Metadata Queries for Complex Database Systems

Gerald O'Connor

A dissertation submitted in partial fulfilment of the
requirements for the award of
Master of Science in Computer Applications

to the

Dublin City University

School of Computer Applications

Supervisor: Mark Roantree

17th September 2004

Declaration

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of Master of Science in Computer Applications is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed 

Student ID 51186128

Date 17th September 2004

Acknowledgments

I would like to thank my wife and family for their endless support and encouragement.

I would like to thank my supervisor, Mark Roantree, for giving me expert guidance and advice in the preparation of this thesis.

I would also like to thank all the members of ISG including Dalen, Damir and Ling.

I would like to mention the practice of Falun Dafa which gave me the patience and endurance to continue until the thesis is complete.

Abstract

Federated Database Management Systems (FDBS) are very complex. Component databases can be heterogeneous, autonomous and distributed, accounting for these different characteristics in building a FDBS is a difficult engineering problem. The Common Data Model (CDM) is what is used to represent the data in the FDBS. It must be semantically rich to correctly represent the data from diverse component databases which differ in structure, datamodel, semantics and content. In this research project we look at the complexity of the FDBS and examine which datamodel is most suited for the CDM. A good metadata interface and query language is essential for the CDM because merging component databases into the FDBS and maintaining and building the FDBS rely on a complete metadata interface and query language. In this research project we analyse the metadata interface and query language of the Object-Relational datamodel with a view to use it as the CDM. Distributed Component databases in a FDBS need to be merged in to the FDBS, current tools can not completely automate this process, we examine these problems and present a mobile solution.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Federated Database Management Systems

A Federated Database Management System (FDBS) is a collection of databases that co-operate together to share information. The databases can be distributed geographically or stored on the same machine. Each database maintains control of its data and decides to what degree its information is shared with the federation. The databases are heterogeneous; they differ in data-model, query language, semantics and how the physical data is stored. For a large federated database incorporating many local databases, it is important that each local database is well defined with metadata otherwise building the federation is a difficult and expensive task.

Each local database is autonomous, and maintains control over its information and decides to what degree it will share information with the federation. A local database can join or leave the FDBS at the local administrators discretion. A classification discussed in [30] includes three types of autonomy: *design, communication and execution.*

Design autonomy means the local administrator has control over how the local database is designed and structured. The administrator maintains control over:

1. the data being managed,
2. the representation and the naming of the data elements,
3. the semantic interpretation (meaning) of the data,
4. constraints (rules over the data),
5. the functionality of the system,
6. the implementation.

Communication autonomy means the local database decides how it communicates with the FDBS and how it manages local queries. It is the local database, which has authority to decide the priority of communications. Similarly, execution autonomy means the local database decides what order the queries are executed in, as the FDBS can not force the local database to execute queries in any order. This illustrates that in a FDBS the local database maintains control of its information but chooses to cooperate with the federation. Usually there is an agreement protocol between the FDBS administrator and the local administrator. In such a complex system with many heterogeneities that need to be overcome and where control remains with the local database it is important that the local database completely describes with metadata what it is offering to the FDBS and how it will treat queries from the FDBS.

1.1.1 Federated Architecture

In [27], a five-layer reference architecture is used to illustrate a broad range of federated databases (*see figure* 1.1). The component database system physically stores the data. The local schema stores the data-model for the component database, thus local database structure can differ.

The local schema is translated in to Common Data Model (CDM) or canonical model, which is stored in the component schema. The CDM needs to be rich enough semantically to accurately capture the meaning of all the local schemas. The CDM is the data-model for the FDBS. Each local schema must be mapped to the CDM. For this to be possible a clear definition of the local schema structure (metadata) with a standard query interface must be available. It is important that the local model is clearly defined with metadata so as not to lose information and to avoid unnecessary mining for data.

The export schema is a subset of the component schema that can be integrated into the federated schema. Export schemas allow for association autonomy in the FDBS, which means certain federated database users have access to a subset of the component schema while other users have access to a different subset. Each export schema needs to be integrated into the federated schema. The federated schema gives a global user the impression that he is querying a single information source. At the federated schema layer information regarding the heterogeneity and autonomy of local schemas is not present, instead the view of a unified information source is presented. Data distribution metadata is included in this layer of the architecture. There can be multiple federated schemas incorporating a different combination of export schemas, each for a different class of federated schema user.

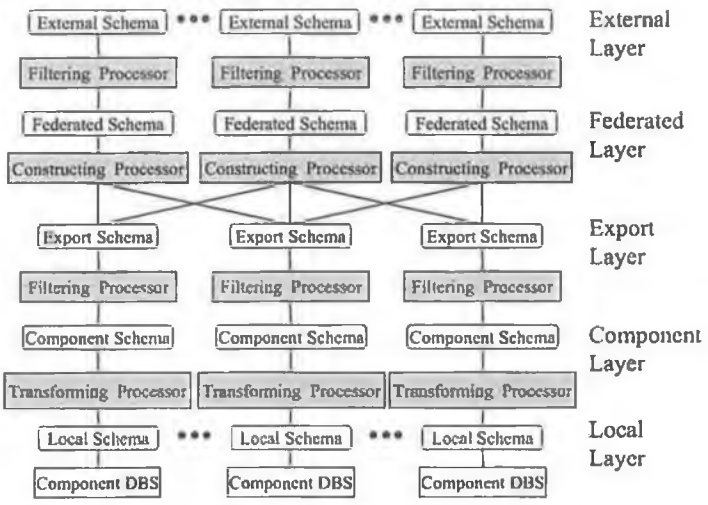

Figure 1.1: Five Layer Schema Architecture

The external schema is a subset of the federated schema. The federated schema can be very large and complex and therefore difficult and expensive to engineer, so the external layer offers a subset of it specific for a particular user. Extracting a subset of information from the federated layer can be made easier if the CDM is clearly defined with metadata.

1.1.2 Schema Integration in a Federated Database

Schema integration refers to the integration of multiple views in to a single schema. In [2], the schema integration process is divided into five steps:

1. pre-integration,
2. comparison,
3. conformation,
4. merging, and
5. restructuring.

Pre-integration involves preparing different schemas to be integrated; in the federated structure this means transforming the local data models to the CDM. The *comparison stage* starts when the local database schema have been transformed to the CDM and one can look for semantic and structural overlaps and similarities between them. The *conformation stage* involves verifying the assumptions made for structural and semantic overlaps between schemas detected in the comparison stage. The *merging and restructuring stages* involve

building the integrated schema while accounting for the confirmed correlations in the different schemas. In a federated database, having a well-defined metadata query interface is important throughout the process but especially during pre-integration and comparison stages. To translate a schema from one data-model to another it is necessary to be able to completely describe the original structure because if meaning is lost in the pre-integration stage the disparities in the data at higher levels of the FDBS structure will cause greater problems and lead to the misinterpretation of data. During the comparison stage if a schema is not described accurately with metadata, it is possible differences are not noticed and overlooked. The CDM must be very expressive to capture the semantics of local schemas but the expressiveness must be well documented with metadata, this will ensure no loss of information.

Each layer of the FDBS's structure offers many complex integration issues. In order to overcome the differences in semantics and structure at each level it is essential that each aspect of the local schema and the canonical model is defined extensively with metadata. In this work we define a complete query interface to the object-relational (O-R) metamodel and illustrate current problems and shortcomings with querying O-R metadata. This is done with the view to ease the difficult task of integration in a federated database structure.

1.2 Metadata Overview

Metadata is commonly understood as any information needed in information technology in order to analyse, design, build, implement and use computer systems. In the case of information systems, metadata particularly facilitates managing, querying, consistent use and understanding of data.

The notion of *meta* is related to modelling, when modelling complex information systems at least four layers of metadata are needed for the data to be well defined. Level 0 is the data (e.g. Library books); level 1 contains metadata (e.g., author, title, date published); level 2 specifies the schema used to store the metadata (e.g., the library cataloguing system) and level 3 contains a metamodel that unifies the different modelling languages specified on level 2 (e.g., a federated system for querying multiple library catalogues).

According to [28] the generation and management of metadata contributes to achieve the following tasks and objectives:

- **Improving interaction with the system.** For information systems it is important that a clear interface to the metadata is available to be queried and browsed to avoid

moving large objects over the network. This metadata interface should be clear and well defined to allow the user to query, access, and use metadata at the least cost.

- **Improving data quality.** In information systems data must be consistent, up to date, accurate and complete. The metadata should describe, who owns the data, when was it created, when was it last modified, who has access, what it means and so on.

- **Supporting the system integration process.** Integrating federated database systems is only possible if the structure of local database schemas and the meaning of the data they hold can be discerned.

- **Supporting system maintenance, analysis and design.** Metadata increases control and reliability of the database by providing information about the structure, meaning and origin of the data and by providing documentation of the existing structures that need to be extended.

1.2.1 Application Areas for Metadata

Metadata greatly assists in situations where data must be shared and reused. Computer systems general application areas include sharing, interpreting, storing and manipulating data, therefore metadata is needed across all areas of computer applications. In the following sub-sections the broad applications of metadata in computer systems are discussed in order to fully understand this important area of computing and how it relates to FDBS.

Software Engineering

Computer Aided Software Engineering (CASE) tools are a primary application of metadata in software engineering. Large computer applications, or systems, need to be thoroughly modelled in order to be developed and maintained and this modelling process is also a process of defining the metadata for future users to understand and reuse the computer system.

As software engineering technology becomes more powerful it is also becoming more complicated and difficult to reuse. For example the object-oriented (O-O) technology has increased the need for metadata as it is used to keep track of defined classes, methods instances and the interdependencies between them.

The concept of *reflection* [29] in software engineering implies that a piece of software comes with enough metadata to be self-describing and has access to this metadata in order to

use it at runtime. If a system not only uses the metadata but also manipulates it and thus has an open implementation, this is called meta-programming [17]. Architectures for distributed computing like Microsoft's (D) COM or OMG's CORBA (Common Object Request Broker Architecture)[24] use metadata, similar to above, to describe all available services and component interfaces in a distributed architecture. Thus independently developed components may dynamically discover each other collaborate at runtime. The key solution is the "Interface Repository" (metadata-repository), which allows this interaction.

Multimedia

The storage and retrieval of multimedia is of great importance to the database research community. The reason is that traditional search and retrieval techniques are no longer applicable for multimedia repositories. Exact-match query processing is not possible and content-based search is either not possible or too expensive (time consuming and resource consuming). So it is necessary to describe multimedia objects with metadata, which will improve load on the system because metadata is smaller in size than multimedia data for querying and retrieval.

Metadata for multimedia systems can be divided into three categories:

- domain-specific metadata,
- content-specific metadata,
- content-domain-independent metadata.

Domain-specific metadata is information that cannot be deduced from the picture, or sound clip, but which adds depth to what can be deduced from the media. For example, a recorded discussion which is complemented with information of context, location, speakers etc, adds more value to the audio file as a piece of information. *Content-specific metadata* can be deduced from the audio file; i.e. background noise, male or female speakers, tone of voice etc. Finally *content-domain-independent metadata* tells the size of a file, its location etc. Metadata plays a important role in the efficient storage and retrieval of multimedia data.

Information Management Systems

One of the first explicit uses of metadata was in the Database Management System (DBMS). The system catalogue and data dictionary stored information about the structure, constraints, physical storage information, access rights etc, for the information stored

in the database. There can also be information about the users, security and access privileges to the DBMS.

The problem is of greater magnitude with federated databases because of issues relating to heterogeneity, autonomy and distribution. Besides the information already mentioned for data dictionaries, FDBS need descriptive metadata, which contains information about the types and sources in the integrated system, and navigational metadata on how to handle a source and how a source is formatted [28] .

Descriptive information means that the integrated data sources content need to be described with metadata so that when queries are formulated at the federated layer the system will know where to look for relevant data with out wasting time and resources. Navigational metadata describes the mappings between the canonical model and the local model and how the local database can be manipulated to retrieve the desired information.

This discussion serves to illustrate the growing complexity of computer systems and information systems, and the need for these systems to be sufficiently described with metadata if they are to be reused and maintainable. It also illustrates the importance for tools to be developed in this area to manipulate metadata effectively.

1.2.2 Metadata in Object-Relational Databases

The Object Relational (O-R) database model is a complex structure combining features of relational and object models. The relational model (SQL-92 standard) consists of tables, triggers, constraints, views and procedures, while the object model consists of types, associations, aggregations, encapsulation, inheritance and other complex structures. This combination of features provides a powerful environment for representing data, but requires a very complex metamodel.

This is further complicated by the fact that structural information is often combined with physical storage information, e.g. where data is physically stored and metadata about how data is formatted and structured. There is also metadata for users and security, which make the metamodel complicated with many tables, and tables with many columns.

In the Interoperable Systems Group in DCU a research project [31] extended the O-R metamodel to include metadata for roles. Roles [9] address a shortcoming in conventional models that fail to completely model real world environments. Modern programming languages and databases can only partially model a real world entity for which they are defined as they lack the temporal aspect of real world entities. These entities characteristics (variables) can change but the underlying structure that defines what they are remains the

same. For instance, a person has an age, height and weight. A student has a school but a professional has a job. In the real world a person can become a student or a professional over a period of time, it is difficult to represent this transition in conventional database models. A role is viewed as an extension of an object that represents temporal aspects of real world objects.

The O-R model is a very expressive model and thus can be used for the canonical model in a FDBS [26]. A FDBS consists of many autonomous parts that need to cooperate together seamlessly in order to present the user with a unified information source, for this to be possible each part must be well defined with metadata and present a clearly defined metadata query interface. Yet due to the complexity of the O-R metadata defining a clear metadata interface is a difficult task.

1.3 Introduction to Mobile Computing

The Personal Digital Assistant (PDA) has become more powerful, less expensive and a wider range of applications are available, which makes it more attractive to a wider market and more useful for a broader range of user.

The main advantages of a PDA are: mobility, size and the fact that through networks they can access data at any time in any place. The main disadvantages are: limited memory size, limited processor speed, security risks, small screen size, limited battery power, low bandwidth, limited services and applications, and non-conventional input devices [6, 7].

As this technology matures, PDAs are becoming less of a novelty and becoming more of an essential tool in certain environments. For example, many applications are being developed for the medical profession. The clinical and administrative suites developed claim to automate the most labour intensive and time consuming aspects of medical treatment with easy to use applications at the point-of-care. A survey carried out in Mount Sinai Medical Centre New York [16] discovered, that half of the 88 physicians surveyed use PDAs, and they use them mostly for professional work. In the computer industry, IBM has developed applications for a PDA that allows a technician to configure UNIX servers using a simple plug and play interface on the PDA. This then saves the system administrator the trip to the server farm and he can complete final system configuration and launch applications from a remote console. These examples illustrate that PDA technology is maturing and its range of applications are broad.

The FDBS consists of many distributed databases that need to be integrated into the CDM. Currently there are no mobile tools to assist the integration specialist analysing

schemas on local distributed sites.

1.4 Motivation

The canonical model for a federation of databases needs to be semantically powerful in order to represent the local database schemas in the federation. Each local database can use a different data model to structure their data. The relational model is the most common model used for local databases however it is not powerful enough to represent multimedia documents well so it is not suitable for the canonical model of the FDBS. The object-oriented model supports complex constructs such as types, methods, inheritance, association etc which makes it suitable for representing any complex object including multimedia objects. Due to the differences in the relational and object-oriented models it is a difficult task to migrate relational data to the object-oriented model. The object-relational model combines the features of the relational and object-oriented models. It is semantically powerful and it provides a means to transform a schema represented in the relational model to the object-relational model. The object-relational model is examined in this research because it is suitable for the canonical model in a federation of databases.

