Communication

In the case of MEntAs, the use of the JDBC API for the communication between the server and the DB server is an adequate alternative since it is tailored to relational data sources [20, 21] (refer to Fig. 1). In addition, practically all well-known DB vendors offer a corresponding JDBC driver for their products. We can employ either a JDBC driver of *Type 2* or *Type 3* [20]. Fig. 3 presents both approaches. In simple words, as type 2 JDBC drivers are classified the ones which convert any JDBC operation to the equivalent operation supported by the client API of the corresponding DBS. Due to this particular feature, in order to apply a type 2 JDBC driver in the MEntAs architecture, we need to install an instance of the DB (middleware) client together with the MEntAs server (Fig. 3a). This type of driver has pros and cons. On the one hand, the platform independence of our MEntAs server is limited, becoming then restricted to the ones supported by the middleware vendor for its DB client. In addition to that, a type 2 JDBC driver is only partially implemented in Java, since many of its functionalities are provided by C libraries which are integrated in Java by means of the JNI (*Java Native Interfaces*) API. Lastly, we have unfortunately found out in the practice that the type 2 JDBC driver offered by our DB middleware vendor cannot execute queries in parallel. It serializes all concurrent queries being performed via all DB connections. In contrast, the type 2 JDBC drivers' implementations have shown to be more mature and stable than the type 3 drivers.

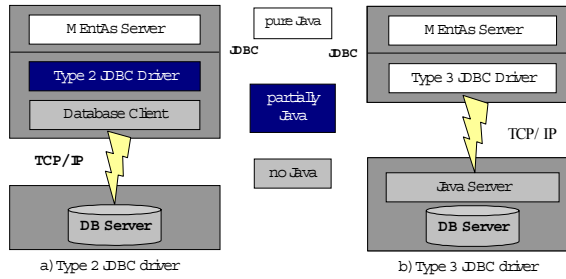

Fig. 3. JDBC driver variations in the implementation of the MEntAs server.

In turn, the type 3 JDBC drivers are implemented in pure Java. They are able to receive JDBC calls made via a DB independent network protocol (e.g., TCP/IP) and pass them on to a special component located at the server side, usually called *Java Server*, which then converts those calls to the corresponding operations understandable by the particular DBS (Fig. 3b). By using this type of driver, we were able to send concurrent queries to the DB server through different DB connections and it has processed all of them in parallel without problems. Unfortunately, the stability of the type 3 JDBC driver provided by our middleware vendor is not at all so high as its counterpart type 2 driver.

In MEntAs, the use of a type 3 JDBC driver is certainly the better approach, due to both its parallel query processing skills as well as its pure Java implementation. In the practice however, we were forced to implement and thus support both variations of the JDBC drivers in our architecture because of the low stability of the type 3 driver currently available. Which of both driver variations is employed in MEntAs can be set up by an input parameter when launching the MEntAs Server.

2.3.2 Client/Server Communication

The *Remote Method Invocation* (RMI) API is an efficient alternative for implementing the communication between distributed Java applications [22]. By means of RMI, distributed objects can be easily and elegantly implemented. Based on the ISO standard TCP/IP protocol, tools and APIs are put available to the software designer which enable a distribution at the object level. In comparison with for example a distribution at the network level via sockets, the distributed objects approach significantly shortens the development and implementation times, and in addition, it drastically reduces maintenance costs.

In comparison with other approaches at the object level, like for example CORBA [23, 24, 25], we believe that the performance of the RMI solution is better. In addition, since in the MEntAs architecture only homogeneous (Java) objects are distributed between clients and server, and hence just such objects must be considered for the integration, the power of the CORBA approach to integrate heterogeneous objects in a distributed environment is irrelevant in our particular case.

2.4 Database Connector

Management of DB Resources

In order to speed up the accesses to the DBs by the different components and at the same time to guarantee a global control over the DB resources, the DB connector manages *connection objects* by means of *object buffers* [26]. Object buffers allow the sharing of instantiated objects. The main advantage of using object buffers is that, since different calls to an object type do not cause a reinstantiation of such objects, no time is dispensed for the creation of those objects. In addition, since the objects are returned to the buffer after use, there is a reduction in the time spent for the garbage collection. This approach boosts performance and minimizes memory use.

