Design Issues of Database Access in a CORBA Environment*

Ulf Leser, Stefan Tai
Technische Universität Berlin
Sekr. E-N 7, Einsteinufer 17
D-10587 Berlin, Germany
E-mail: {leser, stai}@cs.tu-berlin.de

Susanne Busse
Fraunhofer ISST
Mollstr. 1
D-10178 Berlin, Germany
E-mail: Susanne.Busse@isst.fhg.de

Abstract

CORBA is an architecture aiming to provide an infrastructure for the cooperation of distributed and heterogeneous software components in a network. One of the most important components of real-life applications are database systems. In this paper we investigate the use of CORBA to access data stored in a database system. We evaluate the design decisions a developer has to take if he wants to use CORBA to make his data accessible for remote clients, grouping them into five different dimensions. We also evaluate the usability of the Object Query Service and the Object Transaction Service, as specified by the OMG. This leads to the identification of three different scenarios, each of which uses CORBA in a completely different manner, which we call "Old-Fashioned", "Object Collections", and "Pure CORBA".

1 Introduction

Software engineering today has to deal with the situation that systems have to be build in consideration of existing software programs. These programs form autonomous components that are heterogeneous in their platforms, programming languages, application logic, and semantics. The appropriate infrastructure to build distributed heterogeneous systems out of such components is provided by so-called middleware, and one of the most prominent examples is the *Common Object Request Broker Architecture* (CORBA) of the OMG. CORBA defines an object-oriented environment for the interconnection and interoperation of diverse software components.

One of the most important components of many software systems are database systems (DBS) providing several important services for data management, such as automatic backups, transaction management and flexible access through high-level query languages. Database systems in general appear in different roles in the context of distributed object systems; in our context we consider them as existing software components that need to be integrated in the embracing information and communication infrastructure.

*This work was partly supported by the German Research Council, Graduate School on Distributed Information Systems (DFG grant no. GRK 316) and Graduate School on Communication Systems (DFG grant no. GRK 86), and EMBO Fellowship Grant ALTF ASTF

As an open architecture, CORBA allows a variety of strategies to integrate DBS. We distinguish between logical and technical issues of software system design, and take a closer look at the object query and the object transaction service, which are designed to support the development of such applications.

Outline. The paper is organized as follows. Section 2 briefly introduces the basic properties of CORBA and the class of database systems that we are addressing. Section 3 discusses issues in the interface design for the integration of an existing DBS. We particularly consider the question of object and query representation and the potential impact of middleware object services. In Section 4, we outline three different strategies which are reasonable combinations of the options described in section 3 and evaluate them in their ability to cope with typical database problems such as schema evolution, ad-hoc queries and the support of transactions. Section 5 discusses related work, and section 6 concludes.

2 Setting the Scene

2.1 CORBA-based Computing Infrastructure

The Object Management Group's (OMG) *Common Object Request Broker Architecture (CORBA)* is a technology standard for integrating diverse software components in distributed, heterogeneous environments [10, 13, 15]. CORBA specifies a software bus architecture based on an object-oriented communication model and the client-server paradigm, hiding implementation details of distribution. CORBA is part of the OMG's *Object Management Architecture (OMA)* that defines a reference model for distributed object systems, as is depicted in Fig. 1.

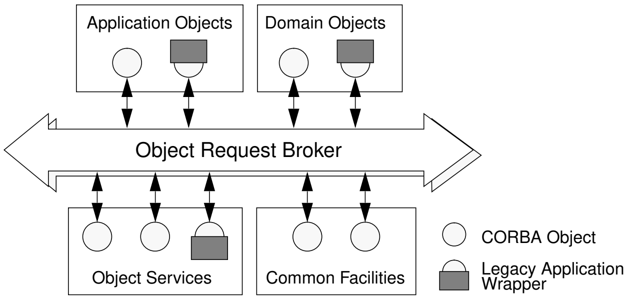

Figure 1. Object Management Architecture

Software components that are to interoperate in a CORBA environment must comply to the specific communication mechanisms and standards as defined with the OMA and CORBA. This includes the use of the interface definition language OMG IDL for specifying object interfaces (types as sets of operation signatures) for components, and for defining data structures for data exchange (operation parameter types may be IDL-defined data structures or interfaces). Objects offering an IDL interface are maintained by a server, and clients issue requests via the ORB to server objects (clients may use a naming service to obtain object references). The CORBA interface model not only enables encapsulation of incompatible component implementations, but also allows the CORBA objects to be of varying granularity: Small programming language objects equally qualify as CORBA objects as entire subsystems, or databases.

