in: Proc. 3rd Int. Symposium on Cooperative Database Systems for Advanced Applications (CODAS'01), Bejing, April 200

To a Man with an ORDBMS everything looks like a Row in a Table

Wolfgang Mahnke, Hans-Peter Steiert
Database & Information Systems Group, University of Kaiserslautern
P.O.Box 3049, D-67653 Kaiserslautern
e-mail: {mahnke, steiert}@informatik.uni-kl.de

Abstract

In large software projects it is required to manage all project-related artefacts in a shared database in order to support cooperation of developers and reuse of design. Unfortunately, such projects have to be supported by various development tools using proprietary strategies for storing their persistent data. Since we need a strong query language to analyse the project-related data we choose an object-relational database system (ORDBMS) as integration platform. In this paper we will discuss possibilities of how to integrate external data in an ORDBMS. Further, we introduce a reference architecture for discussing the architectural options of an ORDBMS-based integration environment. Finally, we present our own system.

1. Introduction

In 1996 a new generation of database management systems appeared on the market. The advocates of the so-called object relational database management systems (ORDBMS) claim that these systems will support non-standard applications better than traditional relational DBMS. Our research goal in the SENSOR[1] project is to examine whether or not this is true. One of the research activities in SENSOR is to develop of guidelines for building ORDBMS-based applications and to provide tools supporting this task. We call this the SERUM[2] approach.

In the SERUM approach, we suggest a model-centred development process based on a tool-supported stepwise refinement of UML models (UML, Unified Modeling Language [14]) until they can be used as input for (semi-automatic) code generators. For this purpose, we have developed an ORDBMS-based UML repository. Managing UML models in a central repository has many advantages. First, a shared repository eases cooperation of developers involved in the SERUM process. Second, the repository serves as a basis for the reuse of design decisions documented as UML models. Third, higher software quality can be achieved by analysing UML models. This way, design errors can be detected early and design guidelines can be enforced. Query facilities provided by ORDBMSs have proven to be helpful for this task [15]. Last but not least, the repository serves as a tailor-made storage for the SERUM tools.

Why did we built our own UML repository instead of using existing products [2, 3]? First, we want to examine whether or not object-relational technology is sufficient to be applied in the field of engineering applications, as suggested by Stonebraker in [13]. Second, the object-relational technology helps integrating development tools. This is very important, since large software projects have to be supported by multiple development tools, most of them using a proprietary strategy for storing their persistent data. In order to use third-party tools and SERUM tools in a seamless manner, we want to provide access to heterogeneous data sources through our UML repository. This means to access external data with exactly the same SQL statements as data stored in the ORDBMS.

What is an ORDBMS? There is no common definition of the functionality of an ORDBMS. Even the SQL:1999 standard [1] can not serve for it because it does not contain some concepts that are already implemented in current ORDBMS whereas current ORDBMS have not implemented all concepts of the standard. Additionally, the evolution of object-relational technology is still moving fast leading to a continuous adjustment of the list of requirements. The properties postulated in the literature [5,12,13] so far can be assigned to three categories: *object-relational datamodel* (ORDM) with user-defined types (UDT), typed tables and inheritance on types and typed tables as well as user-defined routines (UDR); *extensibility infrastructure* like integrating new query language operators, data containers, table access methods or generic search trees [10]; and *functional enhancements* like a general-purpose rule system.

Using an ORDBMS as an integration platform, as proposed by Mattos et. al. in [12], means to access external data with SQL queries. Therefore, we needed an integration schema and had to built so-called 'wrappers' encapsulating the access to external data sources. Hence, our work is related to federated databases [16] and other projects dealing with heterogeneous data sources, as for example GARLIC [17,18] or OLE DB [4].

1. Subproject A3 *"**S**upporting Software **En**gineering Processes by **O**bject-**R**elational Database Technology"* of the Sonderforschungsbereich 501, *"Development of Large Systems by Generic Methods"*, funded by the German Science Foundation.
2. SERUM: Generating **S**oftware **E**ngineering **R**epositories using **UM**L

Because extensibility is the key feature for ORDBMS-based tool integration, we will examine its applicability for our purposes in section 2. We have developed a reference architecture for ORDBMS-based integration environments which we will introduce in section 3. In section 4, we will present how we integrate third-party tools in the SERUM process using our UML repository. Section 5 concludes the paper.