Our implementation of the object buffer for connecting objects works as follows. By checking an object out of the object buffer, one receives an exclusive reference to a connection object. This connection can then be used to process an SQL2 statement, and thereafter by means of a checkin mechanism, it can be given back to the object buffer. Generally, all connections can be reused an unlimited number of times. However, in our case we have implemented a kind of timeout mechanism for the connections. After a connection has been used a pre-defined and adjustable number of times, we discard it and open a new again. In [26], a similar mechanism was suggested using timestamps.

Management of Queries

The JDBC drivers' implementation guidelines state that any Java object, which is no longer being used and which consumes DB resources, should free all used DB resources at the time it is caught by the Java's garbage collector. Additionally, it is supposed that the resources used during the execution of a statement should be implicitly freed or reused with the next statement's execution. By this way, the JDBC API handling should be simplified, since the software designer would be alleviated from the tasks of managing DB resources. Notwithstanding, unfortunately there exist

implementations of JDBC drivers which fulfil those guidelines only partially. Moreover, the close coupling of the garbage collector with the management of DB resources may be problematic and has shown us some disadvantages in the practice. As a matter of fact, the Java's garbage collector is automatically started whenever a Java process becomes memory bound. At this same time, it releases then the corresponding DB resources. On the other way around, the garbage collector is not started when the DB resources, e.g. connections, must be released due to a high overhead at the DB side. Hence, the triggering time for the garbage collector is merely dictated by memory bound situations of Java processes and not of DB processes. Due to that and to the fact that in MEntAs the overhead difference among the Java and DB processes may vary enormously, we follow the approach of explicitly freeing all DB resources by our own after each operation execution, independently of the JDBC driver being used. Hence, on the one hand, we are not subject to the faults of some driver implementations, and on the other hand, we have a better control about the use of the DB resources.

2.5 Results Factory

The successful processing of a query by the DB connector activates the results factory (Fig. 1). This component is used to create the result sets which are shown to the engineer at the client. On the basis of the DB cursor, the results factory fills its result object with a pre-defined, adjustable quantity of tuples by reading (*next*) the cursor repeatedly. By this way, the results of a query are transported to the client neither all at once nor tuple by tuple. On the contrary, the whole result table is divided into small parts, the *result sets*, which are then sent asynchronously to the client. The time that the engineer spends to visualize a result set is exploited by a prefetcher at the client side to load more result sets into the client cache in background. Thus, the engineer can smoothly navigate through the results, without having to wait an unnecessarily long time for the next result sets from the server.

2.6 Interface Connector

Management of the Global Schema's Meta-Information

The *Interface Connector* (Fig. 1) begins its work when the GUI is started at the client. At login time, it establishes a connection with the server. By means of the server's RMI registry, the client receives a reference to a *server object* which is used as a starting point for all other connections with the server. Via the server object, many other objects are loaded into the client containing meta-information about the integrated data sources in the DB server. The global schema's meta-data comprehend: *Table* - name, attributes, and commentary; *Attribute* - name, commentary, and data type. On the basis of these meta-data, a well-defined interface to the GUI is created, by which means the GUI can show to the end-user the attributes of a table, can present the commentary of an attribute/table as a help function, etc. Furthermore, with the information about the data types of the attributes, the GUI can correlate each attribute with its type-specific comparison operations.

Management of Error Codes and Messages

The error messages employed by the GUI to alert the end-user about any mal-function in the system are caught by the interface connector at initialization time either. The many different failure situations are characterized by an error code and its corresponding error message. This code and message correspondence is stored in the DB of the middleware and is loaded into a Java *properties object* at initialization time.