This question of object granularity is of predominant importance for the integration of database systems and for discussing data exchange in CORBA environments. How are database systems and the data managed by these represented in the CORBA world? What information is considered first-class citizens (objects), and what information is accessible only implicitly through method calls (data structs)?

In the following, we summarize the two CORBA object services Query and Transactions, which are relevant to answer and discuss the different options and their consequences.[1]

2.1.1 The Object Query Service (OQS)

The OQS [11] is a simple, generic service for the distribution of queries in a CORBA environment. Objects implementing the OQS interfaces can receive queries that are formulated using defined query languages: the OQS requires the acceptance of SQL-92 or OQL [5] queries, at least.

Queries are represented as either strings or special query objects that support multiple execution. They are distributed using a minimal collection interface that passes a query to all its members. Each member is either again a collection, which leads to a recursive transmission of the query, or an object evaluating the query itself. Questions such as query optimization or query decomposition are not handled by the OQS, nor does the OQS define methods for the retrieval of meta-data such as schema.

2.1.2 The Object Transaction Service (OTS)

The OTS [12] provides distributed transaction processing facilities on top of CORBA, building on the notion of transactions as employed in database management systems and TP (Transaction Processing) monitors of client-server systems. The OTS defines a number of interfaces and interaction models for processing (nested) transactions and for driving the two-phase commit protocol between multiple objects that may manage persistent data by different means (flat files and/or databases). One processing model addressed concerns the use of native DBS transaction facilities, making use of X/Open XA switches of databases (leaving responsibilities to the database system). Other models represent persistent data as own objects implementing specific OTS interfaces (giving responsibilities to the CORBA OTS managing component).

The OTS aims to guarantee transactional properties to all programs that run under its supervision. It handles all resource registrations, up to the coordination of any rollback or commit of a transaction. As databases typically provide transaction facilities, the OTS may well make use of these features. Database integration in CORBA for the OTS therefore is naturally implemented by using the OTS' XA interoperability features.

[1] In this paper, we do not regard the *Persistent Object Service (POS)*, nor the *Persistent State Service (PSS)* which are also concerned with DBS integration. The POS has been subject of controversial discussions, and the PSS is still in its specification process. A proper treatment of these CORBA services should be done once the PSS specification is released.

2.2 Relational and Object-Oriented DBS

We focus attention on relational and object-oriented database systems, but treat them in a uniform way. We therefore assume that the database which is to be interfaced is based on a data model similar to an Extended Entity-Relationship model: data is structured using classes with typed, possibly multi-valued, attributes; classes are arranged in a specialization hierarchies; objects have a unique identity; objects can be connected through uni-directional associations. We presuppose that every DBS schema can be unambiguously mapped to OMG IDL, at least those parts necessary for query formulation. We ignore issues such as keys or cardinality constraints whose consideration will anyway take place within the DBS. We further assume that the DBS offers access through a *high-level query language*. Our understanding of queries, and the principle DBS integration architecture using a database wrapper, are as follows:

Queries. We consider queries as "*a mapping from instances of a fixed input schema to instances of a fixed answer schema that is computable and generic*" [1], restricted to projections, selections, joins and aggregations (excluding, for instance, recursive queries). Typical query languages are SQL-92 or OQL. Query properties that need to be considered when integrating DBS in CORBA are:

- The input and the target schema are not necessarily the same. A query can be a *restructuring* query, i.e. the query produces a target schema which is not a subpart of the input schema. Restructuring queries are commonplace in existing DBS, for instance for the definition of *views*.
- The result of a query is in general not identical to an object of the underlying database. This raises difficult questions: What is the type of the result ? Where is this type placed inside the type-inheritance tree ? If the query combines data from different objects, what is the object identity of the result [9] ?
- Ad-hoc queries require knowledge of the database schema. Interfaces allowing ad-hoc queries are only useful if they also provide mechanisms to retrieve the schema (meta-data).
- The actual performance of queries in languages such as SQL and OQL highly depends on query optimization.