2. How to Integrate External Data Access?

The key functionality for managing external data with an ORDBMS is extensibility, i. e., an extensible data-model and an extensibility infrastructure. Integrating external data into an ORDBMS means that the DBMS extends the effects of SQL queries to externally stored data, too. Therefore, we need an integration schema which reflects the semantics of the external data. The extensible ORDM is much more suitable for this task than the traditional relational data-model.

However, a suitable extensibility infrastructure is even more important, because it enables us to overwrite the functions which an ORDBMS uses to handle values, rows and tables. In this section, we want to present three concepts for integrating external data together with suitable extension interfaces and, if available, known implementations. Since SQL queries refer to values, rows, and tables, we have to integrate external data as external values, external rows, or external tables. In Table 1 these concepts are listed together with examples for an appropriate extensibility infrastructure and known implementations.

Table 1. Concepts to integrate External Data

Concept	Extensibility Infrastructure	Known Implementations
external value	Informix opaque type	SQL:1999/MED (DATALINK)
external row	not available	not available
external table	Informix VTI	e.g. GAURON (see section 4)
	SQL:1999/MED (foreign data wrapper)	not available

2.1. External Values

The idea of handling external values (EV) is to provide a new data type which can be used like a built-in data type, i. e., as the type of an attribute of a row type or a table. EVs represent data items external to the ORDBMS. Operations invoked on EVs as part of the query processing are evaluated on the external data item.

Assume there exists an image database managed by a special-purpose server. Also, assume we are using an ORDBMS to manage the operational data. If we have to answer questions like ‘what is the average of items sold last year by bearded men’ then we have to consider data of both locations in a query [6]. If we make the ORDBMS treat images as EVs, then questions like the one above may be posed in SQL. For an application developer, the images look like values in the rows of a table.

Extensibility Infrastructure. An external value type (EVT) can be implemented as an opaque type. Opaque types are not part of the SQL:1999 Standard [1] but are supported by at least one ORDBMS [7]. In the following we will describe how to utilize this opaque type concept w.r.t. managing external data.

It is necessary to handle three different representations of an opaque type, i.e., the representation used in the API, the persistent representation stored on disk and the opaque main memory representation for operators and UDRs (see Fig. 1).

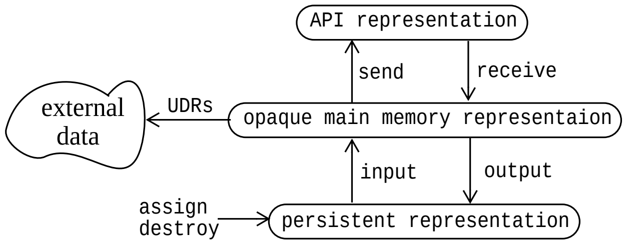

Figure 1. Interface of External Value Types

Using opaque types in the API. If a row with an opaque type is inserted using an INSERT statement then the application developer has to provide the value in a format which can be handled by the API. By calling the ‘receive’ method the ORDBMS creates an opaque representation of the value in main memory. The other way around, the ORDBMS calls the ‘send’ method, if an opaque type occurs in the result of a SELECT statement. This method converts the opaque type to its API representation.

Storing opaque types. Even if the external data item, e. g., an image, is not stored in the DBMS, there must be a persistent representation like the URL (Uniform Resource Locator) of the image. The record manager (RM), i. e., the DBMS component responsible for storing objects in database pages, uses three functions in order to fulfil this task. If it has to store an EV, it first allocates the required resources (‘assign’). Thereafter, the RM converts the opaque type to its persistent format (‘output’) and stores it to disk. Reading an opaque type from disk is performed by the ‘import’ method. If the RM has to delete an opaque type, it calls the ‘destroy’ method which deallocates the resources.

Using operators and UDRs. If the application developer needs operations on opaque types, for example an ‘hasBeard’ operation on images, implementations for these operations have to be provided. Even SQL operators and built-in functions may be adapted to EV characteristics by overwriting the DBMS-internal implementations.

Known Implementations. The upcoming SQL:1999/MED Standard [9] offers a new SQL data type called DATALINK which references externally stored data. This can be seen as one possible implementation of an EVT. A DATALINK value encodes the URL of an external data object managed by an external data source, typically an enhanced file system.