Management of Queries
As soon as the engineer has finished the interaction with the GUI to create a query, the GUI produces a string containing the corresponding SQL2 statement. Thereafter, the interface connector asynchronously sends the query to the server for processing. At this time, the thread implementing this object blocks waiting for an answer from the server. On receiving the awaited answer, this object produces an event of the AWT (*Abstract Window Toolkit*) whose identifier carries information about the success or failure of the query processing. In case of success, the GUI receives the first result set, and in case of failure it receives an appropriate error code to inform the engineer about his query's fate. After showing the first result set, the interface connector starts prefetching the next result sets from the server. Such a prefetching runs in an own thread and creates cache objects containing the next result sets.
Storing Select Statements and Result Sets/Tables
The interface connector offers methods to store the generated SQL2 statement in the DB of the middleware. This option to store pre-formulated SQL2 statements has received a good acceptance from the engineers, since they do not need to formulate frequently used queries over and over again. They only need to open the previously saved select statements, and on the basis of an identifier and a commentary given by their own, the complete select statement is sent to the server for processing by means of a single mouse click. Besides the select statement, the result table can be similarly stored in a text file for further visualization and processing by the user.

2.7 Cache

As shortly mentioned, the interface connector manages a cache at the client side for temporarily storing the results of the queries (Fig. 1). The current implementation of the MEntAs client generates for each executed query an instance of the cache class (let us call it *mini-cache*). Such a mini-cache stores then the results of a single query. This mini-cache also embodies a prefetcher, by means of which result sets are fetched from the DB server even beyond the actual cursor position seen by the engineer. Although this implementation of the cache contains sufficient functionality for the first beta version we have delivered of MEntAs, it has proven to be too simple and sometimes even ineffective in some cases. On the one hand, our actual cache storing algorithms have had storage capacity problems whenever handling huge amounts of data produced by some iceberg queries during the test phases. In addition, the independent management of the results of different queries is not the best approach, since each of them performs the same tasks in principle. A last point our cache is still lacking of is the possibility to store and manage mapping information. We intend to overcome these drawbacks in the next version of MEntAs by extending the actual functionality of the cache with an implementation of the *CAOS Cache* approach [27].

2.8 Consistency Monitor

There are mainly two potential sources of problems to be coped with during DB integration – schematic differences in the DBs as well as differences in the data representations. To cope with these problems is the main goal of the local and global consistency monitors (Fig. 1).
Management of Entry Points
In order to support the very important feature of DB navigation, we have identified in the integrated DBs' data models common data which serve the purpose of *entry points*. In simple words, for our purposes DB entry points are entities' attributes

which carry the same semantic information. Being so, they can be used for join operations which can cross DBs' boundaries.

Parser

The consistency monitor takes care of differences in the data representations of the integrated DBs. As an example of such differences, the identification of an engine in the MENTAS DB has been modeled according to the standard created by Mercedes-Benz, namely an engine must be identified by a *Type*, an *Specification*, and a *Sample of Construction*. Hence, we have simply created three attributes in the engine table to accommodate such information. Unfortunately, this standard and simple engine identification is, in surprisingly all cases, not at all followed by the data models of the integrated DBs. In a DB, one can find the engine type concatenated with the sample of construction by a dot in a single attribute, in another hyphens are used instead of dots to merge the engine type with the specification, in another there are no dots or hyphens at all and the engine type is merged with the specification in a single attribute whereas the sample of construction is represented in a completely different attribute, and so forth. All these peculiarities must be considered by the consistency monitor in order to make the navigation through the integrated DBs possible. On the one hand, without the intervention of the consistency monitor, the middleware is in most cases simply not able to compare the data due to the so different data representations, and thus the joins result in no matches. On the other hand, by using simple rules to resolve schematic heterogeneity of the type one-to-many attributes, as suggested by [28] where an operator for concatenating attributes is proposed to solve such problems, the middleware cannot join the data either, because many special characters are commonly used as separators in the attributes. And worse, in particular cases such separators seem to be simply ad-hoc chosen by the input data typewriter.