The canonical model of a federation of databases needs to be completely represented with metadata. At every layer of the federation the metadata needs to be examined in order to transform a schema from one model to another , extract a subset of a schema (ie specialise a schema) and also for building query processes to accurately and quickly retrieve information from the federation for a user. If the canonical model is not well represented with metadata the data in the federation can be corrupted because the data's meaning is interpreted incorrectly. The object-relational metadata is complex because it is a combination of the relational and object oriented models. In current implementations of the object-relational model the metadata comprises formating information, security information, physical storage information and structural information. There are many views of the metadata many of which are big, cumbersome to use and require expertise in SQL. Also some structures that are needed to view and manipulate the metadata are not present. In this research project we analyse this problem with the intention of providing a complete metadata query interface for an engineer to access and view object-relational metadata.

In this research project an application to use the metadata query language and interface is provided on a mobile device. This is necessary because an integration specialist is faced with the task of integrating component databases that are distributed over a geographic

area. As far as we are aware no current research project has completely automated the process of integrating component databases in to the federation due to difficulties in correctly understanding the semantics of the data. Therefore it is necessary for the engineer to visit local database sites to consult database administrators about the meaning of the data in their databases. Providing a metadata interface and query browser on a mobile device is needed to assist the engineer while visiting local database sites to deduce the structure of a database while discussing semantic meaning with the local administrator.

Chapter 2

Related Research

2.1 The Clio Project

The Clio Project [21] offers analyses of the problems faced when managing and facilitating the complex tasks of heterogeneous data transformation and integration, and suggests solutions to these problems. Integration and transformation are discussed over three broad categories:

- Schema and Data Management,
- Correspondence Management,
- Mapping Management.

In [21] it is argued that at the core of all integration tasks lie the representation, understanding and manipulation of schemas and the data that they describe and structure. It is very important that the metadata is complete so as not to lead to inaccurate information and misrepresentation of the intended meaning of data. Since integration methodologies depend on the accuracy and completeness of structural and semantic information, they are best employed in an environment where specified schema information, constraints and relationships can be learnt, reasoned about and verified. This illustrates a strong argument for mobile tool to analyse schemas as it allows the integration specialist to visit the local sites of the database, manipulate the schema while verifying with the local administrator correspondences and hard to extract semantic information.

Correspondence management: This is the process where correspondences between data and metadata, in different schema can be related and matched; it is referred to as determining "inter-schema" relationships in [25], and in model management it is referred

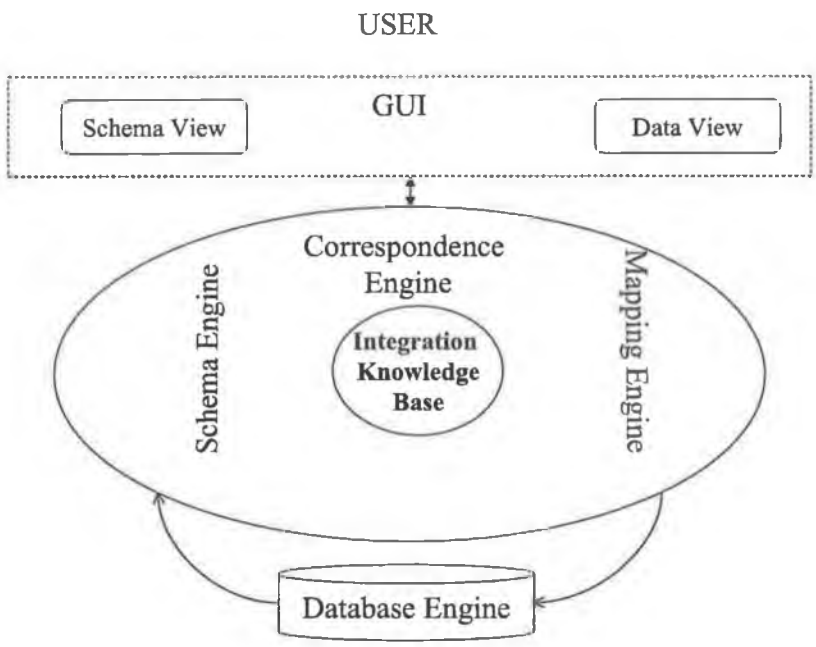

Figure 2.1: Clio's Logical Architecture

to as "model matching" [5]. In [21], it is explained that finding correspondences cannot be fully automated since the syntactic representation of schemas, metadata and data may not completely convey the semantics of different data sets. Whether the correspondence process is automated or manual it cannot always be accurate for all possible schemas, and it is important to counter in a process for verifying the correspondences, either manually or using a knowledge discovery technique. This is especially the case since some schemas can be very large and it may take a number of iterations to verify that the correspondences are correct.

Mapping management: Once correspondences have been derived it is necessary to deduce a set of mappings from the canonical model to the local models. The implementation and maintenance of this mapping is still largely a manual job and extremely complex.

In the Clio project a number of tools have been designed and implemented to make the task of integration and transformation easier taking into account the limitations mentioned above.

Fig 2.1 shows Clio's logical architecture. The schema engine in *Fig 2.1* is an application used to view and manipulate a given schema that has been loaded in to Clio's system. The idea is to provide a means to understand a schema via a graphical user interface (GUI) and manipulation tools.

Clio also provides a Correspondence Engine, which is a tool for generating and managing a set of candidate correspondences between two schemas. The generated correspondences can be augmented, changed or rejected by the using a graphical user interface through which users can draw value correspondences between attributes. In [21], it is argued that Clio could be augmented to make use of dictionaries, thesauri, and other matching techniques considerably enhancing its usefulness.

Finally, the Mapping Engine in the Clio project supports the creation, evolution and maintenance of mappings between pairs of schemas. The mapping engine uses information gathered from the Schema Engine and the Correspondence Engine. As with the previous two engines mappings are verified using a GUI and alternative mappings are suggested and can be manipulated. The usefulness of these tools are illustrated in the building of a data-warehouse.

Other research projects try to completely automate the process of schema transformation and integration. The benefits from completely automating this process are often out weighed by the overhead in preparing schemas to be integrated and post integration examination of the results. Clio has taken another approach and successfully automated parts of the integration process. Yet their work is different our research project as we provide our tool on a mobile device giving more flexibility and power to the integration specialist.

2.2 Comparison of Schema Matching Evaluations

Schema matching is the task of finding semantic correspondences between elements of two schemas [19] . There are a number of systems that have been developed recently to determine schema matches semi-automatically and [12] compares some of them (Cupid [19], LSD [13], Similarity Flooding [20], Automatch [4], Autoplex[3]) to clearly define the advantages and disadvantages of each. This task is made very difficult by the fact that the system evaluations for each respective schema-matcher was done using diverse methodologies, metrics and data, making it virtually impossible to apply them to a common test problem or benchmark in order to obtain a direct quantitative comparison.

[12] attempts to standardise the criteria for future schema-matching evaluations by discussing the major criteria influencing the effectiveness of a schema matching approach. To compare the evaluations, four areas are considered most important:

1. Input: What kind of input data has been used (schema information, data instances, dictionaries etc.)? The simpler the test problems and the more auxiliary information

that is used, the more likely the systems can achieve better effectiveness. However, the dependence of auxiliary information can lead to increased preparation effort.

2. Output: What information is included in the match result (mappings between attributes or whole tables, nodes or paths etc.)? The less information the systems provide as output, the lower the probability of making errors but the higher the post processing effort may be.

3. Quality Measures: What metrics have been chosen to quantify the accuracy and completeness of the match result? Because the evaluations usually use different metrics, it is necessary to understand their behaviour, i.e. how optimistic or pessimistic their quality estimation is.

4. Effort: How much savings of manual effort are obtained and how is it quantified? What kind of manual effort has been measured, for example, pre-match effort (training of learners, dictionary preparations etc.), and post match effort (correction and improvement of the match output)?

The main motivation to develop an automatic schema-matcher is to save in labour of manually matching the schemas. Of all the schema-matchers evaluated none completely automate the process of schema matching. Pre-match and post-match manual work still needs to be completed. Pre-match efforts include;

- training of the machine learning based matchers,
- configuration of the various parameters of the match algorithms e.g. setting different threshold and weight values,
- specification of auxiliary information, such as domain synonyms and constraints.

Post-match efforts include examining and confirming the results. Confirming positive matches and negative matches, examining the threshold of matches (all examined projects evaluate a match as between one and zero) are all post-match labour that needs to be completed. [12] argue that it is possible that the pre-match and post-match manual labour efforts can actually outweigh the benefits gained through trying to automate the process.

This work confirms that the local administrator of a schema or database needs to be consulted to confirm the results of the automated schema matching process. This means that the local administer will also have to learn to understand the schema matching process in order to confirm the results. None of the tools described in [12] for schema-matching

provide mobile solutions, which would allow the local administer and the engineer to work with familiar tools during the schema-matching process at the site of the local database. This research project suggests that the savings made in manual labour in automating the schema matching process are often lost in the pre-schema matching and post-schema matching activities. Our mobile metadata schema browser provides a tool to allow an engineer to examine the structure of a local schema aiding them in the schema matching process but does not try to automate the process.

2.3 SchemaSQL

SchemaSQL is a language for interoperability in relational multi-database systems [18]. As with federated database systems one of the fundamental requirements in a multi-database system is interoperability, which is the ability to uniformly share, interpret and manipulate information in component databases in a multi-database system. The heterogeneity problems in multi-database systems are similar to what is mentioned in the previous section and can be summarised as semantic issues (interpreting and cross relating information in different local databases), structure issues (e.g. heterogeneity in database schemas, datamodels and schema processing) and system issues.

The problem of interoperability among a number of component relational databases storing semantically similar information in structurally dissimilar ways is considered in [18]. They argue, that the requirements for interoperability fall beyond the capabilities of languages like SQL.

A number of key features for a language that supports interoperability are outlined which include:

1. The language must have an expressive power that is independent of the schema. For instance in most conventional relational languages, some queries (e.g. find all department names) expressible against the database `Univ-A` in *fig 2.2* are no longer expressible when the information is reorganised according to `Univ-B` without querying metadata repositories.

2. To promote interoperability, the language must permit the restructuring of one database to conform to the schema of another.

3. The language must be easy to use and yet sufficiently expressive.

Univ-A

salInfo

Category	dept	salFloor
Prof	CS	65,000
Assoc Prof	CS	50,000
Prof	CS	60,000
Assoc Prof	Math	55,000

Univ-B

salInfo

category	CS	Math
Prof	55,000	65,000
Assoc Prof	50,000	55,000

Univ-C

CS

Category	SalFloor
Prof	60,000
Assoc Prof	55,000

Math

Category	SalFloor
Prof	70,000
Assoc Prof	60,000

Univ-D

salInfo

Dept	Prof	Assoc Prof
CS	75,000	60,000
Math	60,000	45,000

Figure 2.2: SchemaSQL Example

4. The language must provide the full data manipulation and view definition capabilities and must be downward compatible with SQL syntax and semantics.

5. Finally, the language must admit effective and efficient implementation. In particular it must be possible to realise a non-intrusive implementation that would require minimal additions to the component RDMS.

The main contribution of SchemaSQL is that it provides a means to query data and metadata, which thus allows restructuring and it does this while maintaining the SQL syntax and backward compatibility with SQL. While providing restructuring capabilities SchemaSQL permits the declaration of query variables which can range over any of the following five sets: (i) names of databases in the federation, (ii) names of relations in the database, (iii) names of attributes in the schema relation, (iv) tuples in a given database in a relation, and (v) values corresponding to a given attribute in a relation.

The following definition is presented from SchemaSQL because it describes the terms used in formulating a SchemaSQL query that is necessary to understand the queries presented later. The concepts of range specifications, constant and variable identifiers are simultaneously defined by mutual recursion as follows:

1. Range specifications are one of the following five types of expressions, where `db`, `rel`, `attr`, are any constant or variable identifiers (defined in two below)

(a) The expression `->` denotes a range corresponding to the set of database names in the federation.

(b) The expression `db ->` the set of relation names in the database `db`.

(c) The expression `db -> rel` denotes the set of names of attributes in the scheme of a relation `rel` in database `db`.

(d) `db :: rel` denotes the set of tuples in the relation `rel` in the database `db`.

(e) `db :: rel.attr` denotes the set of values appearing in the column named `attr` in the relation `rel` in the database `db`.

2. A variable declaration is of the form `<range><var>` where `<range>` is one of the range specifications above and `var` is an identifier. An identifier `var` is said to be a variable if it is declared as a variable by an expression of the form `<range><var>` in the `from` clause. Variables declared over the ranges (a)-(e) are called dbname, rel-name, attr-name, tuple and domain variables respectively. Any identifier not so declared is a constant.

In *example 2.2* a federation of schemas of `univ-A`, `univ-B`, `univ-C` and `univ-D` illustrated in *fig 2.2,* it is necessary for the query language to be able to query data and metadata seamlessly because what is represented as data in one schema is represented as data in another schema.

Example 2.1 *Sample SchemaSQL Query (Relation name metadata)*

```
select RelC
from univ-C-> RelC, univ-C::RelC C, univ-D::salInfo D
where RelC = D.dept and C.category = 'Prof' and C.salFloor > D.Prof
```

Example 2.1 lists the departments in `univ-C` that pay a higher salary floor to their professors compared with the same department in `univ-D`, which illustrates querying of metadata and data across two schemas. The statement `univ-C-> RelC` queries all the names of relations (metadata) in `univ-C` and stores them in variable RelC. The second part of the from clause has the statement, `univ-C::RelC C`, which queries all the tuples of all the relations in `univ-C` and stores them in variable `C`. The new constructs introduced in SchemaSQL make it easier to query metadata. The where clause of the query is interesting because it illustrates the manipulation of metadata and data. `RelC` is a variable that stores metadata from `univ-C`, ie table names (departments), yet in `univ-D`, department names are data. The statement `RelC = D.dept`, is an instance of how SchemaSQL allows the manipulation of metadata and data.

Example 2.2 *Sample SchemaSchemaSQL Query (Column name metadata)*

```
select T.category, avg(T.D)
from univ-B :: salInfo-> D, univ-> D, univ-B :: salInfo T
where D <> 'category'
group by T.category
```

Example 2.2 computes the average salary floor of each category of employees over all departments in univ-B. To do this it is necessary to have access to the column names, which is metadata but also information regarding the department names. Without access to this metadata via the statement `univ-B::salInfo-> D`, which retrieves the column names "Category", "CS", and "Math", it would be necessary to directly query the system catalogue. It is also interesting to note using SchemaSQL more abstract queries can be made where the exact structure of the schema does not need to be known when the query is written and also it is possible that the structure of the schema can change (add a department (column) to `univ-B`) and the query will remain valid.

The query computes the average salary floor of each category of employees over all employees. This query illustrates horizontal aggregation and how to query the unknown schema structure using SchemaSQL.

The system architecture consists of a SchemaSQL server that communicates with the local databases in the federation and remote clients. It is assumed that the meta-information comprising of component database names, names of relations in each database, names of the attributes in each relation and possibly other information (statistical information for optimisation) are stored in the SchemaSQL server in the form of a relation called the federation System Table (FST).

Global SchemaSQL queries are submitted to the SchemaSQL server, which determines a series of local SQL queries and submits them to the local database. The SchemaSQL server then collects the answers from the local databases and using its own resident SQL engine, executes a final series of SQL queries to produce the answer to the global query.