Database Wrapper. A *database wrapper* is a CORBA application server to ORB clients, and a client to the integrated database. The wrapper offers IDL interfaces to clients for reading or modifying data stored in the database, and so provides a higher abstraction layer between clients and databases. A simple three-tiered system architecture with a database wrapper is depicted in Fig. 2.

The database wrapper translates client calls into native calls against the DBS, using standard techniques such as JDBC or ODBC, or proprietary protocols like the OCI (Oracle Call Interface). The wrapper therefore must take care of any necessary pre- and post-processing for data interchange, including client access authorization. The wrapper as a client to the database uses the DBS interfaces, so that queries, for example, will in general be evaluated inside the DBS. Wrapper functionality is implemented by one or several, possibly fine-grained, objects.

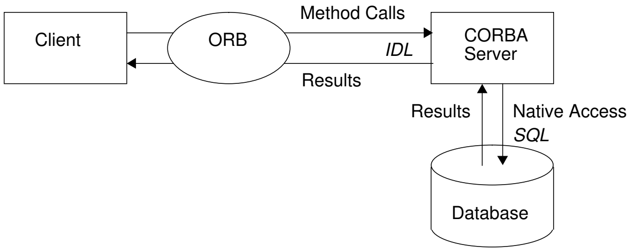

Figure 2. Database Wrapper

3 Design Dimensions

In this chapter, we identify relevant design issues for database-centered applications in a CORBA environment. We distinguish between *logical issues*, addressing the representation of data and queries, and *technical issues*, such as server activation policies or usage of CORBA object services.

3.1 Logical Dimensions

3.1.1 Representation of Database Objects

We distinguish three different ways in which database objects can be represented in the CORBA world. First, database objects are treated as binary data streams and are not interpreted in any way by the CORBA middle layer. Second, the data of a database object is stored in a struct which conforms to the database class definition. And third, each database object is represented by a corresponding CORBA object.

- **R1: As Raw Data.** In this case, database objects do not exist as identifiable entities in the CORBA world. Concerning the ORB, they are unstructured binary datastreams. These streams must be parsed by the client using a fixed encoding of their inner structure.

- **R2: As Structs.** The structure of a database object can be represented in an IDL struct by simple copying all attributes. IDL specifically supports *sequences* which can model multivalued attributes. Upon requesting a database object, the client gets back a struct equivalent to the class definition. Structs can be nested to represent complex attribute types and inheritance; multi-valued attributes are stored in proper sequences or generic set-classes. IDL structs for each database class can be created automatically using the schema information. Structs carry the object's data to the client preserving the type. All subsequent operations are performed locally, reducing network traffic and enhancing performance, but complicating integrity preservation in case of multiple clients working with the same object.

- **R3: As Objects.** In this approach, each database object is represented by an equivalent CORBA object. Therefore, database classes are translated directly into IDL interfaces, which can be carried out automatically. If a client requests a database object, he will get back an identifier, but not the actual data, which remains on the server. Each attribute access will result in a remote call, which increases network traffic but avoids certain problems of outdated data.

In principle, each database object could be registered with the ORB during an initialization phase. This would enable these objects to be accessed directly by clients using the CORBA identifier. However, the administration overhead inside the ORB is considerable, as the number of objects can be tremendous. Furthermore, the question of persistent identifiers is not sufficiently solved yet.

Another possibility would be to use the CORBA level as a further level of abstraction by providing interfaces which in fact are views onto the database schema. Such interfaces can not be generated automatically and the mapping between interface and database schema must be adjusted after each schema change.

3.1.2 Queries

Query languages provide flexible mechanisms for the restructuring, projection, selection and aggregation of data. Although it is often desirable to export all these possibilities to clients, there are many applications where this is not necessary. We distinguish three basic approaches: queries are either hard-coded inside the wrapper objects, or they are restricted to result in proper database objects, or clients can pose arbitrary ad-hoc queries.

- **Q1: Fixed Queries.** Fixed queries are a common method to hide all implementation aspects of the database to the client. The client does not need to have any knowledge of the database schema, the query language, the access path etc., but just has the choice between a certain number of methods, each representing a fixed, though possibly parameterized, query. In general, the result of these methods are not restricted by the database schema, as arbitrary processing can be carried out in the embracing method's code; if this is not the case, R1 - R3 can be used.