In order to bridge this heterogeneity, the consistency monitor manages complex grammatical rules which bring the DB schemas as well as the different data representations into a platform common to all DBs, on which basis the data can be securely compared, joined, and related to each other. These rules have been defined during the data modeling activities which resulted in the MEntAs global schema construction. However, notice that this common data representation is not followed in the integrated DBs' data models (with exception of the MENTAS DB itself). These are autonomous DBs and should not be changed to fulfil the MEntAs requirements. Thus, the consistency monitor must not only parse the input data to this common representation but also the corresponding data in the integrated DBs. Otherwise, the data as entered by the user and parsed could not be compared to the original data representations. Furthermore, since it would be very inefficient to parse all the huge amount of data in the tables where the entry points are contained in every time a navigation is taking place, we employ mapping tables. These are managed in the DB of the middleware and store the entry points previously parsed. All this functionality builds the basis to drive the navigation between the DBs.

Management of Mapping Tables

The mapping tables are automatically generated when the MEntAs server is firstly started. At this time, the consistency monitor scans all the entry points in the affected tables in the remote, integrated DBs, parses them accordingly, and finally generates the mapping tables containing the entry points in the pre-defined common representation. In addition to that, the mapping tables also contain an attribute for the entry points as represented in the original DBs. These are then used as a kind of

pointer to the tuples in the original tables. Hence, whenever a navigation is requested by the engineer, the (parsed) entry point as defined in the where clause is compared to the corresponding data in the mapping tables. Furthermore, by means of the attribute containing the entry point as originally represented, i.e. the pointers, the whole tuples can be found in the original tables. Hence, our mapping tables allow for a very efficient DB navigation because the entry points are all contained in those in the exact same representation as required for the navigation.

Another advanced feature of our consistency monitor is the shadow mechanism used in the generation of the mapping tables. For each entry point requiring schema mapping, the consistency monitor manages two mapping tables – a *current* and a *shadow* version. The current mapping tables give support to all MEntAs clients during the DB navigation. On the other hand, since the mapping tables' generation can be a time consuming operation depending on the original table's size, whenever asked for the consistency monitor starts asynchronous threads which generate the mapping tables in background using their shadow counterparts. When they have finished, it simply changes the current information it manages about the mapping tables, marking the current as shadows and the shadows as current. Of course, synchronization aspects and deadlock problems are appropriately coped with by the consistency monitor by means of refined routines employed in the table generation. By such a means, during the generation of the mapping tables, the engineers are blocked in their DB navigation operations only for a very short period of time, namely the time required to change the mapping table current information – a simple, single DB write operation.

2.9 Security Controller

Essentially, the main tasks of the local security controller are to manage the users' access rights as well as to encrypt and decrypt any data for the network transport (Fig. 1). On the other hand, the security controller plays a kernel role at the server side. It is responsible for managing the user data, access rights to the DBs, and for encrypting the data to the transport to the clients. The security controller is built upon the security mechanisms provided by both the operating system, namely UNIX at the server side [29], and Java [30]. Furthermore, it exploits the DB access controls and the corresponding fine granularity access rights' granting mechanisms supported by the DBMS. These mechanisms provide the basis for the security management in MEntAs.

3 The Functionality of the Graphic User Interface

3.1 The Data Representation

Nowadays, even with innumerous object models being proposed in the literature [31, 32, 33, 34, 35], the *Entity Relationship Model* (ER model [36]) is still spread out and employed in the industry for modeling DBs, and herewith relational DBs [16]. In MEntAs, we have chosen the ER model as the main mechanism for the data representation. Essentially, the data model of each integrated DB is shown to the engineer in the main interface of the GUI (refer to Fig. 4). By means of clicking with the mouse in the presented ER model, the engineer can start formulating his own SQL2 queries. It contains the following components: *Menu list* comprehends the main system commands; *Schema field* shows the ER diagram of the current DB; *Query field* shows the current select statement that is being built by the user (editable); and *Help display*: shows a helping text according to the current mouse position.