SchemaSQL provides a means to manipulate data and metadata in a relational database, yet the relational model is less expressive than the object-relational model, it is argued in [26] that a more expressive model is needed for the canonical model in a federated database. A federated database is generally a distributed structure and due to the complexity of this structure integrating the local schemas can not be fully automated, while SchemaSQL provides a means to query over multiple schemas in a seamless way it provides no tools

to over come the need to visit local database sites and consult local administrators during the difficult task of merging schemas together.

2.4 Querying and Restructuring the Tabular Database Model

Tables are one of the most natural ways that data can be represented [15]. The relational model's structure however limits it to a variety of possible tables. Some tables have names for their columns (like relations) and rows (unlike relations), and these names need not be distinct (unlike in relations). Tables as opposed to relations offer symmetry between rows and columns, and the latitude that row and column names may occur multiply or may even be absent. [15] argues that exploiting this symmetry and flexibility allows for a much broader class of natural data representations than captured by the relational model.

Many applications can significantly benefit from the integration of database systems (whose strength is efficient and robust on-line processing (OLTP) and handling large volumes of data) with analytical tools like spread sheets (which offer on-line analytical processing (OLAP) capabilities). Spreadsheets model data in the form of tables (arguably more liberally than in the relational model) and have several powerful analytical tools built into them. Examples include row and column arithmetic, generalised aggregation on arbitrary blocks of values drawn from tables, and the ability to invoke external functions. It is pointed out in [15] and [14, 10] that an integration of relational database systems and spreadsheets will combine their complementary strengths in OLTP and OLAP respectively, leading to a powerful environment for data processing. [15] describes a powerful model and language that supports convenient restructuring of data between various tabular representations. They also argue that their work is the first fundamental querying and restructuring language proposed for OLAP systems.

In order to describe the tabular model, two types of symbols are distinguished: N (called names) and V called (variables). Names can be thought of as a generalisation of relation and attribute names. To allow a broad class of data representations, names are allowed to appear in positions that are normally thought of as data entry positions. Also variables are able to appear in the usual position for names. Tables need not have entries for every row and column combination. The null value $\perp$ (inapplicable) is used to signify the absence of entries. The set of all symbols S, is then $N \cup V \cup \perp$. A tabular database is a set of tables. If T is a table with row numbers $0...M$ and column numbers $0...N$ then table T is called a table of width M and height N. The width and height of T are denoted $width(T)$ and $height(T)$, respectively. For I a finite sequence over $0...M$, and J, a finite

(Table Name)T_0^0	(Column Attributes)$T_0^>$
(Column Attributes)$T_>^0$	(Data Entries)$T_>^>$

Table 2.1: Representation of a Table

sequence over $0...N$, T_I^J denotes the sub-table of T formed by rows and columns indicated. In particular, for $0 \leq i \leq height(T)$ and $0 \leq j \leq width(T)$, T_i denotes the i^{th} row, T^j denotes the j^{th} column and T_i^j the entry $T(i,j)$. The sequence $(i+1)...\,height(T)$ will be abbreviated to $> i$ and the sequence $(j+1)...\,width(T)$ will be abbreviated to $> j$ (The index position will disambiguate between the two possibilities).

Using the notation in a block diagram illustrated in *table 2.1*, four regions can be distinguished in a table T. The entry T_0^0 is the table name, the entries $T_0^>$ are called the column attributes, the entries $T_>^0$ are called the row attributes, and the entries $T_>^>$ are called data entries.

The tabular algebra consists of assignment statements of the form $T \leftarrow < operation >< parameterlist >< argumentlist >$ with T a table parameter, augmented with an iteration construct. Parameters can be considered as table names, column attributes or sets of column attributes. The argument list is a set of table name parameters. Each time an assignment statement as above is invoked; the operation is invoked on each sequence of tables in the database, whose names match with the table name parameters in the argument list. The resulting table is named T.

Four restructuring operations are implemented, which include, grouping, merging, splitting and collapsing. Grouping and merging are described below with the aim of illustrating schema-restructuring operations outlined in [15].

Grouping

The syntax of a grouping assignment statement is $T \leftarrow GROUP_{by\ A\ on\ B}(R)$ with A and B attribute set parameters. An example of grouping is $Sales \leftarrow GROUP_{by\ Region\ on\ Sold}\,(Sales)$ applied to Schema (A) in *fig 2.3*. The resulting table,

1. Its attribute row is obtained by first extracting from the row of the original table the attributes different from both `Region` and `Sold` (only `Part` in this example) and then adding this together with as many copies of `Sold` as there are data rows in the original table.

2. Next, the column headed by `Region` is added as the first data row of the new table.

Schema A

Sales	Part	Region	Sold
⊥	nuts	east	50
⊥	nuts	west	60
⊥	nuts	south	40
⊥	screws	west	50
⊥	screws	north	60
⊥	screws	south	50
⊥	bolts	east	70
⊥	bolts	north	40

Schema B

Sales	Part	Sold	Sold	Sold	Sold	Sold	Sold	Sold	Sold
Region	⊥	east	west	south	west	north	south	east	north
⊥	nuts	50	⊥	⊥	⊥	⊥	⊥	⊥	⊥
⊥	nuts	⊥	60	⊥	⊥	⊥	⊥	⊥	⊥
⊥	nuts	⊥	⊥	40	⊥	⊥	⊥	⊥	⊥
⊥	screws	⊥	⊥	⊥	50	⊥	⊥	⊥	⊥
⊥	screws	⊥	⊥	⊥	⊥	60	⊥	⊥	⊥
⊥	screws	⊥	⊥	⊥	⊥	⊥	50	⊥	⊥
⊥	bolts	⊥	⊥	⊥	⊥	⊥	⊥	70	⊥
⊥	bolts	⊥	⊥	⊥	⊥	⊥	⊥	⊥	40

Figure 2.3: Restructuring with Grouping

3. Finally the data rows from table (A), after projecting out the region entries, are added to table (B), as follows, consider row i in table (A). The `Sold` entry of this row is added under the i^{th} occurrence of the `Sold` column in table (B), on row i. The remaining entries of row i in table (B) are filled with $\perp$ (non-applicable)

The grouping example illustrates how OLAP functionality can be used with the tabular database model. This is not possible in the relational model using SQL because the tabular model allows column attributes to have the same name where as the relational model does not and also in the tabular model due to the fact that traditional metadata positions (like column names) can be interchanged with data it allows for the grouping operation to take place.

Merging

The syntax of a merging assignment statement is $T \leftarrow MERGE_{by\ A\ on\ B}\ (R)$, with A and B attribute set parameters. Applying the merging assignment $Sales \leftarrow Merge_{on\ Sold\ by\ Region}$ $(Sales)$ on `table A` in *fig 2.4*. The resulting table, `table B` *fig 2.4*, is obtained by reversing the steps in the grouping operation. `Table A` in *fig 2.4* illustrates the feature of the tabular model which allows rows to have attribute names, ie (the row attribute names `Total` and `Region`). The merging example is also an example of OLAP operations in the tabular model that are not possible in the relational model with SQL.

In [15], a tabular model is illustrated with powerful schema-restructuring capabilities. It illustrates the need and usefulness of querying over data and metadata, which is currently

Schema A

Sales	Part	Sold	Sold	Sold	Sold	Sold
Region	⊥	east	west	north	south	Total
⊥	nuts	50	60	⊥	40	150
⊥	screws	⊥	50	60	50	160
⊥	bolts	70	⊥	40	⊥	110
Total	⊥	120	110	100	90	420

Schema B

Sales	Part	Region	Sold
⊥	nuts	east	50
⊥	nuts	west	60
⊥	nuts	north	⊥
⊥	nuts	south	40
⊥	screws	east	⊥
⊥	screws	west	50
⊥	screws	north	60
⊥	screws	south	50
⊥	bolts	east	70
⊥	bolts	west	⊥
⊥	bolts	north	40
⊥	bolts	south	⊥

Figure 2.4: Restructuring with Merging

not provided in O-R database systems. It is a simple yet expressive model with a wide number of applications. Yet, because of its simplicity the difficulties that arise from deducing the structure of a more complex schema do not arise here. The model is more expressive than the relational model but because it does not support complex structures like behaviour or inheritance it is not suitable for the canonical model in the FDBS which needs to be more expressive than any of the component models (which might include object model).

2.5 Noodle: A Language for Data and Metadata Querying in an Object-Oriented Database

In [22], an object-oriented query language for querying over schema and data is discussed. There are several novel features of the *Noodle* system including:

1. Query variables can range over all classes, relations attributes and objects.
2. Queries can do implicit schema querying. Queries such as "*find all classes (or find all subclasses of vehicle class) whose objects have an attribute of engine capacity*" can be expressed without explicitly referencing the system catalogue (see *example 2.3*).

A feature of the *Noodle* system is the ease of expressing queries that would require schema querying in other systems. Consider the case where `MotorVehicles` is a subclass of a class representing vehicles with an attribute `EngineCapacity`. Class `Vehicle` also has subclasses some of which have attribute `EngineCapacity` others do not. Let `FordFastVehicle` be the (view) collection of the vehicles that have an engine capacity greater that 100 and are manufactured by ford. *Example 2.3*, computes the view regardless of how many subclasses of vehicle have attribute `EngineCapacity` and illustrates *Noodles* power to query over a schema. The atom `V[EngineCapacity = E, Manufacturer = M]` is false for all vehicles that do not have the attribute `EngineCapacity` and `Manufacturer` and succeeds for those that do, illustrating how *Noodle* can be used to query over metadata. Also note that the view `FordFastVehicle` does not have to change in regards to changes to the schema (structural changes).

Example 2.3 Example of Explicit Schema Querying in Noodle

```
FordFastVehicle {Vehicle = V} :-Vehicle{Member = V} & V[EngineCapacity
= E, Manufacturer = M] & E > 100 & M[Name = 'ford']
```

Example 2.3 is an example of implicitly querying metadata as it checks whether any sub-class of `Vehicle` has the attribute named `EngineCapacity` and `Manufacturer` before trying to retrieve the values of these attributes and check if the `where` clause is true. *Noodle* also provides a means of explicitly querying metadata via the system catalogue, yet similar to current object-relational databases the system catalogue includes information on security and versioning which irrelevant to the integration specialist and makes the task of integrating and merging the schema in to the federation more complicated. Data and metadata queries in the *Noodle* system are non-standard and are not compatible with SQL. Noodle is an object oriented language therefore it is more difficult to merge the common relational database in to a federation than if we use the O-R model

2.6 Context Aware Mobile Computing

Context-aware computing is a mobile computing paradigm in which applications can discover and take advantage of contextual information (such as user location, time of day, nearby devices and user activity). The technology that has allowed context aware mobile computing to emerge is improvements in mobile computers and the improvement in the bandwidth of wireless networks.

Current work in context-aware mobile computing is largely focused on the context of location of a mobile device and offers services based on this variable [8]. For example [1] describe a system that provides information services to a tourist about her current location, she can find directions and retrieve background information about her current position. In [11] they also offer information based on location. In this system the assistant examines the conference schedule, topics of presentations, users location, and users research interests to suggest presentations to attend. Whenever the user enters a presentation room, the Conference Assistant automatically displays the name of the presenter, the title of the presentation, and other related information. These services are based on a central client-server approach.

The projects that are currently developing context-aware mobile applications are specialised and the servers are dedicated to providing information regarding a particular topic of interest. These client server applications are limited in scope because the broader range of applications they want to cover, the larger and more complex the central server and database must become. It is more suitable for databases on local networks, that the mobile user has access to, to store general information about its local environment. If databases that are on local networks have a complete structural metadata interface to their data and this information is made available then mobile applications can be written to access this metadata interface and thus the local data. In our work we provide a metadata interface and query language for an object-relational database which can be queried on a mobile device. The metadata interface and query language can be used as middleware allowing a roaming mobile user to access databases and query their contents which could include information about the local environment.

2.7 Summary

Recent work in schema-matching techniques illustrates that this part of the integration process cannot be fully automated and it has also been argued that the manual work of setting up a schema-matching application and examining the results actually outweigh the advantages gained in automating the manual task. In the Clio project a number of tools are presented which assist the integration engineer with the integration process. Instead of trying to automate the whole process, they have researched particular parts of it and provide a set of generic tools. Yet for the particular problem of integrating schemas in a federated database system, the issue of the distribution of local schemas is ignored. To successfully integrate a number of distributed local schemas the integration specialist will need to visit local sites and consult the local administer and CLIOs tools do not encompass

this problem. We address these shortcomings by providing a Mobile Metadata Schema Browser (MMSB), which allows the integration engineer to examine local schema on site while discussing semantic details with the local database administrator.

A number of schema query languages where analysed which provide metadata manipulation techniques but it was discovered that the relational and tabular models where not as expressive as the object-relational model, and that the integration specialist needed a clear interface to structural metadata which is not provided in implementations of the object-oriented model. The mobile metadata schema browser specifically browses object-relational metadata because this model is suitable to be the canonical model for the FDBS as it is very expressive. We provide a metadata interface that is useful to the integration specialist when examining a schema during the integration process.

Finally research work on context aware mobile-computing was looked at, and it was illustrated that a complete metadata interface to a database that is made available to mobile device is useful so the device can discover locally stored context information.

Chapter 3

The Object-Relational Metamodel

The Object-Relational (O-R) model provides object extensions to the relational model. Most commercial databases in use today adopt the relational model but the object-oriented model is more expressive and more suitable for storing complex data[26]. In [26] they discuss the suitability of a datamodel for the CDM of a FDBS by comparing their expressiveness and semantic relativism. They judge the relational model as not satisfying the requirements for the CDM but the object oriented model as satisfying all essential characteristics of the CDM. It takes time to migrate to a new data model so a hybrid data model can be more suitable for the canonical model of a FDBS . The canonical model of a federation needs to be very expressive and this is true for the O-R model but due to its power in representing complex structures its metadata is also very complex.

The relational model (SQL-92 Standard) supports tables, constraints, triggers, nested tables, views and procedures. An object-oriented model supports inheritance, classes, behaviour, aggregation, association and polymorphism. Combining the relational and object-oriented model provide a means for the storage and manipulation of complex data structures.

Most of the latest versions of relational databases, such as Oracle, Sybase and Informix extend the relational model with new constructs to support objects. In general, these databases have appeared in the market before the object-relational standard was published. Hence the current versions of O-R databases do not fully support the SQL:99 standard. In our research the Oracle 9i database model is treated as a standard as it supports most of the SQL:99 specification. In this chapter the O-R metamodel , its limitations and the difficultly that currently faces engineers when deducing the structure of an O-R schema will be discussed.

3.1 Object-Relational Metadata

Object-relational metadata can be viewed through a collection of virtual tables. These virtual tables are views of the Oracle metabase. In the view system, metadata is not necessarily unique to a certain virtual table and may be viewed in different ways. Virtual tables are specialised toward a particular users needs.The following are the headings under which metadata for O-R data will be discussed:

1. Types
2. Tables
3. Attributes of Types
4. Columns of Tables
5. Inheritance
6. Behaviour
7. Views
8. Association
9. Constraints
10. Cardinality of Relationships

3.1.1 Types

Existing implementations (Oracle 9i, Sybase and Informix) of the O-R model are effectively relational databases that have been extended to give users the impression they are manipulating objects instead of relational tables. The O-R model supports all object-oriented structures, yet the manner in which these structures are created, stored and manipulated is made more powerful because they are built on and can use relational structures.