- **Q2: Class-restricted Queries.** This approach captures all queries of the form: "Give me all objects of class X which have...". This has a number of consequences:

  - The result is always a uniquely identifiable database object. Restructuring queries are not possible. However, joins are still allowed for the specification of conditions.
  - In many cases projections are vital for better performance, for instance projecting out the image of an employee when the name suffices. Projections are still possible here, but clients must be able to determine whether a value is missing in the database, or whether it was only projected out by the query. The object identifier must always be retained to allow propagation of changes.

  Results, which are always objects of a predetermined class, can be represented using either R2 or R3. R1 makes little sense, as R1 misses the notion of classes. This approach is identical to Q3, with the restriction that the "select" part of each query is bound to attributes of one class only and must contain the object ID.

  Although Q2 seems to be quite restrictive, it is certainly sufficient for many applications, for instance digital libraries.

- **Q3: Arbitrary Ad-hoc Queries.** Here, clients are able to pose arbitrary queries, exploiting the full power of the underlying database query language. Queries are passed as strings, which are directly transfered to the database server for execution.

	R1: Data		**R2**: Structs		**R3**: Objects	
	I1	I2	I1	I2	I1	I2
Q1: Fixed	x	x	o[a]	x	x	x
Q2: Restricted	-	-	o[a]	x	-	x
Q3: Ad-Hoc	x	-	-	-	-	-

Table 1. Combinations of dimensions
'-': impossible or not reasonable; 'x': well possible; 'o': possible; [a]: yields large interfaces, better split up by classes

This is only reasonable in combination with R1; both R2 and R3 would be extremely expensive, as the middleware would need to parse the query and dynamically determine the type of the result.

Usually the result of a query is a set of entities. Any of the above methods will result in (homogeneous) collections of structs or objects for R2 or R3, respectively. Using R1, the result can either be a collection of strings, or all these strings are in some recognizable way concatenated.

3.1.3 Interface Granularity

So far, we did not treat the question of how access methods can be grouped into CORBA interface definitions. We distinguish two basic strategies: the first will pack all database related methods and definitions in one big object, the second splits access methods according to database classes.

- **I1: One Database Interface.** All database relevant features are combined in one single interface. This is reasonable for systems using R1; the extreme case is an interface having only one method which takes strings as queries, which are passed to the database server to return the result as strings again. Using R2, the interface will probably get uncomfortably large, as it must include the struct definitions of all classes. R3 requires the definition of separate interfaces for each class.
- **I2: Per Class Interfaces.** One CORBA interface per database class is defined, including all access method and type definitions. This is not reasonable in the case of R1 (as R1 does not know objects), well possible for R2, where the structural part of each class is defined in one struct, and necessary for R3, where database attributes are mapped to interface attributes.

Table 1 summarizes the compatibility between the different dimensional categories.

3.2 Technical Dimensions

This section discusses technical design dimensions that are specific to the OMA, namely activation of servers in a CORBA environment, use of static versus dynamic invocation mechanisms, and potential impacts on software system design when using CORBA-Services like the OTS.

3.2.1 Server Activation

CORBA distinguishes three different kinds of server activation modes. In case of a *shared server*, client requests for objects of a server are all routed to the same server process. With *unshared servers*, requests for different objects are routed to different server processes. Using the *per-method* mode, each method call of each object results in a new server process. For the first two policies, one furthermore has to choose between *per-client*, *per-client process*, or *multiple-clients* activation, which determines whether calls from different clients or principals are routed to the same or to different objects [3].

Choosing a server activation policy has an impact on the degree of parallelism that can be achieved for client queries. To increase response time, all major DBS already are capable of serving multiple connections: Clients can send queries in parallel without having to wait for termination of earlier queries. This property should be propagated to clients connecting through the DBS wrapper as well. However, the wrapper, which is a process on its own, can easily become a bottleneck. If the *shared server* mode together with *multiple-clients* is used, all requests will be directed to the same wrapper. Receiving a request, this wrapper object will block all further requests until termination of the first. This should be avoided.