2.2. External Rows

EVs have two disadvantages. First, we can not define an built-in index on properties of an EV. Built-in index structures can be customized for EVTs, but they do only work on the values themselves, i. e., we can use a built-in index on images but not on the 'hasBeard' property. For this purpose, we have to create a user-implemented index structure. Second, you can not update the external items with SQL. The UPDATE statement works on table cells, not on the values. Updating a table cell means to delete a value and replace it by a new value. These problems could be solved by external row types (ERT), which make an image look like a row in a table.

Each ERT is related to a user-defined row type which is determined as part of the ERT definition. At the SQL level, the ERT has the same structure as its related type. Therefore, it can be used in any SQL statement in exactly the same way as an UDT with this structure. Moreover, you can define tables typed with an ERT, which can be queried and indexed as any other typed table.

Extensibility Infrastructure. So far, no extensibility infrastructure concerning ERTs is defined, neither in the standard nor in current ORDBMSs. Hence, we present our proposal for such an extensibility infrastructure in the following paragraphs.

An external row type is a user-implemented data type. ERTs differ from user-defined row types by the fact that main memory representation and persistent representation are opaque. The ORDBMS uses the methods given in Fig. 2 to handle external rows (ER). It uses two generic functions, 'getAttribute' and 'setAttribute' in order to access the structural properties. Based on an attribute identifier, the first method retrieves attribute values from the ER whereas the second sets attribute values.

```
class ExternalRowType
{   // external format
    public ExternalRowType();

    // attribute access
    public Value* getAttribute( AttrId* aid );
    public void setAttribute( AttrId* aid, Value* v );

    // persistent format
    public ExternalRowType( PersRow* v );
    public PersRow*             toPersRow();
    public int                  sizeOfPersRow();

    // operators
    ...}
```

Figure 2. Interface of External Row Types

Using external rows in the API. The API representation of the ERT is the API representation of the related type. Hence, the ORDBMS can generate it using the definition of an ERT and the attribute access methods without an additional cast method. If an ER is inserted, then the ORDBMS calls the constructor and creates an opaque representation. It sets the values of the attributes using the attribute access methods.

For INSERT and UPDATE statements the extension developer must ensure that the properties of the resulting ER are consistent with the properties of the external data item, which it does represent. In the case of an UPDATE statement, this means that the developer either has to reject the operation or to change the external data item. Of course, the 'hasBeard' property is difficult to change. However, if the image is stored in GIF format and the update changes the 'format' property to 'JPEG', the developer can use an appropriate filter to convert the image. Due to space restrictions we can not discuss the consistency problems in more detail here.

Storing external row types. Why does the RM not store the API representation to disk? From our point of view this would restrict the extension developer too much. Since main memory representation and persistent representation are opaque, they can contain information different from the external representation. The RM calls the 'sizeOfPersRow' method and determines the amount of resources needed. It allocates the resources and stores the result of the 'toPersRow' method to disk.

Using operators and UDRs on external row types. Operations needed for ERTs have to be implemented by the developer. As for EVTs the developer can overwrite SQL operators and built-in functions. The operations are called with the opaque representation as parameter.

Known Implementations. Because there is no extensibility infrastructure concerning ERTs available so far, there exist no implementations based on this concept.

2.3. External Tables

External values and external rows represent data items external to the ORDBMS and allow access to these items through user-implemented methods. In contrast, external tables do not represent an external data item but define an access method which can be used to iterate over a set of external data items. Using external tables you can scan almost everything like rows in a table.

Extensibility Interface. The VTI (Virtual Table Interface) offered by the Informix Dynamic Server [8] is a good example for an extensibility infrastructure which allows to implement external tables. The extensibility infrastructure defined by the VTI offers interfaces for the following tasks: creating and dropping external tables; scanning, inserting, updating and deleting the rows of an external table.

Since the VTI is very complex, we do not describe it in detail. We just want to consider the scan operation as an example (see

Fig. 3). The 'open' method is called before the scan begins processing. For example, this method can establish the connection to the external data source. The 'beginscan' method initializes anything needed to perform the scan on the external table. It receives a parameter, the scan descriptor, which describes qualifications and projections. If the external data source has appropriate functionality, it can perform these tasks. Now, the ORDBMS fetches the rows one by one by calling the 'getnext' method. When the table scan is finished, the 'endscan' method is called. If no further scans occur during processing the actual SQL statement, then the ORDBMS calls the 'close' method. Here, resources allocated during the scan operation are deallocated. The connection to the external data source is cut.