O-R types are user defined data types that make it possible to model complex real world entities such as a client or an order as unitary entities (called objects) in the database. New object types can be composed of any built in database type and any previously created object types, object references, and collection types. O-R types include the behaviour in a structure called a method. For example, if you have an object called `Customer`, a method called `make_purchase` would change the internal structure of the object `Customer`, an attribute called `cash` will be reduced and attribute called `purchased_items` will be

incremented to include the new item. In relational database structures it is possible to include a form of behaviour. Stored procedures and functions can be used to manipulate relational data but they are separate entities to the data and are not encapsulated by a type. On the other hand it is possible to fetch, retrieve and manipulate a set of related objects and methods as a unified entity in an O-R database because they are linked together as a instance of a type.

Inheritance allows an engineer to create type hierarchies by defining successive levels of increasingly specialised subtypes that derive from a from a common ancestor object type. Derived sub types contain ("inherit") the structure of the super type (ie methods and attributes etc) and are permitted to extend the structure.

The metadata view that allows access to the information that describes types is called the `ALL_TYPES` view. This gives the general structure of the types and points the user where to look for more information. It contains fourteen columns, which include:

- `TYPE_NAME` and `SUPERTYPE_NAME` give the name of the type and its supertype (if a super type is part of the type specification).
- `OWNER` and `SUPERTYPE_OWNER` are metadata which show the creator of the type and the super type.
- `ATTRIBUTES` and `METHODS` are metadata that show the amount of attributes and methods in the type (inherited attributes and methods are included in this number).
- `LOCAL_METHODS` and `LOCAL_ATTRIBUTES` are metadata that describe the number of attributes and methods in the type excluding inherited attributes and methods.
- The `INSTANTIABLE` metadata indicates whether or not it is not possible to create an instance of this type. It is possible to have methods and types are not instantiable but still complete and valid. They may be used as the root in an inheritance tree where subtypes are instantiable.
- `FINAL` is a boolean value that indicates in an inheritance hierarchy if this types structure can be inherited. If it is true no type can inherit this types structure.
- The `INCOMPLETE` metadata indicates that a type is incomplete. For example, an O-R type references a type that does not exist.
- `TYPE_OID`, `TYPECODE` and `TYPEID` are implementation details for manipulating O-R types.

The metadata for types is complex and includes metadata that a vendor would use to efficiently manipulate types. Also some metadata is repeated without any obvious benefits. For example it is possible to tell if a type is inherited from the `SUPERTYPE_NAME` column, the extra information `ATTRIBUTES` and `METHODS` which includes the number of inherited attributes and methods is of little tangible use. If the user needs to discern the structure of the super type he can retrieve it using `SUPERTYPE_NAME`.

3.1.2 Tables

Tables are used to store data in the relational model and this structure is extended to be capable of storing instances of objects in the O-R model. The structures that are related to tables include constraints, triggers and views. In the O-R model tables are used for storing instances of objects and the manner in which they can be manipulated is extended beyond the capabilities of the standard relation model. In an object table each row represents an object. There are two ways to manipulate objects in an object-table:

- As a single-column table where each column is an object. This allows a user to perform object-oriented operations.
- As a multi-column table where each attribute of the object type is a column. This allows the user to perform relational operations.

Object tables support triggers and constraints in much the same way as relational tables. There are two exceptions; constraints can be implemented on leaf level scalar attributes of a column object, with the exception of `ref`'s that are not scoped and triggers cannot be defined on the storage table for a nested table column or attribute.

The O-R model supports two collection types: nested tables and varrays. A nested table is an unordered set of data elements all of the same data type. It has a single column and the type of that column is a built in type or a user defined type. If the type is user defined, the table can also be viewed as a multi-column table, with a column for each attribute of the object type. A varray is an ordered set of data elements of one data type. The size of the varray is fixed and must be set when the type is defined.

There are three metadata views in the O-R model to view tables: `ALL_ALL_TABLES`, `ALL_TABLES` and `ALL_OBJECT_TABLES`. The `ALL_TABLES` view allows a user to view metadata for relational tables. It has 43 columns that describe physical database storage metadata, user access metadata, formatting metadata and general statistics. The `ALL_OBJECT_TABLES` is a view of all the object tables in the database. It includes

information on the name of the table (TABLE_NAME), the type of object that will be stored in the table (TABLE_TYPE), the owner of the table and type stored in the table (TABLE_OWNER and TABLE_TYPE_OWNER), whether of not it is a nested table (NESTED), and physical database storage information, user access information, formatting information and general statistics. The ALL_ALL_TABLE is a view which includes information on all the relational and object tables in the database. It views all the previously mentioned metadata.

These tables have many columns of metadata. Much of the metadata stored in these tables is vendor specific and relates to how the vendor allows an administrator to manage the database.

3.1.3 Attributes of Types

Attributes hold data about an objects features of interest. For example, object type Person has attributes called name, address and date of birth. An attribute has a declared data type which is another object type, a built in data type (such as NUMBER, VARCHAR2 or REF etc), or a collection. Taken together, the attributes of an objects instance contain that objects data and all an objects attributes taken together at any time describe the state of the object.

An attribute can be a collection which is a VARRAY or a NESTED_TABLE. A VARRAY is an ordered collection of elements, at the time of creation you have to specify the length of the VARRAY. A NESTED_TABLE is an unordered list of elements. It can have any number of elements and no maximum is specified at creation time.

An attribute can reference another object. A similar idea in a relational database is a foreign key. The object that is referenced is a stand alone object and can be manipulated outside of referencing object. It is also possible to manipulate the attribute of type REF in the same way as any other attribute.

The metadata for attributes is taken from the ALL_TYPE_ATTR's view. The view has eleven columns which include:

- The TYPE_NAME is the name of the type that owns the attribute.
- The ATTR_NAME is the name of the attribute.
- The ATTR_TYPE_OWNER is the owner of the type of the attribute.
- ATTR_TYPE_NAME is the name of the type of the attribute.

- `ATTR_NO` is the order number of the attribute when the type was created.
- `INHERITED` is a boolean value that illustrates whether or not the attribute is inherited.
- `ATTR_TYPE_MOD` illustrates whether or not this attribute is a reference to another object.
- `LENGTH`, `PRECISION`, `SCALE` and `CHARACTER_SET_NAME` are formating information.

The metadata for attributes includes formatting metadata and structural metadata. Although attributes can be collections there is no reference to this type of metadata in this view.

3.1.4 Columns of Tables

Attributes and columns are discussed in different sections because the metadata relating to each are different. An attribute is part of a type, it can be a reference, a user defined type, a system defined type or a collection. A column of a table stores data. Constraints and triggers can be placed on a column but they can not be directly associated with a type but instead the data stored in the column.

In the O-R model tables are used to store the data and many of the constructs associated with relational tables can be applied in much the same way to O-R tables. Tables consist of one or more columns were each column stores a particular type of information. In the O-R model instances of objects are stored in tables in two ways; the object can be stored in a single column and manipulated like an object as a complete entity, or the table that holds the object can be treated as a multi-column table, were each attribute is stored in a column. The multi-column table approach to viewing an object allows a user to manipulate the instance data in the same way as columns in a relational table.

Columns are the initial building block when storing anything in an O-R database, for this reason there are many views of the metadata for columns. These views include:

- `ALL_COL_COMMENTS` stores comments relating to a column in a table.
- `ALL_COL_PRIVS` describes the privileges to a column in a table.
- `ALL_CONS_COLUMNS` stores metadata describing the constraints on a column.

- ALL_PUBLISHED_COLUMNS stores metadata on whether a column is published. Not all columns are published and may be used by the system hidden from the user.
- ALL_TAB_COL_STATISTICS stores statistics for a column.
- and ALL_TRIGGER_COLS store metadata relating to the triggers that depend on columns.

The general view for columns is ALL_TAB_COLUMNS. It contains thirty columns of metadata describing the owner of the column, the data type of the column, the table the column belongs to, whether the columns can be null or not and also access information, formatting information, statistics and physical storage information.

If an engineer is looking for metadata about columns to rebuild a schema it can be difficult to decipher it from the views presented. In tables that span thirty columns very brief explanations are given for each field in the available documentation which makes the job of finding relevant metadata to an engineers current task more difficult.

3.1.5 Inheritance

Inheritance in the object-oriented model allows an engineer to create type hierarchies by defining successive levels of increasingly specialised subtypes that derive from a from a common ancestor object type. There are three metadata views to view inheritance information in the O-R model;

- ALL_TYPES,
- ALL_TYPE_ATTRS,
- and ALL_TYPE_METHODS.

The ALL_TYPES meta-view contains metadata about whether or not this type inherits data from another type in the SUPERTYPE_NAME and SUPERTYPE_OWNER field. From the fields LOCAL_METHODS, LOCAL_ATTRIBUTES, ATTRIBUTES and METHODS it can be distinguished how many attributes and methods are local and how many have been inherited. The ALL_TYPE_ATTRS and ALL_TYPE_METHODS views both contain a field called INHERITED that tells a user if this attribute or method was inherited from a super type.

3.1.6 Behaviour

Behaviour is represented in an O-R database in two ways: triggers and methods. Triggers are defined on object-tables (tables that hold instances of objects) in the same way as they can be defined for relational tables but a trigger cannot be defined on a storage table for a nested table column or attribute. Triggers can also be defined on a database or a schema in the database and also on a view.

Metadata for triggers can be viewed from the `ALL_TRIGGERS` view. It has fourteen columns of metadata that describe the varies aspects of a trigger. The metadata includes;

- Metadata on the name of the trigger and owner (`OWNER, TRIGGER_NAME`).
- The type of the trigger (`TRIGGER_TYPE`).
- The triggering event (`TRIGGERING_EVENT`).
- The object on which the trigger is defined (`BASE_OBJECT_TYPE`).
- The table owner, table name and column name (`TABLE_OWNER, TABLE_NAME, COLUMN_NAME`).
- The cause for the trigger to be fired (`WHEN_CLAUSE`).
- Whether the trigger is enabled or not (`STATUS`).
- A description of the trigger (`DESCRIPTION`).
- Its action type (`ACTION_TYPE`).
- The statements executed by the trigger when it fires (`TRIGGER_BODY`).

Behaviour on types is implemented as methods. Methods are functions or procedures that a user can declare in an object type definition to implement behaviour that a user wants objects of that type to perform. They are how type attributes (data) can be accessed and manipulated at runtime. The signature for a method consists of method name, method type, parameters and the results. The method type could be member, static or constructor. Every object has a constructor that is implicitly created by the system; it can also be created by the engineer. It is a function that returns a new instance of a type and sets up the values of its attributes. A member method is a function that manipulates the attributes of a type and a static method is invoked on a type but does not manipulate the attributes. The parameters that are passed to the method and the results that are returned by a method can be user defined types, built in types, a collection or a reference.

There are three metadata views for accessing method metadata, ALL_TYPE_METHODS, ALL_METHOD_PARAMS and ALL_METHOD_RESULTS. ALL_TYPE_METHODS has eleven columns of metadata;

- The owner of the type and type name (OWNER, TYPE_NAME).
- the method name, method number (order of methods of the type) and method type (METHOD_NAME, METHOD_NO, METHOD_TYPE).
- the number of parameters and results (PARAMETERS, RESULTS).
- inheritance metadata (FINAL, OVERRIDING, INHERITED).
- Whether the method is instantiable (INSTANTIABLE).

The ALL_METHOD_PARAMS meta-view consists of OWNER, TYPE_NAME, METHOD_NAME, METHOD_NO and METHOD_TYPE which are the same as the ALL_METHODS view. It also includes;

- parameter name, number and modifier (if its a REF) (PARAM_NAME, PARAM_NO and PARAM_TYPE_MOD).
- The type of the parameter (PARAM_TYPE).
- The owner of the type (PARAM_TYPE_OWNER).
- Formatting metadata (CHARACTER_SET_NAME).

The ALL_METHOD_RESULTS meta-view consists of OWNER, TYPE_NAME, TYPE_OWNER, and METHOD_NAME which are the same as the previous two views. It also includes;

- The type of the result (RESULT_TYPE).
- The owner of the type (RESULT_TYPE_OWNER).
- Whether or not the result is a reference (RESULT_TYPE_MOD).
- Formatting information (CHARACTER_SET_NAME).

The O-R model provides three views of the metadata for methods. In most instances when engineers are examining a method they must examine all three views. When the metadata is presented in a system of views this leads to the user viewing repeated information or needing to use SQL to manipulate it.

3.1.7 Views

A view in a relational database allows a user to only see part of schema. For example a view on an employee may hide sensitive salary information including information of address, name and phone number which could then be made available safely to third parties without modifying the original storage structure. In an O-R database object instances are stored in object-tables and it is possible for the user to create *relational* views of the object-tables in the same manner as on relational tables.

The O-R model has extended the idea of views to include *object* views. A relational view is a virtual table and an object view is a virtual object. Each row in the view is an object, it has attributes and methods, and it is possible to create a reference that points to it. Object views can be created from columns in relational tables or object tables. This is a useful feature for a database engineer migrating a database to the O-R model from the relational model.

The metadata for views can be retrieved from the meta-view `ALL_VIEWS`. It consists of;

- The name of the view and its owner (`OWNER, VIEW_NAME`).
- The text length and the text for the relational query (`TEXT, TEXT_LENGTH`).
- The text length and query for the typed view (`TYPE_TEXT` and `TYPE_TEXT_LENGTH`).
- The object identifier length (`OID_TEXT_LENGTH`).
- The text for making the object identifier (`OID_TEXT`).
- The type of the view and its owner (`VIEW_TYPE, VIEW_TYPE_OWNER`).
- Inheritance metadata (`SUPERVIEW_NAME`).

The `ALL_VIEWS` view combines metadata for object-views and relational views. These are very different structures and are suitable to be used in different circumstances. Combining the metadata for both types of views leads to confusion to users not familiar with the O-R model.

3.1.8 Association

Association is achieved in the O-R model using the system defined type `REF`. A `REF` is a logical pointer to a row object. An attribute for a type can be of type `REF` which means that the object it references can be accessed or manipulated by using the `REF`. The object

that is referenced also exists independently and can be accessed or modified in its own right. Parameters and results for methods can also be REF's. If an attribute, parameter or result references another object a metadata column called modifier will hold the value REF.

3.1.9 Constraints

Constraints are a relational feature that can be implemented on tables or object-tables. There are five types of constraints, primary key constraint, referential constraint, check constraint on table (depending on a search condition), unique key constraint, and the read only constraint on a view.

The view for accessing constraint metadata is ALL_CONSTRAINTS. It consists of;

- The owner of the constraint (OWNER).
- The type of the constraint and the name of the constraint (CONSTRAINT_NAME, CONSTRAINT_TYPE)
- The table or view on which the constraint is defined (TABLE_NAME).
- The search condition (SEARCH_CONDITION).
- Owner of table referred to in a referential constraint, name of unique constraint definition for referenced table, and the delete rule for a referential constraint (R_OWNER, R_CONSTRAINT_NAME, DELETE_RULE).
- Enforcement status of the constraint (STATUS).
- Whether it is deferrable and whether it is initially deferred (DEFERRABLE, DEFERRED).
- Whether all data obeys the constraint (VALIDATED).
- Whether the constraint is user or system generated (GENERATED).
- If it is a bad constraint (badly formed logic) (BAD).
- Whether the enabled constraint is enforced or not (RELY).
- The date it was last modified (LAST_CHANGE).
- Information about index's (INDEX_NAME, INDEX_OWNER).

3.1.10 Cardinality of Relationships

The cardinality of a relationship is a definition of numeric relationships between occurrences of entities on either end of a relationship line. Viewing a schema's cardinality relationships can give an engineer an overview of how the schema relates together and how complex it is. Metadata for cardinality relationships are not supported in the O-R model. Metadata for cardinality relationships can not be easily mined using SQL and a procedural programming language is needed to manipulate the metadata to create these relationships. This will be examined further in the next chapter.