Activation mode policies are somehow subsumed by the use of threads in the wrapper object. However, programming with threads is considerable more complex than programming with processes, and threads are also not available on all platforms, which affects portability of applications. Furthermore, they require thread-safe libraries for the low-level database access [6, 17].

Server activation mode is especially important when I1 is chosen; then, all accesses are possibly routed through one single object. On the other hand, chosing I2 together with R3 and unshared-mode, will result in a great number of processes: one process for each access to each database object. If per-method mode is used, it will even be one process for each attribute access.

3.2.2 Static versus Dynamic Invocation

CORBA offers two different mechanisms of how clients invoke server operations. With *static invocation*, clients know at compile-time the IDL of the server and can use stubs that are generated from the IDL specification. If the IDL of the server changes, however (for example, reflecting a schema change in the database), both client and server must be recompiled and re-linked. An alternative is to use *dynamic invocation*, where a client queries at runtime the ORB's *interface repository* to determine IDLs and supporting servers (use of the *dynamic invocation interface (DII)*), and where servers may register new interfaces dynamically (use of the *dynamic skeleton interface (DSI)*).

Dynamic invocation may be used to better cope with schema evolution in the database. This only applies if R2 - R3, respectively Q1 or Q2 are used, as these are the cases where the database schema is reflected in IDL.

- **DSI.** The DSI could be used for two purposes: First, if database objects should be directly mapped to CORBA objects (R3) and arbitrary queries should be allowed (Q3), the wrapper could parse the query, determine the type of the result and dynamically construct a compliant interface. However, we believe that this is only a theoretical possibility, because of the tremendous increase in complexity of the wrapper code. Second, if R2/R3 combined with Q2 is used, the server code could be made resistant against schema changes by the following procedure: when informed of a schema change, the wrapper queries the data dictionary of the database to achieve the updated schema. Then, it dynamically constructs the new interfaces and announces them using the DSI.

Again, we do not believe that the increased effort justifies the gain in flexibility. Therefore, we will not consider DSI in the following.

- **DII.** In contrast to the DSI, using the DII is a more attractive mechanism as it adds flexibility on the client side. This is of importance for the following reasons:

  - It is difficult to inform all potential clients of a schema change in the database.
  - Clients potentially use many database services - reflecting each servers' changes by a re-compilation of the code can become infeasible.

  A client may use the DII in the following way: upon startup, it queries the interface repository of the appropriate ORB for the interfaces exported by the database wrapper, and creates each call according to these interfaces. If the interface changes, the client only has to be restarted. Specifically, in the case of R2 / R3, the client will extract a IDL description of the result of a query, giving him the possibility to react properly.

3.2.3 Using the OTS

The introduction and use of CORBA object services like the OQS or the OTS describes another technical impact on software system design and object representation.

Consider for example the OTS, which defines an IDL interface `Resource` that offers operations similar to the X/Open XA interface for driving the two-phase commit protocol (like prepare, rollback, or commit). All persistent data that is affected by a transaction is a `Resource` in the OTS context (if not, alternatively, the XA switch of a DBS is used directly), which suggests a fine granularity level for transactional data objects: all data is to be identified and registered separately to the OTS as separate, own CORBA objects offering the IDL `Resource` interface. Use of the OTS in this respect may be an argument for choosing R3.

3.2.4 A Closer Look at the OQS

The OQS specifies basically two interfaces. *QueryEvaluators* (QE) take queries as strings and evaluate them, possibly by delegation. The specification requires each QE to accept queries either in OQL (or a subset of OQL, OQL-basic) or SQL-92. This makes it very likely that a QE will usually delegate the query execution to an underlying DBS.

The second important interface of the OQL arranges sets of objects in *QueryableCollections* (QC). QCs are simple collection-type objects with a query method inherited from the QE interface. Because these collections can have heterogeneous members, for instance other collections, they offer an elegant way to distribute queries; this does however not include the decomposition or translation of queries.

So which of the possible and reasonable combinations of the design dimensions described in Section 3.1 and 3.2 are supported by the OQS? QE expect queries as strings; this directly meets ad-hoc queries (Q3), and makes Q2 a tedious task, requiring expensive pre-processing of each query to check whether it conforms to the restrictions. Q1 is not in the scope of the OQS.