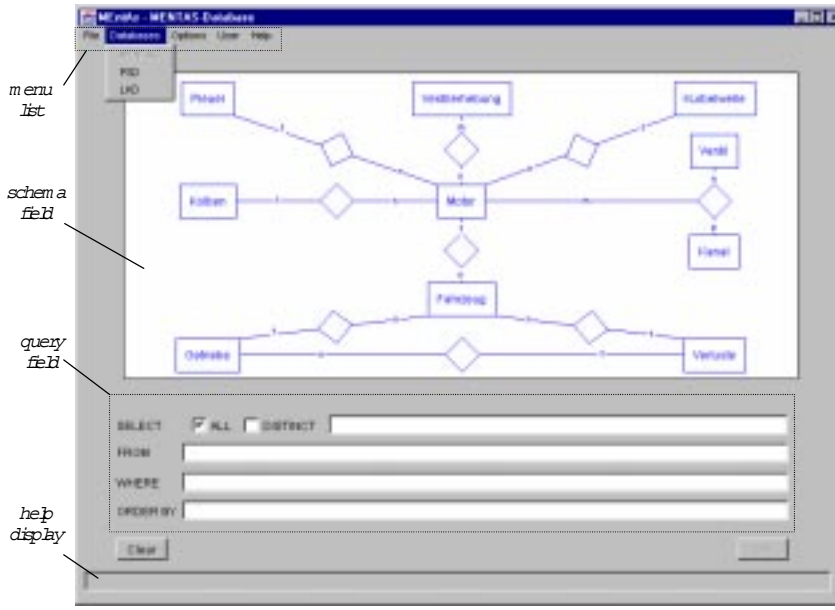

Fig. 4. The main window of MEntAs's DB access interface.

3.2 Formulating the Queries

In order to formulate a query, the user must focus the desired entity with the mouse and right click it. On doing that, a *popup menu* is activated with the following options (Fig. 5): *Select* specifies the projection for the select statement; *Where* defines the condition of the select; *Order by* specifies a sort ordering for showing the results; and *Other Databases* enables the navigation through the other integrated DBs.

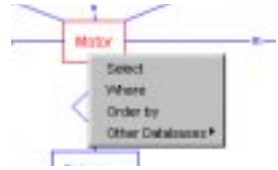

Fig. 5. Popup menu in the entities for formulating the queries.

3.2.1 Select/From

On choosing the select item in the popup menu of the schema field (refer to Fig. 5), the user receives a window containing all the attributes of the entity (Fig. 6). In particular, this same component is reused every time the user is asked to select some items of a list (let us generically refer to it as the *select window*). By means of mouse clicks, the user can select the attributes for the projection.

3.2.2 Where

On choosing the where item in the popup menu (Fig. 5), the select window is firstly shown to the user (Fig. 6). After that, the user can exactly define the condition for his select statement (Fig. 7).

Fig. 6. Showing the attributes for the projection.

Fig. 7. Showing the attributes for the condition definition.

3.2.3 Order by

The order by item of the popup menu (Fig. 5) allows the user to define a sort ordering for attributes. Similarly as before, the user firstly selects the attributes for the order-by definition (Fig. 6). Thereafter, he can choose the order of sorting (Fig. 8).

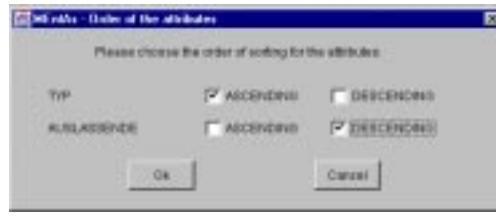

Fig. 8. Showing the attributes for the sort ordering.

3.3 The Results Interface

The results are presented to the user by means of the *results interface* (Fig. 9). Its components are: *Menu list* is similar to the main interface; *Query field* shows the user-formulated select statement as executed by the DB server; *Table/Attribute field*

presents the table and attribute names of the query's projection; and *Results field* presents the current result set in form of a table.

3.4 Joining Relations

The DB access interface of MEntAs allows users to make joins in a very friendly way, without even having to know the semantic meaning of primary and foreign keys. In principle, any relations that are related via a relationship can be joined. The only things the user needs to do for this are, firstly, to formulate a query on one entity as described in the previous sections, and secondly, to focus another related entity and perform similar steps. By such a means, the join condition, namely primary (foreign) key equals foreign (primary) key, is automatically generated by the GUI and appended to the complete select statement.