The cardinality of a relationship can be *one : one, one : many*, and *many : many* etc. The many side can (but does not have to be) a collection. There are two collection types in the O-R model; nested tables and varrays. A varray is an ordered set of elements; each element has an index number and this is used to access thcolectione elements. A nested table however, can have any number of elements and the ordering of the elements is not preserved. Both structures only hold a collection of one data type.

The main meta-view for viewing collection metadata is `ALL_COLL_TYPES`, which is a table of eleven columns describing nested tables and varrays. It includes the following metadata;

- Both nested tables and varrays are defined as types, so they include metadata type_name (collection name) and owner (`TYPE_NAME, OWNER`).
- The type of the collection (varray or nested table) (`COLL_TYPE`).
- Whether it is a collection of references (`ELEM_TYPE_MOD`).
- The upper bound (if it is a varray) (`UPPER_BOUND`).
- The name of the type in the collection (`ELEM_TYPE_NAME`).
- The owner of the type in the collection (`ELEM_TYPE_OWNER`).
- The precision and scale if its a number in the collection and the length if its a string (`LENGTH,SCALE, PRECISION`).
- Formatting and storage information (`CHARACTER_SET_NAME, ELEM_STORAGE, NULLS_STORED`).

Separate views are available in the O-R metamodel for accessing varrays and nested tables but they do not give all the necessary metadata. For example the meta-view

`ALL_VARRAYS` does not include the number for the maximum number of elements in the varray which is an important part of its structure. Combining the metadata for varrays and nested tables can lead to confusion to an engineer without in depth knowledge of the O-R structure as they will not be able to discern which column of metadata relates to which structure.

3.2 Summary

The O-R metamodel is a complex structure. The way the metamodel is currently presented combing structural metadata, formating metadata and storage metadata does not help the engineer to easily understand the underlying schema. The cardinality of relationships are not included in the metamodel and can not be easily mined using SQL. These relationships are important because they give an engineer an overview of the complexity of the schema and how the structure fits together. Finally the system of views that are used to present the metadata is not always intuitive; for example two different types of collection structure are combined in to the one view which can lead to confusion about which columns are related to which collection type. In the next chapter a simple O-R metadata query language is presented to address these issues. Its purpose is to provide an engineer with a simple and intuitive way of viewing and navigating O-R metadata.

Chapter 4

Metadata Query Language

Federated databases are very complex. They can be autonomous, heterogeneous and distributed. Each component database in the federation can have a different datamodel, different query language, different structure and vocabulary to describe their respective data. The common data model of the federation must be able to completely encompass all the component databases so their data can be correctly represented in the FDBS. This means the common data model must be semantically very powerful and expressive.

The Object-Relational (O-R) model is a powerful data model which is suitable for modelling complex data structures. Where as the relational database support tables, constraints, triggers, nested tables, views and procedures, the object-oriented model supports inheritance, classes, behaviour, aggregation, association and polymorphism. Combining the relational and object-oriented model provide a means for the storage and manipulation of complex data structures. The O-R model also provides structures which ease the task of migrating data from relational model to the object-relational model. As the need for storing more complex data increases this aspect accommodates relational database administrators in migrating their data to a more expressive model.

The O-R model is very rich and powerful and because of this the metamodel is also very complex. It is viewed through many virtual tables which can in turn have tens of columns of data. Federated database engineers are faced with a difficult task when querying the metadata as they need both expertise in SQL and the O-R model. For federated database engineers it is important to be able to deduce the high level structure of a database schema so they can quickly deduce where a component schema may fit in to the federation. In this chapter these issues will be addressed as we present our metadata query language which was designed in this research project. The query language was designed to be used on a PDA, so as to accommodate the federated database engineer who may need to visit

distributed component sites. The language is simple enough to be used through a PDA but powerful enough to retrieve all desired metadata.

This chapter is structured as follows: section one introduces the issues with querying the current O-R metamodel. Shortcomings in the metamodel are analysed and problems with querying the model through SQL are discussed. Section two illustrates through a series of examples how our metadata query language addresses these problems. Section three is a summary.

4.1 Accessing O-R Metadata with SQL

Complex SQL statements are sometimes needed to mine metadata from the O-R metabase. Other metadata are represented but good knowledge of the metamodel is necessary in order to find the necessary metadata. In this research, a clear metadata interface and query language is provided to ease the task of integration engineers. This section highlights the difficulties that exist with the current O-R metadata interface and using standard SQL to query it. In the remainder of this chapter metadata from Oracle 9i will be printed with a bold font.

4.1.1 Long Queries and the Need for O-R Expertise

In the O-R model metadata related to different O-R structures is grouped together in virtual tables because the information is related but this can lead to confusion to the O-R database user. For example, the **ALL_COL_TYPES** virtual table contains metadata for two different types of collection *virtual tables* and *varrays*. Virtual tables are of variable length and the contents of the table are not in any particular order, the varray however needs to specify a length at creation time and does maintain a specific order. Although the difference is only one column of metadata combining the metadata implies that the metadata is common to both structures but this is not the case. *Example 4.1 (A)* shows the query for retrieving varray metadata and *example 4.1 (B)* shows the query for nested table metadata.

Example 4.1 Varry and Nested Table Metadata queries

(A) Varray Metadata Query

select owner, type_name, upper_bound,

elem_type_mod, elem_type_owner, elem_type_name

from all_col_types

(B) Nested Table Metadata Query

select owner, type_name, elem_type_mod,

elem_type_owner, elem_type_name

from all_col_types

The FDBS engineer, querying an O-R database for metadata needs to write long SQL queries. To retrieve the metadata for *constraints* the query in *example 4.2* is needed. Generally when trying to rebuild a constraint a typical set of metadata is needed each time and not a random subset. In the O-R model different types of metadata are combined with the structural metadata which means one cannot select all the contents of a virtual table but instead a long and awkward query is needed to retrieve the desired information, see *example* 4.2. For a FDBS engineer the ability to query the structure of a schema through a PDA is useful so the engineer can visit distributed sites to query component schemas. Long SQL queries are not suitable for a PDA as the input device is non-conventional and unsuitable for long text input.

Example 4.2 Constraint SQL Metadata Query

select owner, constraint_name, constraint_type, table_name,

search condition, r_owner, r_constraint_name, delete_rule,

status, deferrable, deferred, validated, generated, bad, rely

from all_constraints

4.1.2 Attribute Metadata

The virtual table for attribute metadata is the **ALL_TYPES_ATTRS** virtual table. From this virtual table SQL can retrieve part of the necessary metadata as shown in *example 4.3*. When analysing the metadata for attributes it is useful to be able to deduce whether or not an attribute is a collection. In the O-R model however collections of a particular type are defined as types; for example a collection of type person object can be contained in an object that is of type people, although it is a collection of another type it is defined as a type. The attribute metadata for **attr_name** and **attr_type** represent the wrapper type for the collection and only the name (semantic meaning) gives a hint to whether it is a collection (i.e. *collection_person*). In *example 4.3* the query is made over two tables **ALL_TYPE_ATTRS** and **ALL_TYPES**. From the **ALL_TYPES** table

metadata which tells the user whether a type is a collection is added to the metadata view on attributes that is described in this report. To correctly retrieve the metadata for attributes using SQL and deduce what is necessary for a engineer to rebuild a schema needs expert knowledge on the O-R model and SQL.

Example 4.3 SQL Attribute Metadata Access

SELECT type_name, type_owner,all_types.typecode

attr_name, attr_type_name, attr_type_owner

attr_type_modifier, inherited, attr_no

FROM all_type_attrs, all_types

WHERE all_types.owner ='schema_owner'

AND all_types_attrs.owner ='schema_owner';

4.1.3 Cardinality of Relationships

Currently in O-R metadata there is no view that directly represents the cardinality of relationships. In order to illustrate the cardinality of relationships it is necessary to be able to deduce whether an attribute is a collection. From the available O-R metadata we can create an `ATTRIBUTES_COLLECTIONS` view which is shown in *example 4.4*. In the **`ALL_TYPE_ATTRS`** view it is not possible to learn whether an attribute is a collection, as each collection is defined as an independent type therefore in the **`ALL_TYPE_ATTRS`** view every attribute is listed as a type. The details of the type of each attribute must be checked from the **`ALL_TYPES`** view in order to discern whether it is a collection. To discern what type is in the collection, **`ALL_COLL_TYPES`** need to be examined. Hence the structure of the view in *example 4.4*. `owning_type` is the type in which the attribute appears. `attribute_name` is the name of the attribute (name of collection) and `type_of_attribute` is wrapper type for the collection. `type_in_list` gives the type of object in the collection and `coll_type` gives the type of collection ie `VARRAY` or `NESTED_TABLE`.

Example 4.4 The `ATTRIBUTE_COLLECTIONS` view.

CREATE or REPLACE view ATTRIBUTE_COLLECTIONS as

SELECT all_type_attrs.type_name OWNING_TYPE, attr_name

ATTRIBUTE_NAME, all_type_attrs.attr_type_name TYPE_OF_ATTRIBUTE,

elem_type_name TYPE_IN_LIST, COLL_TYPE from all_types, all_type_attrs, all_coll_types

WHERE all_types.type_name = all_type_attrs.ATTR_type_name

AND typecode = 'COLLECTION'

AND all_type_attrs.ATTR_type_name = all_coll_types.type_name

AND all_coll_types.owner = 'schemaName' and all_types.owner = 'schemaName'

AND all_type_attrs.owner = 'schemaName'

The cardinality relationship between types A and B depends on how many of type A is in type B and how many of type B is in type A which is then described as *one:many*, *many:many* or *one:one* relationship etc. For an O-R type the *many* side of the relationship can be *'deep'* or *'shallow'*. The many side is *deep* when we are dealing with a collection and it is necessary to look beyond the **ALL_TYPE_ATTRS** view. On the other hand the *many* side of the cardinality of a relationship can be *shallow*, which means directly in type A there are instances of type B that can be discerned from the **ALL_TYPE_ATTRS** view. For example type student has a teacher and a headmaster, the headmaster may also be of type teacher therefore the student has many instances of type teacher (but does not have a collection). The many side of a cardinality relationship can be shallow or deep, nested table or `varray`.

In the **ALL_TYPE_ATTRS** view there is no metadata on whether an attribute is a collection. Each attribute needs to be checked in the `ATTRIBUTE_COLLECTIONS` view to see if it is a wrapper type for a collection or a normal type. Once this information is discerned from the available O-R metadata it is possible to build the cardinality relationships. It is difficult to implement cardinality relationships using SQL therefore in this research project a procedural programming language was used to manipulate the existing metadata.

4.1.4 Inheritance

In the **ALL_TYPES** metadata view there is inheritance metadata. The columns `FINAL`, `INSTANTIABLE`, `SUPERTYPE_NAME` and `TYPE_NAME`, `LOCAL_ATTRIBUTES` and `METHODS` describe the inheritance relationships for a single type. The metadata views **ALL_TYPE ATTRS** and **ALL_TYPE_METHODS** each have a single metadata attribute `"INHERITED"` which indicates whether a method or attribute has been inherited or not. This metadata is adequate to describe the inheritance relationship for a single type, attribute or method.

FDBS engineers on the other hand, need to be able to quickly deduce the over all structure of a schema so they can quickly integrate it into the FDBS. To get all the inheritance

information relating to a certain type (root type), the entire inheritance tree would need to be traversed and metadata related to whether the type was instantiable or final would need to be collected so as to be presented to the user. A relational query, see *example 4.5*, can be used to get the necessary metadata however in order to present the metadata in a meaningful way to the engineer a procedural programming language is needed to create the inheritance tree either graphically or in terms of ordering the text result.

Example 4.5 The inheritance metadata query.

SELECT SUPERTYPE_NAME, TYPE_NAME, LOCAL_ATT

LOCAL_METHODS, FINAL, INSTANTIABLE

FROM ALL_TYPES

GROUP BY SUPERTYPE

4.2 Metadata Query Language and Interface

In this section aspects of the O-R Metadata Query Language (OR-MQL) that was designed in this research project will be presented. The differences between this language and what is currently available through SQL will be highlighted though a set of examples and diagrams. The full language listing is in Appendix B. OR-MQL offers a simple but complete query language for querying all aspects of O-R metadata. While describing the language a description of the O-R metadata interface will be presented.

The schema used in this section describes a news agency which consists of reporters, editors, presenters and different types of news programs and reports.

4.2.1 The Cardinality of relationships

Having the ability to discern the cardinality of relationships in a schema greatly aids the FDBS engineer in deducing the overall structure of a schema. The cardinality of relationships tell the engineer the relationships between complex types in the schema.

Metadata for the cardinality of relationships between types is not directly represented in O-R databases. The available metadata needs to be manipulated to produce them. The views **ALL_TYPES**, **ALL_TYPE_ATTRS** and **ALL_COLL_TYPES** are manipulated by a procedural programming language to produce the cardinality of relationships. Low level metadata and high level metadata are provided to allow the browsing of cardinality metadata on two different levels, they are described in *table 4.1* and *table 4.2* respectively.

Column Name	Data Type	Description
TYPEA	VARCHAR2(50)	The first type in the relationship.
OWNER	VARCHAR2(50)	The owner of the types in the relationship.
TYPEB_OWNER	VARCHAR2(50)	The owner of the second type in the relationship.
CARDINALITY	VARCHAR2(15)	The cardinality between the two types.
REFTABLE_NUM	NUMBER	Number to reference cardinality definition table.

Table 4.1: High-level cardinality table.

The low level view provides too much information for the general user so we also provide a high level view which is described in *table* 4.1. This high level metadata provides the names of types, their cardinality and a reference to a look up table. The look-up table describes variations of the cardinality relationships and it is listed in the appendix.

When discerning the structure of an O-R schema and examining the cardinality between types an engineer needs the metadata listed in *table 4.1* or *table 4.2*. The queries in *example 4.6* illustrates the `cardinality` schema queries. If the query is *cardinality OWNER.TYPEA* then all the relationships related to type A will be returned. If the query is *cardinality OWNER.TYPEA.TYPEB* the cardinality relationship between type A and type B will be returned. Finally if the query is *cardinality OWNER.SCHEMA* then the cardinality relationships for the entire schema will be returned, grouped by type. The third query listed in *example 4.6* returns the cardinality look up table.

Example 4.6 Schema query for O-R cardinality.

[select] cardinality OWNER.[[TYPEA[.TYPEB]] | SCHEMA_NAME]

[select] lowllevelcardinality OWNER.[TYPEA[.TYPEB]]|SCHEMA_NAME]

[select] carRefTable

Querying for Cardinality of Relationships in Example News Agency Schema The query *select cardinality Admin.Newsagency* will return the cardinality of all relationships in schema *Newsagency* owned by *Admin*. In our example *News Agency* schema this will return descriptions of the cardinality for all seven complex relationships in the schema. This is illustrated in *figure 4.1*. It can be discerned from this diagram how the instantiable objects in the schema relate to each other. For example, there can be many presenters that present the news (or only one), a single news item can be in many weekly-news programs

Column Name	Data Type	Description
OWNER	VARCHAR2(50)	The owner of the schema where the types are defined.
TYPEA	VARCHAR2(50)	The first type in the cardinality relationship
TYPEB	VARCHAR2(50)	The second type in the cardinality relationship
DEPTA	VARCHAR2(8)	Whether or not the type of the attribute A was found deep or shallow.
COLLECTION_TYPEA	VARCHAR2(50)	Name of the type in the collection for attribute A.
COLL_TYPEA	VARCHAR2(50)	Whether the collection of type A is a NESTEDTABLE or VARRAY.
CARDINALITY	VARCHAR2(10)	The cardinality relationship of the type.
COLL_TYPEB	VARCHAR2(12)	Whether the collection of type B is a NESTEDTABLE or VARRAY.
DEPTB	VARCHAR2(8)	Whether the type of the attribute B was found to be deep or shallow.
COLLECTION_TYPEB	VARCHAR2(50)	Name of the type in the collection for attribute B.