The representation of the results is not determined by the OQS. As ad-hoc queries are allowed, R1 is the natural choice. From Table 1 it is clear that the concept of classes is only reasonable when supported both in the interface, on the query level and for data representation. Although this seems to be advantageous in many areas, the OQS does not support it in the first place. The closest 'match' would probably be achieved if queries are restricted to classes (Q2), and QCs are used to represent query results. This nice picture of iterative querying (the result of a query against a QC is again a QC, containing now only the positive subset of members for this queries) also seems to be envisaged in the original OMG specification[18], but does not fit well with the permission of arbitrary queries.

Another question concerns the support for meta-data. As mentioned in Section 2.2, ad-hoc queries require schema knowledge in the clients. Interfaces such as JDBC offer special methods that access data dictionaries provided by all commercial DBS to read class names, attribute names etc. The OQS does not contain such methods. A reason for this might be that also systems not capable of meta-data support should be addressable, but this is somehow inconsistent with the stipulation of SQL or OQL capabilities.

4 Integration Strategies

In this section, we present three different integration strategies. They all use CORBA for database integration, but each follows a distinct flavor of programming style, therein representing different combinations of the dimensions described earlier. Of course, other combinations make also sense in certain situations. We have chosen those three discussed here to demonstrate the consequences of basic design decisions for the integration of DBS.

4.1 Old-Fashioned

With the term old-fashioned we summarize an approach that follows the spirit of conventional client-server programming. Clients submit their queries as strings or use hard-coded, fixed queries (Q1 / Q3), and receive results as strings or collections of strings. The middleware has no notion of the schema of the underlying database. It is a combination of R1 - Q1/Q3 - I1.

Although this approach is truly old-fashioned, it has a number of advantages. Using Q3, clients can use the full power of the query language of the DBS. As there is no schema representation encoded in the middleware layer, it is immune against schema changes in the database. Furthermore, queries are transmitted 'as it', and can hence by well optimized by the DBS.

Disadvantages include the tiresome parsing of all results on the client side. This becomes a challenging task if the transmitted data types are for instance multimedia objects such as images or audio data, or if an integration of multiple databases is needed. Because the ORB does not know the database objects, they can not make use of any object service. The absence of schema data in the middleware requires that clients are developed in close collaboration with the server, passing meta information through some other “channel”, probably by human communication. This does not suit well to the general spirit of distributed object computing, where objects should be rather self-describing and should be usable from any client within the network without human interference. This problem can be alleviated by implementing some methods that provide meta-data.

An old-fashioned approach is supported by the OQS in a trivial way, as it certainly will implement a query method as the QueryEvaluators. The OTS also supports the old-fashioned approach as the OTS offers XA interoperability: the XA interface of the DBS is used.

4.2 Object Collections

Object collections describes a useful strategy for applications that do not need restructuring queries. For instance, a digital library service is usually based on queries that specify conditions on attributes of publications, such as the appearance of certain keywords in the abstract or a certain authorship. The result is a list of publications; it makes little sense to construct queries that result in a combination of the first name of an author with the ISBN number of another publications. In such an environment, a combination of R2 - Q2 - I2 is a proper choice. I1 would offer no advantages over I2, but leads to less readable interfaces.

This results in a comfortable, fully typed database interface. Clients can retrieve the struct definitions through the ORB's interface repository, and, using DII, they can flexibly react on schema changes. Schema evolution only requires a re-compilation of the server. Once an object is passed to the client, reading attribute values does not require any remote calls any more.

However, this strategy does not offer full query power. It is also not clear how query conditions can be specified; using SQL or OQL is not possible. Simple, template-based mechanisms will fail to construct complex conditions that require joins between database classes. Furthermore, it is difficult to retain integrity in the database if two clients have requested the same object and perform changes on its data at the same time. This can for instance be done by putting a read-lock for every object that is read by a client, until this client releases the object again.

The OQS can not easily be adopted to this scenario, as discussed in section 3.3. It is also not clear how the OTS could be used to manage problems as described above, because the "objects" involved are actually no CORBA objects, but simple structs.

4.3 Pure CORBA

With "Pure CORBA", we refer to a strategy that fully employs the object-oriented paradigm within CORBA: All database objects are represented as CORBA objects. As seen in Section 3.1, this makes the use of ad-hoc queries almost impossible, so we have to fall back on class restricted queries. Hence, we can only follow a combination of R3 - Q2 - I2.