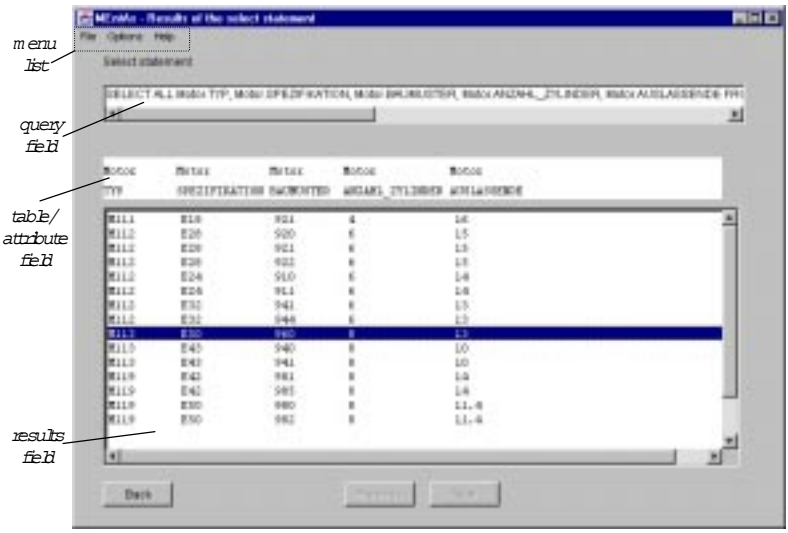

Fig. 9. The results interface.

In a very simple example, consider the ER diagram sketched in Fig. 4, and furthermore, suppose the engineer clicks the entity *engine* (*Motor*) and defines a projection on the engine's type and a condition to find all engines with *Type* equals *M111*. Now, if he clicks the entity *automobile* (*Fahrzeug*), chooses for the projection the automobile's class, and confirms with the OK button, he will lastly receive all automobile classes where the engine *M111* is built in. Internally, the GUI transforms these interactions to the following SQL2 statement:

```
SELECT  engine.type, automobile.class
FROM    engine, automobile
WHERE   engine.type = 'M111' AND
engine.pk_engine = automobile.fk_engine_automobile;
```

where *pk_engine* is the primary key of the entity engine, and *fk_engine_automobile* is a foreign key of automobile referring to the engine's primary key. This procedure of joining two relations can be recursively cascaded to neighbor relations. Extending the above example, if the engineer further clicks on the *gear* (*Getriebe*) and defines a

projection containing its *Type*, he will receive not only the automobile classes using the engine *M111* but also the respective gear types of those automobiles.

3.5 The Database Navigation

The DB navigation is based upon the concept of entry points. The DB entry points create a kind of spatial relationships between the entities of the data models of the integrated DBs. By selecting the *Other Databases* item in the popup menu of the main interface (Fig. 5), the user indicates that he wants to make cross-DB join operations via the entry point defined in the clicked entity. This is a very powerful feature by means of which the engineer can catch from another DB any information of a particular entity that is not contained in the current DB.

As a simple example, suppose the engineer currently queries the MENTAS DB, and further, that he wants to be informed about the fluid dynamics of a particular engine he is working on. Such information is not contained in the MENTAS DB itself but in the Fluid DB. In this case, he can exploit the DB entry point defined in the engine entity and navigate to the Fluid DB in order to select the desired fluid dynamic information. Whenever choosing another DB for navigation, the engineer receives the ER diagram of the corresponding DB sketched in the schema field of the main interface. At this time, the necessary join conditions are automatically generated by the GUI and appended to the select statement defined so far. Hence, the engineer can simply go ahead specifying his select statement in the very same way as before.

4 Conclusions

In large corporations, the data are usually distributed along departmental and functional lines, contributing to the emergence of islands of information. Unfortunately, these data are managed and organized by multi-vendor DBSs, which have no ability to relate data from heterogeneous and distributed data sources. In this paper, we have presented a practical approach to overcome DB heterogeneity via a homogeneous, user-friendly DB access interface. By means of it, engineers can create their own SQL2 select statements which may even traverse DB boundaries. On processing the queries against the DB federation, the DB heterogeneity is completely hidden from the engineers which only perceive a global DB schema. This allows the engineer to use his creativity and to find the right information much faster than in usual developing environments.