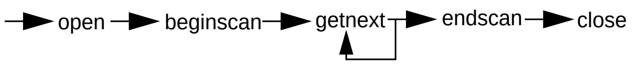

Figure 3. Scan Operation

Another extensibility infrastructure for external tables is introduced in SQL:1999/MED [9]. A foreign server is an external data source, managing external data objects. The ORDBMS accesses such external data objects through a mechanism called a 'foreign data wrapper'. Foreign data wrappers provide implementations for a set of interface routines similar to those introduced above. The ORDBMS calls these implementations in order to access the external data objects during query processing.

Nevertheless, there are differences to the concept of the VTI. The interfaces defined for foreign data wrappers are more powerful. For example, the ORDBMS can deligate the responsibility for a partial query to the foreign data wrapper which is quite more than a table scan. Hence, it seems to be intended that the foreign server is an SQL-aware data source, i. e., it can process SQL queries.

Known Implementations. An example for implementing external tables based on the VTI is our GAURON system which is presented in section 4. As far as we know, there are currently no implementations based on the concept of foreign data wrappers.

3. A Reference Architecture for Integrating External Data

In section 2, we have introduced external values, external rows, and external tables allowing the use of the SQL query power for computing external data. In current ORDBMSs, the extensibility infrastructure already offers interfaces for implementing external values and external tables. Which concept should be used to encapsulate access to external data depends on the requirements. But even if this choice is taken, further decisions have to be done as well:

- Which interfaces do applications use to access external data? Will all applications use the ORDBMS API or are there alternatives?
- Will the data reside only in the external data sources or do we have to manage copies in the ORDMBS?
- How much functionality should be implemented as an ORDBMS extension? Do we want to exploit the functionality of external data sources or do we directly use low-level interfaces?

In this section, we want to introduce a reference architecture in order to identify the options for building an ORDBMS-based integration environment.

Before presenting the options, we want to make some assumptions. In our scenario (Fig. 4), we assume that an external data object is a binary data stream. The lowest interface available for accessing it provides simple read and write operations, e. g. file system operations. An 'external server' is an independent software system, external to the ORDBMS. It manages external data objects and provides some value-added services as, for example, high-level operations on data items stored in the data objects. The term 'integration logic' encompasses all user-implemented code which is registered in the ORDBMS for the purpose of handling access to external data objects. In the case of external values, the integration logic consists of constructor and destructor functions, cast functions, SQL operator functions and UDRs.

In this scenario an application program can access external data objects using three different interfaces:

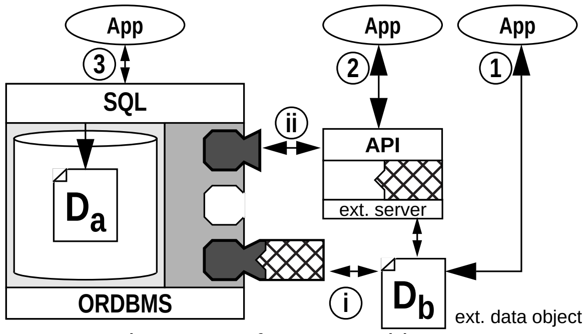

Figure 4. Reference Architecture

Direct Access (1). If application programs directly access the external data objects using the low-level operations on binary data streams, the domain-specific interpretation of the data stream has to be part of the application program. It is difficult to establish synchronisation of different applications and access control in this scenario.

External Server Access (2). In this scenario, an external server, for example, another DBMS, a document management system, or a special-purpose file system, provides some value-added service. Application programs use the external server API for computing external data items. An external server may provide additional functionality, as for example transaction control or access control.

ORDBMS Access (3). If access is provided via an ORDBMS the application program uses the ORDBMS API for accessing external data. Instead of using special operations the data is manipulated through SQL operations. Because all SQL operations refer to a database schema we have to provide an integration schema. To which extend advantage can be taken of other strengths of ORDBMSs besides query power, i. e., transactions and access control, is strongly dependent on the environment outside the ORDBMS.

These options are not mutual exclusive. Instead different combinations determine the properties of the integration environment.

If data access is performed by using an ORDBMS, then we can choose between two kinds of integration:

Materialized Integration (D_a). Materialized integration means to store a copy of the external data in the ORDBMS. This results in the problem of how to guarantee that all copies are consistent. Updates on the data in the database must be repeated on the external server and vice versa. This has to be done in the integration logic. In domains having strong requirements regarding consistency materialized integration is not adequate.