This is similar to the strategy discussed in the previous subsection. The only, but significant difference is the existence of CORBA objects instead of structs. Consequently, CORBA services can be applied over these objects. For example, the OTS may be used to guarantee integrity in cases as described above. However, objects are not passed by value in CORBA, but only by reference. Therefore, each access requires a remote procedure call, which creates considerable network traffic.[2] One further has to distinguish different models of data access: the created CORBA object can either read all data from the database at once and then answer client requests from its internal data, or the object delegates each single access to the database server. In the former case, the integrity problem exists

[2] Recently, the OMG has come up with a proposal for passing objects by-value.

	Old-Fashioned	Object Collections	Pure CORBA
Query restrictions	None	Full conditions, restricted selection	Full conditions, restricted selection
Visibility of schema for clients and ORB	Requires special methods	Through IDL	Through IDL
Development expense (middleware)	Low (generic approach)	Could largely be automated	Could largely be automated
Nr. of objects handled by ORB	One	Nr. of classes	Very many; potentially nr. of database objects
Nr. of remote calls	Low; per complex query	Medium; per object	High; per attribute access
Effects of schema evolution	Many in the client, none in the server	If client uses DII, no effort; Server generates new IDL	If client uses DII, no effort; Server generates new IDL
Query optimization	In the DBS	possible on per-object basis	Impossible; many different small queries
Compatible with OQS	Trivially	Unclear	Unclear
Compatible with OTS	Yes	Difficult	Yes

Table 2. Properties of integration strategies

again (but only among objects on the same machine), while in the later case, the database automatically enforces integrity, but for the price of an increased number of accesses. In this case, query optimization becomes virtually impossible.

Table 2 summarizes the different properties of the discussed strategies.

5 Related work

The MIND project [6, 7] uses CORBA as the middleware for a system of federated databases, both relational and object-oriented. They describe several design decisions they made for this purpose: the use of unshared server activation mode for parallel query execution; static method invocation, because they assume a static database schema; and the decision to only register the database gateway as a CORBA object, to avoid extensive IDL coding and to be immune against schema evolution. Hence, they took an approach similar to our 'Old-Fashioned' scenario. They do not treat CORBA services, probably because they were not yet readily specified during the writing of these papers. Some experiences in the integration of a scientific database can for instance be found in [14]. Semantic issues are treated in [4].

Wells and Thompson [18] give a critical discussion of the four original submissions to the RFP for the OQS. It points out many weak points that can still be found in the final specification, such as missing meta-data services, no treatment of query optimization or indexing, and a generally insufficient perception of queries.

Baker [2] discusses integration of DBS into CORBA. For OODBS, especially *Object-*

Store, he describes a special-purpose object adapter which was developed by IONA [8] (now also existing for *Versant*). This circumvents the overhead that is imposed by the standard *basic object adapter* (BOA) in the case of fine-grained database objects. It seems that the OMG is following a similar strategy with their recent RFP for a *persistent object adapter* (POA). Baker [2] furthermore highlights the advantages of using CORBA, especially the portability of the IDL. For the integration of RDBS he proposes two ways of representing database objects, which are similar to 'Pure CORBA' and 'Old-fashioned'.

6 Conclusions

This work discusses the integration of an existing DBS into a CORBA environment. There are many ways how this can be achieved. We described some of the logical and technical dimensions that make up a complex design space.

There is for instance a strong relation between the way database objects are represented and the expressiveness of queries. Without creating an immense overhead, it seems that full query capabilities, including restructuring queries, exclude proper object-oriented representation; and requiring typed data excludes restructuring queries. This problem is similar to questions that arise in the context of object-oriented query languages and database systems, but they are worse in the CORBA context, because clients and servers are usually developed independently. The 'old-fashioned' strategy admits arbitrary ad-hoc queries, but does not follow the general flavor of the OMA.

Three aspects are particularly important from a database point of view: maintainability, performance, and data integrity. The latter two clearly mismatch – high performance is only achievable by following a struct-based approach together with a optimistic locking strategy, but this makes integrity preservation difficult. Integrity can be enforced by a fine-grained locking of objects, but this requires a large number of objects to be handled by the ORB. Maintainability is achieved if clients are largely immune against schema changes, which is only possible through a proper representation of the database schema in the middleware (R2/R3), combined with dynamic interface invocation at the client site. Performance also depends on query optimisation, which is in general better if large queries are submitted to the database engine, as vey few commercial DBMS are capable of inter-query optimisation.