Our interface is based upon a client/server architectural approach. The *DB middleware system* provides a uniform and transparent view over the *integrated databases*. The *DB connector* manages the communication connections with the middleware system by using JDBC. The *results factory* produces sets of objects containing parts of the whole result tables of the SQL2 statements, giving support to the cache's prefetcher. The *interface connector* intercepts the SQL2 statements produced by the GUI and sends them to the server for processing via RMI. The *cache* serves the purpose of temporarily storing result sets, allowing for a smooth navigation through those. The *consistency monitor* allows for cross-DB operations. It brings the data to a common platform by means of the parser's grammatical rules, and employs mapping table mechanisms which support a secure DB navigation. The *security controller* manages users, authorizations, access rights, and encrypts/decrypts the data for the transport. Finally, our platform independent *GUI* interacts with the engineers

in a very friendly way, and guides them in the process of generating their own SQL2 select statements against a federation of heterogeneous DBs transparently.

References

1. Rezende, F.F., Hergula, K.: The Heterogeneity Problem and Middleware Technology: Experiences with and Performance of Database Gateways. In*: Proc. of the 24th VLDB*, USA, 1998. pp. 146-157.
2. Melton, J. (Ed.): *Database Language SQL 2*. ANSI, Washington, D.C., USA, 1990.
3. Date, C.J., Darwen, H.: *A Guide to the SQL Standard*. Addison-Wesley, 4th Ed., USA, 1997.
4. Härder, T., Mitschang, B., Nink, U., Ritter, N.: Workstation/Server Architectures for Database-Based Engineering Applications (in German). *Informatik Forschung & Entwicklung*, 1995.
5. Rezende, F.F., Härder, T.: An Approach to Multi-User KBMS in Workstation/Server Environments. *In: Proc. of the 11th Brazilian Symposium on Data Base Systems*, São Carlos, Brazil, 1996. pp. 58-72.
6. DeWitt, D.J., Maier, D., Futtersack, P., Velez, F.: A Study of Three Alternative Workstation/Server Architectures for Object-Oriented Databases. In: *Proc. of the 16th VLDB*, Australia, 1990. pp. 107-121.
7. Rezende, F.F.: Transaction Services for Knowledge Base Management Systems - Modeling Aspects, Architectural Issues, and Realization Techniques. infix Verlag, Germany, 1997.
8. Mitschang, B., Pirahesh, H., Pistor, P., Lindsay, B., Südkamp, S.: SQL/XNF-Processing Composite Objects as Abstractions over Relational Data. In*: Proc. Int. Conf. on Data Engineering*, Austria, 1993.
9. Küspert, K., Dadam, P., Günauer, J.: Cooperative Object Buffer Management in the Advanced Information Management Prototype. In: *Proc. of the 13th* VLDB, Brighton, U.K., 1987. pp. 483-492.
10. Härder, T., Hübel, C., Meyer-Wegener, K., Mitschang, B.: Processing and Transaction Concepts for Cooperation of Engineering Workstations and a Database Server. *DKE*, Vol. 3, 1988. pp. 87-107.
11. Roussopoulos, N., Delis, A.: Modern Client-Server DBMS Architectures. *ACM SIGMOD Record*, Vol. 20, No. 3, Sept. 1991. pp. 52-61.
12. Ananthanarayanan, R., Gottemukkala, V., Käfer, W., Lehman, T.J., Pirahesh, H.: Using the Coexistence Approach to Achieve Combined Functionality of Object-Oriented and Relational Systems. In: *Proc. of the ACM SIGMOD Int. Conf. on the Management of Data*, USA, May 1993. pp. 109-118.
13. Keller, A., Jensen, R., Agrawal, S.: Persistence Software: Bridging Object-Oriented Programming and Relational Database. In: *Proc. of the ACM SIGMOD Int. Conf. on the Management of Data*, Washington, D.C., USA, May 1993. pp. 523-528.
14. Thomas, J., Mitschang, B., Mattos, N.M., Dessloch, S.: Enhancing Knowledge Processing in Client/Server Environments. In: *Proc. of the 2nd ACM Int. Conf. on Information and Knowledge Management* (CIKM'93), Washington, D.C., USA, Nov. 1993. pp. 324-334.
15. Lee, B., Wiederhold, G.: Outer Joins and Filters for Instantiating Objects from Relational Databases Through Views. *IEEE Transactions on Knowledge and Data Engineering*, Vol. 6, No. 1, 1994.
16. Cood, E.F.: A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, Vol. 13, 1970. pp. 377-387.
17. Rezende, F.F., Hergula, K., and Schneider, P.: *A Comparative Analysis and Performance of Database Gateways*. Technical Report Nr. FT3/E-1998-001, DaimlerChrysler, Ulm, Germany, March 1998.
18. Hergula, K., and Rezende, F.F.: *A Detailed Analysis of Database Middleware Technologies* (in German). Technical Report Nr. FT3/E-1998-002, DaimlerChrysler, Ulm, Germany, June 1998.
19. IBM Corporation: DB2 DataJoiner Administration Guide. IBM, 1997.