Table 4.2: Low-level cardinality metadata.

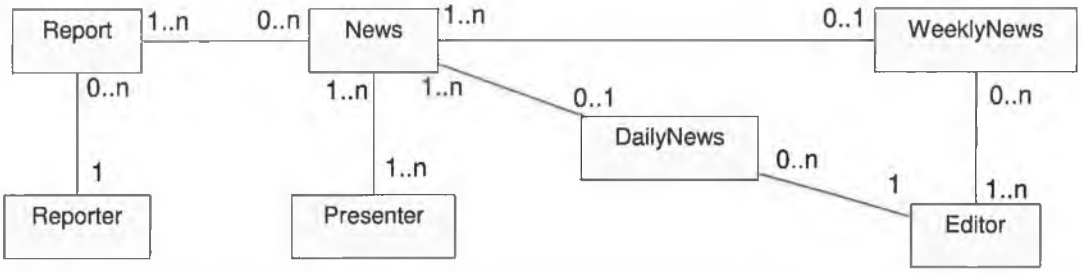

Figure 4.1: Cardinality of Relationships in Example News Agency Schema.

Name	Type	Description
PARENT	VARCHAR2(50)	Parent type in the inheritance relationship.
CHILD	VARCHAR2(50)	Child type in the inheritance relationship.
INSTANTIABLE	VARCHAR2(5)	Whether or not this type is instantiable (true||false).
FINAL	VARCHAR2(5)	Whether or not this type is final (true||false).

Table 4.3: OR-MQL Inheritance Metadata.

or many daily programs or none at all and a news program can consist of many news reports, or only one.

4.2.2 Inheritance Metadata

Objects are defined in terms of classes. Objects of the same class all have the same structure, characteristics and behaviour. The inheritance relationship allows a user to specialise a particular class, by adding some extra characteristics and at the same time it saves the engineer the time of developing a new structure. Currently in the O-R metamodel it is possible to deduce the inheritance information for an individual, class, attribute or method.

A FDBS engineer needs to examine the entire inheritance tree in order to deduce how to integrate this structure in to the federation. More generally for an engineer looking to extend a complex schema, the ability to see the entire inheritance structure, will allow him to more clearly see where he might need to extend th inheritance tree. In OR-MQL we have provided an interface to inheritance metadata that will allow engineers to retrieve inheritance information for the entire schema, or a subset of it.

Example 4.7 Schema query for O-R inheritance.

[select] inheritance OWNER.SCHEMA_NAME

[select] inheritance OWNER.TYPE

Example 4.7 illustrates OR-MQL queries for inheritance metadata.The first query queries inheritance metadata over an entire schema.The results from the query are in the format described in *table 4.3*. The results are returned with the root of the largest inheritance tree in the schema first. Then all child nodes of this tree in alphabetical order are listed. The metadata *instantiable* and *final* relate to the child node and indicate to the engineer whether or not objects can be created from this type structure and whether or not this type definition can be extended further. After listing all the children of the root, next the children's children will be listed and so on until all the leaf nodes are reached. When one inheritance tree has been listed the next inheritance tree (in order of size) will be returned and listed in the same way.

Some models are very complex and their inheritance tree's are extensive. Retrieving all the inheritance information in such a case may not be suitable. The second inheritance query in *example 4.7* retrieves inheritance information starting at a certain type in the inheritance tree. The inheritance information is retrieved in the same format as the first example, but since only a fraction of the inheritance tree is being returned, the metadata is easier to read and use.

Querying for Inheritance Metadata in the Example News Agency Schema The query *select inheritance admin.NewsAgency* will return the metadata listed in table 4.4. From this table one is able to clearly discern the inheritance relationships for the *News Agency* schema. One can distinguish which types are abstract and which are instantiable as well as where an engineer might want to extend the structure (leaf nodes). From a FDBS engineers perspective this metadata query and interface is useful because it gives the engineer an overview of the entire inheritance relationships for a schema, which can then be examined to see how they will merge with other component schemas into the federation. *Figure 4.2* illustrates the inheritance metadata from *table 4.4*.

4.2.3 Attribute Metadata

The attributes of a type describe the structure of a type. The values of the attributes of an object, at any instance in time, describe the state of that object. An attribute can be a number, string, reference, collection or a user defined type. All of these can be deduced from the **ALL_TYPE_ATTRS** except whether the type is a collection. Therefore an extra column of metadata is added to the metadata interface (`COLLECTION`).

The OR-MQL schema query for attribute metadata is listed in *example 4.8*. The query will either return all the names of the attributes for a particular type in a particular schema or

Parent	Child	Instantiable	Final
NULL	PERSON	FALSE	FALSE
PERSON	EMPLOYEE	FALSE	FALSE
EMPLOYEE	REPORTER	TRUE	FALSE
REPORTER	EDITOR	TRUE	TRUE
REPORTER	PRESENTER	TRUE	TRUE
NULL	PROGRAM	FALSE	FALSE
PROGRAM	DAILYNEWS	TRUE	TRUE
PROGRAM	NEWS	TRUE	FALSE
PROGRAM	WEEKLYNEWS	TRUE	TRUE
NEWS	INTERNATIONALNEWS	TRUE	TRUE
NEWS	REGIONALNEWS	TRUE	TRUE
NEWS	SPECIALINTERESTNEWS	TRUE	TRUE
NEWS	WEATHERNEWS	TRUE	TRUE

Table 4.4: Retrieving Inheritance Metadata for News Agency Schema.

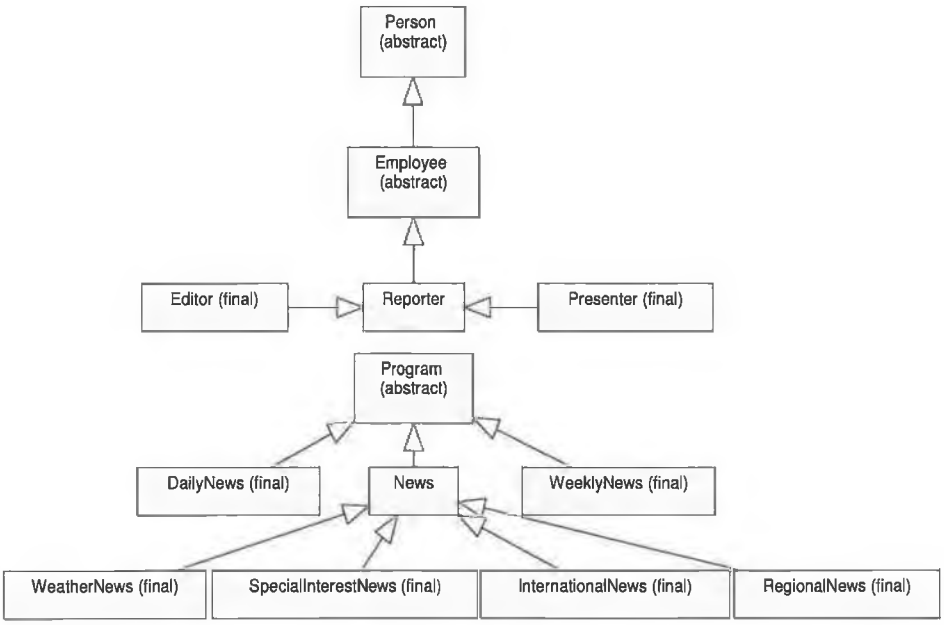

Figure 4.2: Inheritance Relationships in Example News Agency Schema.

ATTR_NAME	Date Produced	Length	NewsPrograms	Editor
ATTR_TYPE_MODIFIER	NULL	NULL	NULL	REF
ATTR_TYPE_NAME	DATE	INTEGER	NEWS_ITEMS	EDITOR
ATTR_TYPE_OWNER	SYS	SYS	ADMIN	ADMIN
INHERITED	TRUE	TRUE	FALSE	FALSE
COLLECTION	FALSE	FALSE	FALSE	TRUE

ATTR_NAME	Title	Media Type	Format	Media	Date
ATTR_TYPE_MODIFIER	NULL	NULL	NULL	NULL	NULL
ATTR_TYPE_NAME	TEXT	TEXT	TEXT	BLOB	DATE
ATT_TYPE_OWNER	SYS	SYS	SYS	SYS	SYS
INHERITED	TRUE	TRUE	TRUE	TRUE	FALSE
COLLECTION	FALSE	FALSE	FALSE	FALSE	FALSE

Table 4.5: Retrieving Attribute Metadata from the News Agency Schema.

all the details of the attributes depending on whether the key word *NAMES* is included or not.

Querying Attribute Metadata in the Example News Agency Schema The query *select attributes Admin.NewsAgency.DailyNews* will return detailed metadata describing the structure of attributes for type *DailyNews* in schema *NewsAgency* owned by *Admin.* The retrieved metadata is listed in *table 4.5.*

Using the three OR-MQL queries presented in this section an FDBS engineer who is trying to integrate the example *News Agency* schema in to the FDBS will have a clear idea of the structure of the schema and where it will merge into the federation. *Figure 4.3* illustrates the results of using the queries that we have presented so far. The complete metadata interface and query language for OR-MQL is presented in the appendix. These examples suffice in illustrating the power and simplicity of OR-MQL.

Example 4.8 Schema query for attribute metadata.

[select] attributes [NAMES] OWNER.SCHEMA.TYPE

4.2.4 Relational Metadata

The relational and object part of the O-R database do overlap but are also distinct. The object part describes the logical structure of the schema, i.e. how objects relate together, the behaviour of objects, the structure of types and how and where to extend the schema. The relational part describes how and where the physical objects are stored; how when objects are accessed or when certain external events occur, events are triggered; how

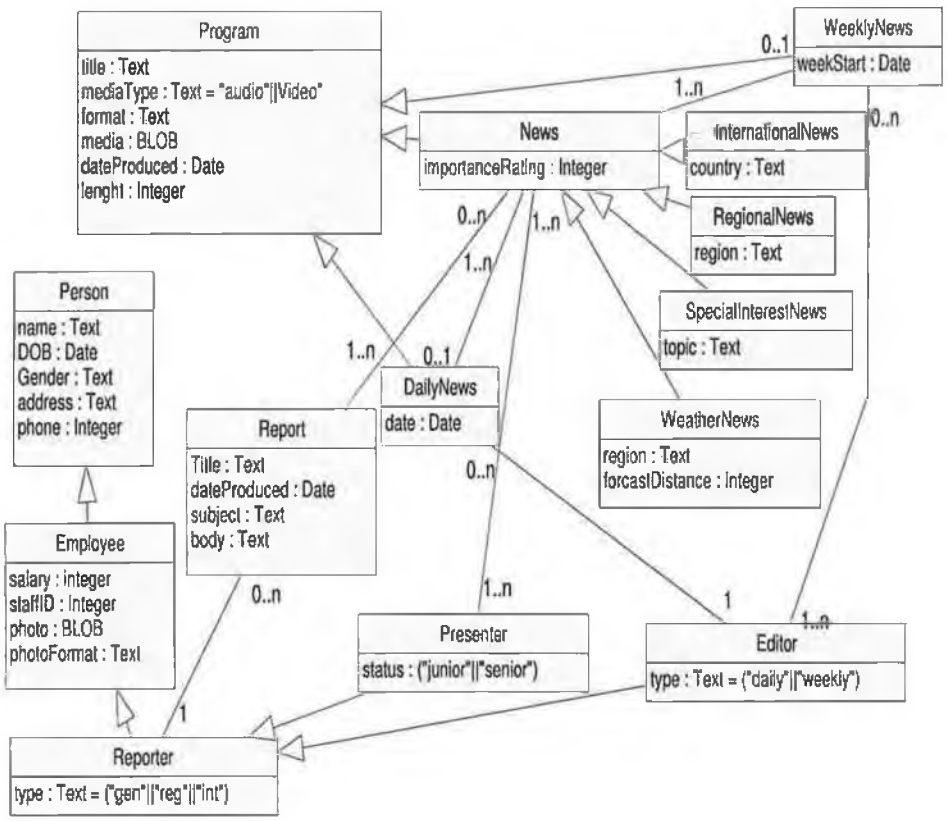

Figure 4.3: News Agency Schema.

access to objects or tables are constrained by certain rules; and how physical objects can be viewed differently. Generally the object part describes the logical structure of data and the relational part describes the physical storage and manipulation of the physical data.

The O-R model is built on the relational model. In this research project, views have been added in order to completely describe the object aspect of O-R metadata. The interface to object and relational metadata has been standardised and simple queries can be used to access all metadata. The physical storage metadata, statistical metadata and redundant metadata has been removed from the metamodel. The full metadata interface for O-R metadata and descriptions of the metadata is in Appendix A.

4.3 Summary

In this chapter we analysed currently existing problems with accessing O-R metadata. Through a set of examples OR-MQL was presented which addresses the problems with the current O-R metadata and the interface to it. OR-MQL was demonstrated to be simple enough to be used on a PDA but powerful enough to query the O-R metamodel. It was also illustrated that OR-MQL is suitable for a FDBS engineer who needs to quickly discern the entire structure of a schema in order to merge it in to the canonical model. The complete metadata interface and query language of OR-MQL is presented in the appendix.

Chapter 5

The Mobile Metadata Schema Browser Architecture

Integration Engineers are often faced with a requirement to display and analyse the complex schemas of information systems to be merged. As these systems can be dispersed over a wide geographic area, a Portable Digital Assistant (PDA) provides a flexible means of viewing and displaying schema information. However the browsing process, which is often complex and problematic on a workstation screen, becomes more difficult on the smaller PDA. Using our metadata interface and query language as middleware, a mobile user can query metadata on an object-relational database and automatically display its structure. This application exploits our interface to the extended object Object-Relational (O-R) schema repository to manipulate complex metadata information. The deployment architecture described in this chapter was published in the 54th edition of ERICM (European Research Consortium for Informatics and Mathematics) news[23].

5.1 Deployment Architecture

In *figure 5.1*, the deployment architecture for O-R metadata access is illustrated. In the Mobile Layer, a PDA uses the metadata interface to O-R metadata and query language as middleware in a specific application. The Schema Browser queries the schema using the metadata query language described in *chapter 4*. This includes metadata query options for any object-relational schema and role views. The metadata queries and result set are wrapped in XML to provide a robust, non-proprietary, persistent and verifiable file format for the storage and transmission of data. The result of the metadata query is wrapped in XML. Every modern Internet browser has the capability to present XML in a user friendly

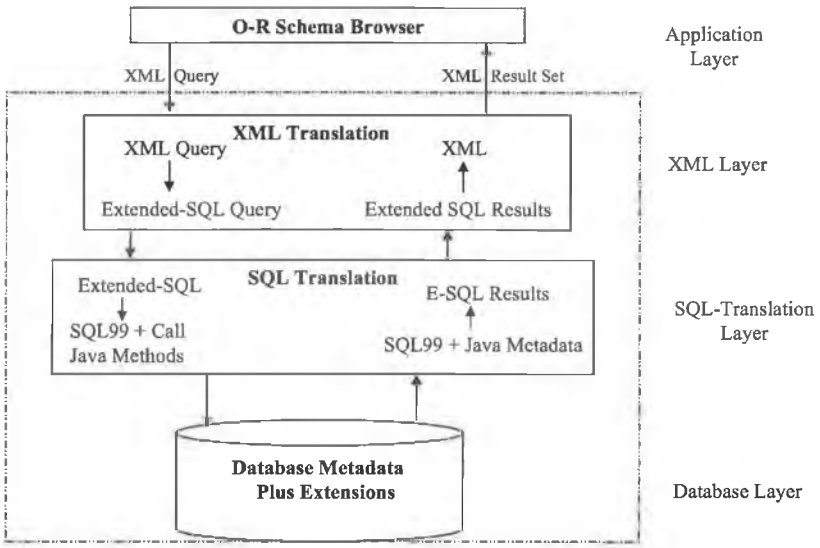

Figure 5.1: Deployment Architecture

way, so we did not need to develop a means to present the metadata results. XML is a standard, trusted format that can be sent safely across computer networks.