Virtual Integration (D_b). If virtual integration is chosen, all data remains in the external data source and no copies are stored in the ORDBMS. External data is transmitted to the ORDBMS on demand as part of the query evaluation process. The integration logic has to implement the data transfer. This has two advantages. First, only the data needed for the current operation is transmitted to the ORDBMS and, second, it ensures an operation consistent view. On the other hand, you have to accept higher access costs in comparison to materialized integration.

External data accessible with low level operations needs to be interpreted and converted to the internal format. When using an ORDBMS as integration platform, the decision has to be made on how much integration logic should be implemented as an ORDBMS extension. There are again two options:

i. Tight Coupling. If the ORDBMS is tightly coupled with the external data, it directly accesses the binary data stream using the low-level operations. Hence, the integration logic includes functionality for reading, preprocessing, interpreting, and converting the data. Developing DBMS extensions is very expensive, because developers are restricted in using development tools and programming languages. Furthermore, the programming environment is very complex. Hence, the more integration logic resides in the DBMS the higher are the development costs.

ii. Loose Coupling. In a loosely coupled environment the developer exploits the functionality of an external server. Only data conversion and communication are implemented as a DBMS extension. Additionally, the external server can synchronize access and enforce access control, if applications are allowed to access external data without using the ORDBMS.

Compared to a tightly coupled environment the development costs are lower, because the extension developer can reuse the functionality of the external server.

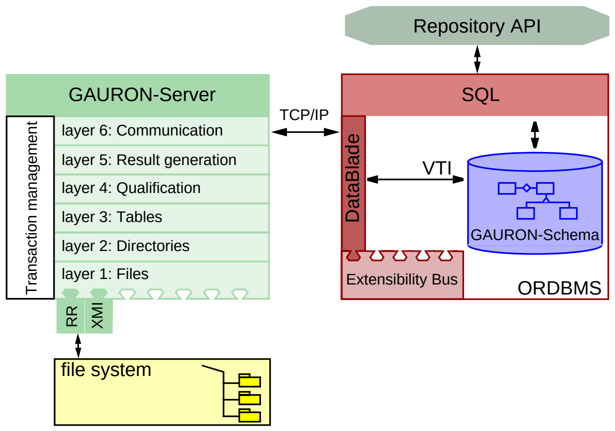

Figure 5. GAURON system architecture

Our reference architecture introduces three ways of data access, two choices for data integration and two options for coupling the ORDBMS with external data sources. This results in many different combinations for building integration environments. A more detailed discussion concerning the properties of the resulting systems is beyond the scope of this paper. Nevertheless, we have found our reference architecture very helpful in designing our implementation of a data integration environment as described in section 4.

4. GAURON: An ORDBMS-based Integration Environment

In the following we discuss our implementation of an integration environment, called GAURON (**Ga**teway to **U**ML-**R**epositories using **o**bject-relational exte**n**sions [19]). First, we motivate why we developed GAURON. Then a system overview of GAURON is given and the implementation aspects are described. We finish this section with some remarks of what we have learned so far.

4.1. Motivation

The application scenario of the UML repository has been presented in the introduction. It was not our intention to build just another graphical UML tool. Instead we examined existing UML tools in order to integrate their data into our repository. Unfortunately, most tools store their data in files and can not be customized to use our repository instead. Therefore, we want to extend our UML repository and enable it to deal with exchange formats like XMI (XML Metadata Interchange [14]) or proprietary file formats. So arbitrary file formats have to be supported.

SERUM tools should access external data in order to analyse UML models, to prove their validity regarding project-specific guidelines, and to use them as an input for code generators [11]. Especially the analysing and the checking of model guidelines take great advantage from the query power of SQL:1999. Therefore a stable database schema has to be provided, i.e., external data should be accessible exactly the same way as internal data.

In addition, manipulations of the external data should be done directly with the third-party tools and cost-intensive round-trips should be prevent.

4.2. System Overview

We chose external tables as concept to integrate external data. External rows where out of question because no extensibility infrastructure exists for this concept and external values does not offer a stable database schema, so we had to modify our tools.

Using the terms of our reference architecture, GAURON implements an integration environment with ORDBMS access, virtual integration and a loose coupling. ORDBMS access is needed to use the query power of SQL:1999 and virtual integration to avoid cost-intensive round-trips. The loose coupling was chosen for strategic as well as implementational reasons. One strategic thought was that using an external server gives us control over the data at a single point. The server can read data from different sources and provide access to the data for different clients. Hence, it can synchronize and control access to the data. The implementational reasons were that development outside a ORDBMS is much easier, less fault-prone, and has less restrictions regarding tools and programming languages.