20. Sun Microsystems Inc. http://java.sun.com/products/jdbc/overview.html, 1998.
21. Hamilton, G., Cattell, R., Fisher, M.: *JDBC Database Access with Java: A Tutorial and Annotated Reference*, Addison-Wesley, USA, 1997.
22. Sun Microsystems Inc. http://java.sun.com/products/jdk/rmi/index.html, 1998.
23. Object Management Group. The Common Object Request Broker Architecture and Specification (CORBA), OMG, Framingham, USA, 1992.
24. Object Management Group. *The Common Object Request Broker Architecture and Specification – Rev. 2.0*, Technical Report, OMG, Framingham, USA, 1995.
25. Orfali, R., Harkey, D., Edwards, J.: *The Essential Distributed Objects Survival Guide*. John Wiley & Sons, USA, 1994.
26. Davis, T.E.: Build your own Object Pool in Java to Boost Application Speed. *JavaWorld*, http://www.javaworld.com/javaworld/jw-06-1998/jw-06-object-pool.html, Jun. 1998.
27. Hermsen, U.: Design and Implementation of an Adaptable Cache in a Heterogeneous Client/Server Environment (in German). M.S. Thesis, Univ. of Kaiserslautern, Germany, 1998.
28. Kim, W., Choi, I., Gala, S., Scheevel, M.: On Resolving Schematic Heterogeneity in Multidatabase Systems. In: Kim, W. (Ed.), *Modern Database Systems – The Object Model, Interoperability, and Beyond*, Addison-Wesley, USA, 1995. (Chapter 26).
29. Garfinkel, S., Spafford, G.: *Practical Unix & Internet Security*. O'Reilly & Associates Inc., USA, 1996.
30. Fritzinger, J.S., Mueller, M.: *Java Security*. White Paper, Sun Microsystems Inc., 1996.
31. Atkinson, M., Bancilhon, F., DeWitt, D., Dittrich, K., Maier, D., Zdonik, S.: The Object-Oriented Database System Manifesto. In: Bancilhon, F., Delobel, C., Kanellakis, P. (Eds.), *Building an Object-Oriented Database System: The Story of O2*, Morgan Kaufmann, USA, 1992. pp. 3-20. (Chapter 1).
32. Cattell, R.G.G. (Ed.): *The Object Database Standard: ODMG-93*. Morgan Kaufmann, USA, 1994.
33. Kim, W.: *Introduction to Object-Oriented Databases*. The MIT Press, Massachusetts, USA, 1990.
34. Loomis, M.E.S., Atwood, T., Cattell, R., Duhl, J., Ferran, G., Wade, D.: The ODMG Object Model. *Joop*, Jun. 1993. pp. 64-69.
35. Stonebraker, M., Rowe, R.A., Lindsay, B.G., Gray, J.N., Carey, M., Brodie, M., Bernstein, P., Beech, D.: Third-Generation Database System Manifesto. *ACM SIGMOD Record*, Vol. 19, No. 4, Dec. 1990.
36. Chen, P.P.: The Entity-Relationship Model: Toward a Unified View of Data. *ACM Transactions on Database Systems*, Vol. 1, March 1976. pp. 9-37.