The XML Translation Layer resides at the database server. Its purpose is to provide a standard interface to the metadata query language. When receiving a query it is unwrapped to form a metadata query, which is subsequently passed to the SQL translation layer. After execution, results received from the SQL Translation Layer are XML-wrapped using a basic rule set, and then returned to the application.

The SQL Translation Layer is where most of the metadata processing takes place. Current approaches to interfacing metadata for object-relational databases were examined before this layer was specified. The SQL Translation Layer accepts a metadata query, which is parsed to invoke a sequence of actions against the schema repository. The results may be comprised of conventional object-relational metadata or role metadata. After the results are restructured according to the interface in *chapter 4*, they are passed back to the XML Translation Layer.

The object-relational schema repository was extended by other EGTV researchers to provide new interfaces to role metadata. Role metadata was added because it adds to the expressiveness of O-R databases. Roles provide temporal aspects to entities, a feature that

is missing in conventional models. Without roles, a new object must be created each time the structure of an object evolves and many complex issues are involved in maintaining such an operation. All of the metadata can be accessed using the interface and the metadata query language described in *Appendix B.* Thus, it becomes accessible to integration engineers using mobile devices.

5.1.1 Implementation of the Mobile Metadata Schema Browser

The mobile device used for implementation and testing was an Compaq IPAQ. The client application ran on the PDA using Personal Java which is a slightly cut down version of Java 1.1. A simple GUI was designed using Java AWT which accepted the query from the user and wrapped it in XML. The query was then sent to the server. The retrieved result is an XML file that is written to a specified directory on the PDA. The result set can be viewed through any web browser that supports XML.

XML Schema files are defined for the queries and result sets. They are developed in XML Spy. On the server unwrapping the queries and wrapping the result set is implemented with Java. The SQL-Translation Layer is a Java program that accesses the database using JDBC libraries and manipulates the available metadata to produce the interface described in *chapter 4.* The SQL-Translation Layer also includes a parser that was written using ANTLR which parses the metadata queries described in *chapter 4.*

5.2 Application to Grade Schema Complexity

The Mobile Metadata Schema Browser is an application that runs on a PDA that allows an engineer to connect to and browse a local database schema, and enables him to discern its structure. This application is unsuitable to throughly test the metadata extensions because it is designed to examine one database at a time and a particular aspect of the database is examined with each query. The program to grade schema complexity was designed and implemented in order to completely test the interface to O-R metadata that is outlined in this research project.

The initial stage of this research project involved extensively researching the O-R metadata for issues and difficulties in order to see what is easy, hard and impossible to do with the available O-R metadata. Much of the necessary metadata for the O-R metadata interface is available but extracting metadata needs long and sometimes complex SQL queries. The cardinality of relationships are not available in the O-R metadata interface and it is not

possible to mine this information using simple SQL queries because data processing and manipulation is needed.

The schema grader program takes an O-R schema and checks it's complexity by measuring the number of types, attributes, methods, constraints, triggers, the different types of cardinality relationships and inheritance. The grade is then computed based on a complexity algorithm. All the types and tables in the schema, the relationships and behaviour on them are graded and the best five types contribute to the overall grade of the schema.

5.2.1 Implementation

The Schema Grader program consisted of four main classes:

- SchemaChecker,
- SchemaGrader,
- HighLevelReport,
- Driver class.

The *SchemaChecker* class generates high and low level cardinality relationship meta-tables. As was discussed in *chapter 4* creating cardinality relationships for O-R schemas needs a procedural programming language. The low level table stores metadata about the actual structure of the cardinality relationship, ie the many side of the relationship can be a varray, nested table, deep or shallow. The high level meta-table is a view of the low level table which includes `TYPEA, TYPEB, CARDINALITY` and an additional column is added on to reference a lookup table. The lookup table describes the different types of cardinality relationship and is listed in the appendix.

The *SchemaChecker* class generates the metadata for the inheritance relationship. The details for this query can be deduced from the available metadata but without a procedural programming language it is not possible to order the results or return them graphically in the form of an inheritance tree. The procedural programming language is used to traverse the tree in the order of: root type, the children of this type (one layer deep in the inheritance tree) and then in alphabetical order the same process is used with each child node until all leaf nodes are reached. The results are stored as a two dimensional array with the first dimension representing the structure of the inheritance tree and the second representing the details of the inheritance relationship for that type. With the

Name	Data Type	Description
OWNER	VARCHAR2(50)	The owner of the type.
TYPE_NAME	NUMBER	The name of the type.
NUM_ATTRS	NUMBER	The number of attributes belonging to the type.
INHERITANCE	VARCHAR2(5)	Does this type have inheritance.
ONE_TO_ONE_REL	NUMBER	The number of one to one cardinality relationships for this type.
ONE_TO_MANY_REL	NUMBER	The number of one to many cardinality relationships for this type.
MANY_MANY_REL	NUMBER	The number of many to many cardinality relationships for this type.
METHODS	NUMBER	The number of methods for this type.
TRIGGERS	NUMBER	The number of triggers on the object table that is implemented to hold instances of this type.
CONSTRAINTS	NUMBER	The number of constraints on the object table that is implemented to hold instances of this type.

Table 5.1: High level table report.

inheritance information stored in this manner it gave us the option of giving extra marks for the complexity of the inheritance tree used.

The *HighLevelReport* class creates a meta-table that is a detailed description of the types for a particular schema. The details for this table are described in *table 5.1.* This table holds statistics for the number of different O-R features that is used by a particular type and the object table that is used to store instances of the type. For instance object features are methods, attributes, inheritance and the different variations of cardinality relationships, where as relational features are the triggers and constraints that are implemented on the object tables that store instances of a particular type.

The *SchemaGrader* class contains algorithms to grade the schemas based on the statistics presented in `HighLevelReport` meta-table. The types are graded on the following scale:

- .25 per attribute (max mark = 2),
- 2 points if inheritance is included,
- 2 points for each instance of behaviour ie constraints, triggers and methods (max points = 6),
- 1 point for each one to one cardinality relationship,

- 2 points each one to many cardinality relationship,
- 3 points for each many to many cardinality relationship.

The maximum number of points for each O-R type is twelve. The best five types in a schema are taken together giving a maximum points of sixty for the schema. The *HighLevelReport* class then writes the results for the grade of the schema to a `SchemaGrades` table that has two columns, SCHEMA_NAME and SCHEMA_GRADE.

An administration tool was designed for the schema checking application but not completely implemented. One feature of the tool would allow the user to place different weights on the object-relational features that are graded. Graphs could be presented to the user of the tool illustrating how many of each O-R feature each student used. This would allow the user to see which areas the student as a whole were not confident in. Finally it was planned to include in the tool a feature that would allow the user to view a graph of the distribution of student's grades. This tool would allow the user to view which areas the students as a whole were strong in and which areas they needed improvement. It would allow the user to re-weigh the marks given for each feature based on these statistics.

5.2.2 Testing

This program was used to mark undergraduate student schemas over the college year. While marking 200 schemas atomically the interface and cardinality relationship extensions were tested rigorously. The distribution of percentage grades for the student schemas was focused around 50%, with the majority between 40% and 60% which is the same distribution as previous years when the schemas were manually marked.

Ten percent of the total schemas, taken from the higher range (>60%), lower range(<40%) and middle range of grades(40%-60%), were selected to be tested manually. Testing graded schema's from the lower range would ensure that structures that were supposed to get marks were not being overlooked. Testing a sample from the higher range would ensure that there was no scenarios were a student could be awarded marks in error. These sample schemas were checked manually against `HighLevelReport` table to ensure the statistics generated automatically are correct. The algorithm for marking the schemas was also checked against this sample selection of schemas.

5.3 Conclusions

The Mobile Metadata Schema Browser is an aid to integration engineers who are faced with the complex task of integrating schemas that can be widely distributed and semantically different. It is not possible to completely automate the integration process because understanding the meaning of information must be achieved in collaboration with the local administer who created the schema. Some tools were researched that try to automate parts of this process but the overheads in preparing the schema to be integrated and making reasonable deduction from the results proved greater overhead than manually integrating the schema. The Mobile Metadata Schema Browser is a light client application that runs on a PDA that allows an engineer to navigate metadata of an O-R database on site while discussing semantic details with the administrator. A tool to view the results did not need to be implemented as a web browser can be used to view the XML results.

Prototypes of the Mobile Metadata Schema browser have been implemented. The application has the potential to be substantially augmented in many ways. For example, currently we are using a web browser to navigate the XML result sets. If XML Style Sheets were used to present the data in a more meaningful way it would be easier for the engineer to navigate the results. It is not possible to compile the XSL (XML Style Sheets) on the PDA because it does not have the necessary processing power. Instead the XSL file can be generated and compiled on the server and HTML can be sent to the client, or posted to the web where it can be viewed.

Currently the Mobile Metadata Schema Browser is useful for navigating and understanding Oracle's O-R schema's, a substantial improvement in this tool would be to extend it's capabilities to navigate other vendors schemas. Since we have defined an interface and query language to the O-R model it is possible to use our query language to query Object-Oriented models, or relational models but the mappings to the underlying metadata will be vendor specific and needs to be specified for each vendor. Such an improvement to the tool would aid the integration specialist who is working in an environment where the local databases store the data in different datamodels.

An O-R model schema grader was designed, implemented and tested in this research project. The purpose of this application was to prove our metadata interface to the O-R database was complete and that the extensions that were written to address existing shortcoming were implemented correctly. The application checked which O-R structures were present and graded them according to a certain criteria. 200 undergraduate students were given a project over two months to create a schema using many of the complex O-R

structures and their schema was marked atomically according to what was present. A number of schemas were manually checked (using *Oracle Enterprise Manager*) to ensure the metadata interface (the part of it used in this program) successfully picked up the complex structures in an O-R schema and the relationships between them, and that the grade awarded to the student, that was generated atomically was correct. This program demonstrated that the metadata interface designed in this research project were implemented correctly and useful as middleware in an application to mark the complexity of O-R schemas.

Chapter 6

Conclusions

In this research project we examined the difficulties faced by an integration engineer when integrating component database schemas into a federation of databases. Through our research we discovered that the most suitable database model for representing data in the federated database management system is the object-relational (O-R) model. This model is expressive enough to capture the semantic meaning of all component databases but due to its expressive power it is also very complex.

Through thorough examination of the O-R model it was discovered that its metadata interface and the means for navigating it was cumbersome and awkward to use. It is true that that O-R model is much more expressive than the relational model, which is widely used but is unsuitable to as the CDM for an FDBS because it is not expressive enough. It is also true that O-R model leads to easier migration of data from the relational model to the FDBS than the object-oriented model because it supports relational features and constructs especially designed to migrate the data. The combination of relational and object-oriented features means that O-R model is more expressive than either of these models alone. Without a clear and complete definition of metadata for the object-relational model its potential as the CDM for the FDBS can not be reached, as a user can not be sure the metadata description of the data is accurate and the overhead of mining the required metadata would be too great.

A complete and concise metamodel interface is needed in order to allow interaction with the database, improves its data quality, support the system integration process and also database maintenance, analysis and design. In this research project we analysed the object-relational metamodel from the perspective of a federated database management system integration engineer and the tasks that he must carry out during the integration process. We discovered that the process of integrating component schemas into the FDBS can not be

completely automated and collaboration with the local database system administrators is necessary in order to discern the semantic meaning of the metadata in the local databases. For example, income in database one is take home pay, while in database two income is gross pay. It will remain impossible to automate the integration of component database schemas completely until a dictionary of common terms can be agree upon and each database administrator that is part of the FDBS can agree to adhere to them strictly.

A dictionary of terms would only be useful if it is used when the FDBS is built and each component database used is forced to only use words from the dictionary. When the databases are already in place and more importantly in use before the FDBS is built the cost of implementing a strict set of terms that are clearly defined is too expensive and not practical. The larger the number of component databases becomes the more expensive this task becomes. Other tools used for assisting the integration specialist that were designed in various research projects have been examined. They mainly try to some degree to automate the integration process but because of the difficulties in discerning semantic correlations between respective schemas in the FDBS the pre-configuration of the automation tool and the post-checking of results proves in most cases a larger overhead than integrating the schemas manually.

The Mobile Metadata schema browser described in this research project took a different approach by providing the integration specialist with a tool that would assist him in a certain aspect of the integration process that we think can not be automated and will be a permanent obstacle to the integration specialist. The function of this tool is to provide the integration specialist a means to query local schemas in the presence of the local database administer. The local database administer can then be asked semantic details about the metadata and illustrate his point on tools that he is familiar with. Similarly for the integration specialist for each local site that he visits he has a tool that he is familiar with and confident with, which ensures that his job will be completed with less inconvenience, in less time, and with less cost.

The FDBS is a complex structure consisting of mainly five different layers. The local database layer consists of the schemas from local databases. The component layer is the layer that holds the local schema that has been migrated to the CDM (common data model). The export layer consists of a portion of the component layer that the local administer deems suitable to share with the federation. This layer is useful as the local administrator can produce different export layers that are suitable to be share with different groups of users. Each of these different export layers are incorporated into a federated schema (federated layer). The federated layer joins together seamlessly a set of

export schemas. Export schemas will be joined together that are suitable to viewed by the same type of user. The federated layer is very expensive to engineer, merging the various export schemas and finding correlations between the semantics of their metadata is a time consuming and a complicated engineering task. It is possible that certain data are stored simultaneously in multiple databases. It is also possible that this data is managed and presented differently. At this layer it is imperative that the metadata describes the data completely and correctly, and it is also helpful if a common dictionary of terms is used to build each export schema. The final layer is the external layer which is a view of the federated layer suitable for a certain subset of users.

Each layer of the FDBS is a layer of metadata. Its goal is to provide a seamless information source that a user can query with out needing to know the complexities that comprise the FDBS. The CDM needs to be very expressive to ensure that all the data that is stored in the component databases is represented correctly. If the CDM is not expressive enough it is possible that a data item in the local databases can not be represented in the federation correctly or at all. If it is represented incorrectly it can cause inconsistency in the whole system, which makes the FDBS of little value and actually a burden to users who are unaware that the information they receive is inaccurate. Not only is it important that the CDM is expressive, it is also essential that the CDM is clearly and well defined with metadata. Due to the fact that the CDM needs to be powerful and expressive, it means that the metamodel that describes the datamodel also needs to be complete, accurate, clear and concise.

Many problems arise if this is not the case. Each layer of the FDBS is a layer of metadata which is augmented with new constructs, logic and algorithms. The basis of the FDBS is the CDM which is its foundation. At the lowest layer the local schema is translated into CDM to make the component layer. A number of problems may exist at this layer due to poor metadata. The first is due to the metamodel of the local database schema not being clear, accurate or concise. If the metamodel for the local database is not accurate when migrating the data to be represented in the CDM the data may be misrepresented. If the metamodel for the local database is unclear or cumbersome to use this makes it difficult to accurately discern the data's meaning. Furthermore since most of the integration process is manual and implemented by the integration specialist; if it is unclear it will lead to human error and again the misrepresentation of data at higher layers of the FDBS.