Fig. 5 shows the GAURON system architecture. It can be seen that the repository API directly uses the SQL interface. Therefore, providing access to external data means to provide an integration schema which offers the same access as used for data directly stored in the UML repository. The only difference is that the tables are implemented as external tables by overwriting their access methods. This is transparent to application programs. The user-implemented access methods are put together in a package called GAURON-DataBlade. The GAURON-DataBlade communicates via TCP/IP with the GAURON-Server, an external process which provides the UML models in an internal exchange format. The GAURON-Server uses wrappers for reading and extracting the data from files in different formats, e. g., XMI.

4.3. GAURON-DataBlade

As extensibility infrastructure we used the VTI introduced in section 2.3. The VTI allows to overwrite operations for read access as well as operations for write access with user-implemented functions. Although write access is supported, too, we did not exploit it for two reasons. First, in our application scenario we do not need it because we use external data access mainly for analysing tasks. Second, proprietary file formats do include data describing UML models and, in addition, tool-dependent data, as for example presentation information, which is difficult to handle if updates are possible. Therefore, we have just implemented the scan operation. In Fig. 3 you can see which functions are called during a scan and therefore had to be overwritten.

4.4. GAURON-Server

As can be seen in Fig. 5 the GAURON-Server consists of six layers. Each layer abstracts from the layer below. This has two advantages: First, a better understanding of each layer and, second, a more flexible implementation, e. g., a whole layer can be exchanged. Thus, our GAURON-Server can be used not only to read UML models out of files but also to retrieve data from another DBMS. In this case, only layer 4 has to be changed. Additionally, some tasks, e.g. qualification and projection can be delegated to the external DBMS.

Beneath the six layers there is an independent component, the transaction management. The GAURON-Server offers an open interface to transaction management. Using this interface external applications can synchronize their file access with the GAURON-Server. This enables them to access files directly (read and update files, add or remove files from a directory) and not violate isolation. If tools are not cooperative, synchronization has to be done at the organizational level. Our GAURON-Server works on a private directory tree. Files are exchanged between the private area and the public area with a checkout/checkin tool which cooperates with the GAURON-Server's transaction management. External tools must work on files in the public area.

4.5. What we have learned so far ...

The realization of GAURON has proven that ORDBMSs are a suitable technology for integrating heterogeneous data sources in our context. The architectural options chosen (ORDBMS access & direct access, virtual integration, loose coupling) have lead to an integration environment which meets our requirements. GAURON enables us to do both, developing and maintaining UML models with external tools and using SQL for model checking and analysing. At the same time we avoid to copy the UML models in a time-consuming process into the ORDBMS at each round trip.

GAURON allows all three categories of access introduced in section 3. While cooperative tools can read and write files in place we have to use our checkout/checkin mechanism for other tools. In both cases, working with the files is easy and fast. We benefit from the loosely coupled architecture, because the external server can synchronize data access. The virtual integration avoids consistency problems and allows to implement the fast checkout/checkin mechanism at the file level. On the other hand, SQL queries run slower than using materialized integration, because of the communication overhead, the additional transaction management and the time consuming data extraction process. We think that future enhancements, like a cache in the GAURON-Server, will increase performance significantly.

Because of the layered design of the GAURON-Server, integration of data sources different from files is possible. Maybe some layers may not be needed anymore, if their functionality is already implemented in the new data source. We plan to integrate another DBMS in layer 4 but this requires some additional examinations about synchronization. The concept of wrappers for file access enables us to add new file formats without changing the GAURON-Server.

Furthermore, we had to learn that the extension interfaces are difficult to use and that developing a DBMS extension is much harder work than building usual applications. Although the GAURON-DataBlade does only consist of a few functions, it required a lot more realization efforts than the much more complex GAURON-Server.

5. Conclusions & Outlook

In this paper, we have discussed ORDBMS-based data integration environments. ORDBMSs are a suitable technology for this purpose for the following reasons:

- ORDBMSs provide an extensible data-model. Using the expressive power of the object-relational data-model allows to define integration schemas which reflect the data stored in external data sources in a natural way.
- ORDBMSs provide an extensibility infrastructure, which enables an extension developer to implement external values, external rows or external tables. These concepts support the implementation of a broad range of integration environments.