Some of these problems can be alleviated by using the OTS, which is a powerful and flexible service that is independent from the underlying database (as long as it supports the XA interface), data model or data schema. This is not true for the OQS. Queries depend on the schema, and the schema depends on the data model. We think that the OQS is not addressing these problems properly. It appears to be the least common denominator of a number of different and incompatible DBS techniques, where this denominator is so small that it becomes almost useless.

Our evaluation of the query service yields very little advantage one could gain by using this, which might be a reason for the fact that vendors do not seem to have an interest in implementing this service.

The three scenarios we have sketched in this work are on an increasing scale regarding their integration into CORBA. While the 'old-fashioned' approach uses the middleware basically as a comfortable RPC mechanism, 'Object Collections' also greatly benefit from the system- and language independent IDL. IDL is herein used as a common data model for data exchange. 'Pure CORBA' eventually fully integrates into the bigger picture of self-describing, interoperable and distributed components.

References

[1] Abiteboul, S., R. Hull, et al. (1995). *Foundations of Databases.* Reading, Massachusetts, Addison-Wesley Publishing Company.

[2] Baker, S. (1996). *CORBA and Databases - Do you really need both ?* Object Expert, May 1996.

[3] Baker, S. (1997) *CORBA Distributed Objects using Orbix*, ACM Press, Addison-Wesley.

[4] Busse, S. and R. D. Kutsche (1997). *Objektbasierte Integration einer externen heterogenen Informationsbasis.* Technische Universität Berlin, Informatik Fachbericht 97-7.

[5] Cattell, R. G. G. and D. K. Barry, Eds. (1997). *The Object Database Standard: ODMG 2.0.* San Francisco, California, Morgan Kaufmann Publishers.

[6] Dogac, A., C. Dengi, et al. (1996). *A Multidatabase System Implementation on CORBA.* 6th Int Workshop on Research Issues in Data Engineering: Nontraditional Database Systems, New Orleans, Louisiana.

[7] Dogac, A., C. Dengi, et al. (1998). *Building Interoperable Databases on Distributed Object Management Platforms.* Communications of the ACM (to appear).

[8] IONA Technologies (1996). *ORBIX + ObjectStore Adapter.* IONA Technologies PLC White paper.

[9] Kim, W., Ed. (1995). *Modern Database Systems. The Object Model, Interoperability and Beyond.* New York, ACM Press, Addison Wesey Publishing Company.

[10] Object Management Group (1995). *The Common Object Request Broker: Architecture and Specification*, Rev. 2.0. (available at http://www.omg.org)

[11] Object Management Group (1997). *CORBAServices: Common Object Service Specification: Query Service Specification.* (available at http://www.omg.org)

[12] Object Management Group (1997). *CORBAServices: Common Object Service Specification: Object Transaction Service Specification.* (available at http://www.omg.org)

[13] Orfali, R., D. Harkey, et al. (1997). *Instant CORBA*, Wiley Computer Publishing, John Wiley & Sons Inc.

[14] Rodriguez-Tome, P., Helgesen, C. et al. (1997) *A CORBA Server for the Radiation Hybrid Database*, 5th Int. Conf. on Intelligent Systems in Molecular biology, Halkidiki, Greece.

[15] Vinoski, S. (1997). *CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments*, IEEE Communications Magazine, Vol. 14, No. 2.

[16] Vossen, G. (1997). *The CORBA Specification for Cooperation in Heterogeneous Information Systems.* 1st Workshop on Cooperative Information Systems; LNCS 1202, Kiel, Germany.

[17] Wallace, E. and K. C. Wallnau (1996). *A Situated Evaluation of the Object Management Group's (OMG) Object Management Architecture (OMA).* 11th Conf. on Object-Oriented Programming, Systems and Applications, San Jose, California.

[18] Wells, D. L. and C. W. Thompson (1994). *Evaluation of the Object Query Service Submissions to the Object Management Group.* IEEE Quarterly Bulletin on Data Engineering 17(4): 36 - 45.