Because ORDBMSs represent a new technology, so far little experience is available for such systems, especially within our context. Therefore, we have introduced a reference architecture in order to categorize the architectural options available. The three categories were data access, data integration, and data source coupling. Future work will be concerned with the development of a 'cookbook' which relates a specific combination of architectural choices to the properties of a resulting integration environment. For this purpose, we will further refine our reference architecture.

Last but not least, we have shown that the contents of this paper is not just paperwork. The GAURON system is a running example, illustrating that ORDBMSs are a suitable technology for integrating heterogeneous data sources. The chosen architecture has proven to fit our requirements. Further work will enhance the systems functionality in order to cope with requirements from other application scenarios.

Altogether, an ORDBMS given to a skilled team of developers can make look (almost) everything like a row in a table, but note that not everything that could be treated as a row should be treated as a row.

6. References

[1] ANSI/ISO/IEC 9075-2-1999: Database Languages - SQL - Part 2: Foundation (SQL/Foundation). American National Standard Institute, Inc., 1999

[2] P. A. Bernstein, B. Harry, P. Sanders, D. Shutt, J. Zander: The Microsoft Repository. Proceedings of 23rd International Conference on Very Large Data Bases, 1997, Athens, Greece, Morgan Kaufmann 1997

[3] P. A. Bernstein, T. Bergstraesser, J. Carlson, S. Pal, P. Sanders, D. Shutt: Microsoft Repository Version 2 and the Open Information Model. Information Systems 24(2)

[4] J. A. Blakeley: Data Access for the Masses through OLE DB. Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, 1996, ACM Press 1996

[5] M. J. Carey, N. M. Mattos, A. Nori: Object-Relational Database Systems: Principles, Products, and Challenges (Tutorial). Proceedings ACM SIGMOD International Conference on Management of Data, 1997, Tucson, Arizona, USA. ACM Press 1997

[6] S. Chaudhuri, U. Dayal, T. W. Yan: Join Queries with External Text Sources: Execution and Optimization Techniques. Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, 1995, ACM Press 1995

[7] Informix Press: Extending Informix Dynamic Server.2000. Version 9.2, 1999

[8] Informix Press: Virtual Table Interface - Programmers Manual. Version 9.2, 1999

[9] ISO/JTC 1/SC 32: Database Language SQL - Part 9: SQL/MED, Final Committee Draft ISO/IEC FCD. November 1999

[10] M. Jaedicke: New Concepts for Parallel Object-Relational Query Processing, Ph. D. thesis, University of Stuttgart, 1999

[11] W. Mahnke, N. Ritter, H.-P. Steiert: Towards Generating Object-Relational Software Engineering Repositories. Tagungsband 8. GI-Fachtagung „Datenbanken in Büro, Technik und Wissenschaft" (BTW'99), Freiburg, March 1999, Springer Verlag

[12] N. M. Mattos, J. Kleewein, M. T. Roth, K. Zeidenstein: From Object-Relational to Federated Databases. Tagungsband 8. GI-Fachtagung „Datenbanken in Büro, Technik und Wissenschaft" (BTW'99), Freiburg, März 1999, Springer Verlag

[13] M. Stonebraker, M. Brown: Object-Relational DBMSs: Tracking the next great Wave. Morgan Kaufmann Series in Data Management Systems, 1999

[14] OMG: OMG UML v. 1.3. OMG Document ad/99-06-08

[15] N. Ritter, H.-P. Steiert: Enforcing Modeling Guidelines in an ORDBMS-based UML Repository. International Resource Management Association Conference 2000, Anchorage, Alaska, May 2000

[16] A. Sheth, J. Larson: Federated database systems for managing distributed, heterogeneous, and autonomous databases. ACM Computing Surveys, 22(3), September 1990

[17] M. T. Roth, M. Arya, L. M. Haas, M. J. Carey, W. F. Cody, R. Fagin, P. M. Schwarz, J. Thomas, E. L. Wimmers: The Garlic Project. Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, 1996. ACM Press 1996

[18] M. T. Roth, P. Schwartz: Don't Scrap It, Wrap It! A Wrapper Architecture for Legacy Data Sources, VLDB Conference 1997

[19] A. Weber: GAURON - Gateway to UML-Repositories using object-relational Extensions. diploma thesis, University of Kaiserslautern, October 1999 (in german)