If the CDMs metamodel is not complete, accurate, clear and concise it renders the FDBS useless as an information source because the data will be corrupted. For example when migrating data from a local database to the CDM (component schema) if the data in the

local schema is misrepresented in the component schema there is no way to compensate at higher levels of the FDBS and the corruption of the data can only get worse. Each layer of the FDBS builds on the layer below it, before another layer is built the engineer must be confident the data and metadata are correct. If at the export layer there are many discrepancies, that problem will be many factors greater when the data is merged into the federated layer. In the federated layer there will be inaccurate correlations of data, data that match might not be supposed to match and data items that were supposed to match might not be discovered.

In this research project we examined in detail the O-R model with the view that this model could be used for the local databases in the FDBS and also for the CDM. A number of deficiencies where discovered with the models metamodel. These deficiencies were addressed and in this research we have presented a complete, accurate and concise interface to the O-R metamodel which can be used by the integration specialist. We also designed and specified a metadata query language that is powerful enough to query all aspects of the O-R metamodel but simple and concise enough to be implemented on a PDA. The query language is also simple enough to not need expertise in SQL or profound knowledge of the complex O-R metamodel.

In the process of doing this research work different applications of a well defined metadata interface and query language for the O-R metamodel were examined. Context aware mobile computing is one aspect of the computer industry which can benefit when databases are well defined with metadata and an interface to them is simple enough that it can be used on a PDA. Context aware mobile computing is a mobile computing paradigm in which applications can discover and take advantage of contextual information (such as user location, time of day, nearby devices and user activity). At the moment most of the context aware applications are client server based, the client sends its position to the server and the server checks its information repositories for information it has regarding the users current position. Generally each server offers a specific service to the user which the user must sign up for, examples include, restaurant information, traffic information, shopping information, weather updates etc.

The growing popularity of wireless networks offers another avenue for the research of context aware mobile computing which moves away from the client-server approach. If there is a database on the wireless network that stores context information about its local environment and this database has a clear metadata interface it is possible that the metadata interface can be used as middleware between the database and the mobile users mobile application. The mobile users application could discern the structure of the

database from the metadata middleware and present the user with the contents of the database. For example a mobile user walks into record store and the record store keeps all of its information about what it sells (i.e. records, videos, CDs etc.) in a database on its wireless network. This database is made available to the mobile user through the wireless network and the metadata middleware which means that the user can browse for information about their purchase without consulting a member of staff. This is made possible with the metadata interface described in this research project and the fact that the query language associated with it is suitable to be accessed from a mobile device. This change in focus in mobile aware context computing has the potential to open it up to a larger market. It is no longer the mobile users role to pay to subscribe to context aware mobile services; instead it is in the sellers interest to make their products available to the consumer so they can be sold.

6.1 Future Work

The main focus of the Mobile Metadata Schema Browser is to provide a means to allow an integration engineer to query local O-R schemas in a federation at the local database site while he is in the process of integrating the local schemas into the FDBS. A considerable improvement to the Mobile Metadata Schema Browser is to incorporate the ability to query different metamodels. Then the integration specialist can query a wider range of information sources while in the process of integrating them into the federation.

This work could be augmented by incorporating the metadata query language described in this research project with SQL. The query language described in this research project was designed specially for a PDA. While using a PDA, a keyboard is available but it is cumbersome, time consuming, error prone and often irritating to use, therefore we implemented a simple query language were it is possible to implement a point and click application using a stylus. Incorporating the metadata interface and query language with SQL would produce a powerful (but complex) metadata tool that is suitable to not only query data and metadata, but also to manipulate it.

This research project opened up the area of the area of context aware mobile computing but because this was not the focus of the research project this avenue was not explored exhaustively. By providing a complete metadata query language and interface to a database that can be used on a PDA to discern the structure of a local database that is on a wireless network, this moves the focus of Context Aware Mobile Computing from the currently popular client server approach. Future research in this area could prove fruitful and

interesting.

Bibliography

[1] Rapid Prototyping of Mobile Context-Aware Applications: The Cyberguide Case Study. In *The Proceedings of the Second Annual International Conference on Mobile Computing and Networking*, pages 3:421–433. ACM Wireless Networks, 1997.

[2] C. Batini, M. Lenzerini, and S. Navathe. A Comparative Analysis of Methodologies for Database Schema Integration. *ACM Computing Survey*, Vol 18, 1986.

[3] J. Berlin and A. Motro. Autoplex: Automated Discovery of Content for Virtual Databases. In *Lecture Notes in Computer Science*, pages 108–122. Springer, 2001.

[4] J. Berlin and A. Motro. Database Schema Matching Using Machine learning with Feature Selection. In *The Fourteenth International Conference on Advanced Information Systems Engineering*. Springer, 2002.

[5] P. Bernstein, A. Halevy, and R. Pottinger. A Vision for Management of Complex Models. *SIGMOD Record*, 29(4):55–63, 2000.

[6] O. Buyukkokten, H. Molina, A. Paepcke, and T. Winograd. Power Browser: Efficient Web Browsing for PDAs. In *Proceedings of the Conference on Human Factors in Computing Systems*, pages 430–437. ACM Press, 2000.

[7] Y.H. Chang. A Graphical Query Language for Mobile Information Systems. *ACM SIGMOD Record*, 32(1), 2003.

[8] G. Chen and D. Kotz. A Survey of Context-Aware Mobile Computing Research. Technical Report TR2000-381, Dept. of Computer Science, Dartmouth College, 2000.

[9] W. Chu and G. Zhang. Associations and Roles in Object-Oriented Modeling. In *International Conference on Conceptual Modeling / the Entity Relationship Approach*, pages 257–270. Springer-Verlag, Berlin, 1997.

[10] E. F. Codd, S. B. Codd, and C. T. Salley. Providing OLAP to User-Analysts. *White Paper, www.arborsoft.com/papers/coddTOC.html*, 1995.

[11] A. Dey, M. Futakawa, D. Salber, and G. Abowd. The Conference Assistant: Combining Context Awareness with Wearable Computing. In *The Proceedings of the 3rd International Symposium on Wearable Computers*, pages 21–28. IEEE Computer Society Press, 1999.

[12] H. Do, S. Melnik, and E. Rahm. Comparison of Schema Matching Evaluations. In *The Proceedings of the 2nd International Workshop on Web Databases (German Informatics Society)*. Erfurt, 2002.

[13] A. Doan and J. Madhavan. Reconciling Schemas of Disparate Data Sources: A Machine Learning Approach. In *Proceedings of ACM SIGMOD Conference on Management of Data*. Santa Barbara, 2001.

[14] R. Finkelstein. Understanding the Need for On-Line Analytical Servers . *White Paper, www.arborsoft.com/papers/finkTOC.html*, 1995.

[15] M. Gyssens, L. Lakshmanan, and I. Subramanian. Tables as a Paradigm for Querying and Restructuring. In *Symposium on Principles of Database Systems*, pages 93–103. ACM Press, 1996.

[16] W. Ho, J. Forman, and J. Kannry. Portable Digital Assistant PDA Use in a Medicine Teaching Program. Center for Medical Informatics, Mount Sinai Medical Center, New York, 1998.

[17] G. Kiczales, J. Ashley, L. Rodriguez, A. Vahdat, and D. Bobrow. *Metaobject Protocols: Why We Want Them and What Else They Can Do*, pages 101–118. The MIT Press, Cambridge, MA, 1993.

[18] L. Lakshmanan, F. Sadri, and I. Subramanian. SchemaSQL: A Language for Interoperability in Relational Multidatabase Systems. In *22nd International Conference on Very Large Databases (VLDB 1996)*, pages 239–250. Morgan Kaufmann, 1996.

[19] J. Madhavan and P. Berstein. Generic Schema Matching with Cupid. In *VLDB*. Springer, 2001.

[20] S. Melnik and H. Garcia-Molina. Similarity Flooding: A Versatile Graph Matching Algorithm . In *18th International Conference on Data Engineering*. IEEE Computer Society, 2002.

[21] R. Miller, M. Hernandez, L. Haas, L. Yan, H. Ho, and R. Fagin. The Clio Project: Managing Heterogeneity. *ACM SIGMOD Record*, 30:78, March 2001.

[22] I. Mumick and K. Ross. Noodle: A Language for Declarative Querying in an Object-Oriented Database. In *Deductive and Object-Oriented Databases*, pages 360–378. Springer, 1993.

[23] G. OConnor and M. Roantree. A Mobile Schema Browser for Integration Specialists. *European Consortium for Informatics and Mathematics*, 54, 2003.

[24] R. Orfali and D. Harkey. *Client Server Programming with JAVA and CORBA*. John Weily and Sons, 1997.

[25] S. Ram and V. Ramesh. *Schema Integration: Past Present and Future*, chapter Management of Heterogeneous and Autonomous Database Systems, pages 119–155. Morgan Kaufmann Publishers, 1999.

[26] F. Saltor, M. Castellanos, and M. GarciaSolaco. Suitability of Data Models as Canonical Models for Federated Databases. *SIGMOD Record*, 20(4):44–48, 1991.

[27] A. Sheth and J. Larson. Federated Database Systems for Managing Distributed, heterogeneous and Autonomous Databases. *ACM Computing Surveys*, 22(3):183–226, 1990.

[28] M. Staudt, A. Vaduva, and T. Vetterli. Metadata Management and Data Warehousing. In *International Journal of Cooperative Information Systems*, number ifi-99.04, pages 273–298. World Scientific Publishing Company, 1999.

[29] C. Szyperski. *Component Software; Beyond Object Oriented Programming*. Addison-Wesley, 1998.

[30] J. Veijalainen and R. Popescu-Zeletin. Multidatabase Systems in ISO/OSI Environment. In *Standards in Information Technology and Industrial Control*, pages 83–97. Standards in Information Technology and Industrial Control, 1988.

[31] L. Wang. The Extension of the Object-Relational Model to Facilate the Storage of Roles. Technical Report ISG-02-06, Dublin City University, 2002.

Appendix A

Metadata Interface to Object-Relational Metadata

Name	Data Type	Description
TYPE_NAME	VARCHAR2(30)	Name of the type.
ATTRIBUTES	NUMBER	Number of attributes in the type.
METHODS	NUMBER	Number of methods in the type.
FINAL	VARCHAR2(3)	Indicates whether the type in final.
SUPERTYPE_OWNER	VARCHAR2(30)	Name of the super type owner. Null if it is not a sub type.
SUPERTYPE_NAME	VARCHAR2(30)	Name of the super type NULL if it is not a sub type.

Table A.1: OR-MQL all types view.

Name	Data Type	Description
OWNER	VARCHAR2(30)	Owner of the table.
TABLE_NAME	VARCHAR2(30)	Name of the table.
NESTED	VARCHAR2(3)	Is the table nested?

Table A.2: OR-MQL all tables view.

Name	Data Type	Description
OWNER	VARCHAR2(30)	Owner of the table.
TABLE_NAME	VARCHAR2(30)	Name of the table.
TYPE_NAME	VARCHAR2(50)	The name of the type.
TYPE_OWNER	VARCHAR2(50)	Owner of the type.

Table A.3: OR-MQL all object tables view.

Name	Data Type	Description
OWNER	VARCHAR2(30)	This tells us the owner of the type.
TYPE_NAME	VARCHAR2(30)	Name of the type.
ATTR_NAME	VARCHAR2(30)	Name of the attribute
ATTR_TYPE_MODIFIER	VARCHAR2(30)	Modifier of the type.
ATTR_TYPE_NAME	VARCHAR2(30)	Type of the attribute.
ATTR_TYPE_OWNER	VARCHAR(30)	Owner of the type of the attribute.
LENGTH	NUMBER	Length of the CHAR or maximum for VARCHAR or VARCHAR2 attribute.
SCALE	NUMBER	Scale of the number or decimal attribute.
CHARACTER_SET_NAME	VARCHAR2(44)	The name of the character set.
INHERITED	VARCHAR(5)	Whether the attribute is inherited or not.
COLLECTION	VARCHAR(5)	Whether the attribute is a collection or not.

Table A.4: OR-MQL all type attributes view.

Name	Data Type	Description
OWNER	VARCHAR2(30)	This tells us the owner of the table, view or cluster.
TABLE_NAME	VARCHAR2(30)	Table view or cluster name.
COLUMN_NAME	VARCHAR2(30)	Name of the column.
CHARACTER_SET_NAME	VARCHAR2(30)	Name of the character set.
DATA_TYPE	VARCHAR2(30)	Data type of the column.
DATA_TYPE_MOD	VARCHAR2(3)	Data type modifier of the column.
DATA_TYPE_OWNER	VARCHAR2(30)	Owner of the data type of the column.
DATA_PRECISION	NUMBER	Decimal precision for number data type, binary precision for float data type, null for all other data types
DATA_SCALE	NUMBER	Digits to the right of a decimal point in a number.
NULLABLE	VARCHAR2(1)	Specifies whether a column allows allows NULLs. Value is N if there is NOT NULL constraint on the column or if it is part of the PRIMARY KEY
DATA_DEFAULT	LONG	Default value for the column

Table A.5: OR-MQL all table columns view.

Name	Data Type	Description
OWNER	VARCHAR2(30)	Owner of the table.
CONSTRAINT_NAME	VARCHAR2(30)	Name of the constraint definition.
CONSTRAINT_TYPE	VARCHAR2(1)	Type of constraint definition.
TABLE_NAME	VARCHAR2(30)	Name associated with the table(or view) with constraint definition.
SEARCH_CONDITION	LONG	Text of search condition for a check constraint.
R_OWNER	VARCHAR2(30)	Owner of table referred to in referential constraint.
R_CONSTRAINT_NAME	VARCHAR2(30)	Name of the unique constraint definition for referenced table.
DELETE_RULE	VARCHAR2(9)	Delete rule for a referential constraint(CASCADE or NO ACTION).
STATUS	VARCHAR2(8)	Enforcement status of constraint (ENABLED, DISABLED).
DEFERRABLE	VARCHAR2(14)	Whether the constraint is deferrable.
DEFERRED	VARCHAR2(9)	Whether the constraint is initially deferred.
VALIDATED	VARCHAR2(13)	Whether all data obeys the constraint (VALIDATED or NOT VALIDATED).
RELY	VARCHAR2(4)	Whether an enabled constraint is enforced or unenforced.

Table A.6: OR-MQL all contrainsts view.

Name	Data Type	Description
OWNER	VARCHAR2(30)	Owner of the view.
VIEW_NAME	VARCHAR2(30)	Name of the view.
TEXT_LENGTH	NUMBER	Length of the view text.
TEXT	LONG	View text.
TYPE_TEXT_LENGTH	NUMBER	Length of the type clause of the typed view.
TYPE_TEXT	VARCHAR2(4000)	Type clause of the typed view.
OID_TEXT_LENGTH	NUMBER	Length of the WITH OID clause of the typed view.
OID_TEXT	VARCHAR2(4000)	With OID clause of the typed view.
VIEW_TYPE_OWNER	VARCHAR2(30)	Owner of the type of the view if the view is a typed view.
VIEW_TYPE	VARCHAR2(30)	Type of the view if the view is a typed view.
SUPER_VIEW_NAME	VARCHAR2(30)	Name of the super view.

Table A.7: OR-MQL all object views.

Name	Data Type	Description
OWNER	VARCHAR2(30)	Owner of the view.
VIEW_NAME	VARCHAR2(30)	Name of the view.
TEXT_LENGTH	NUMBER	Length of the view text.
TEXT	LONG	View text.

Table A.8: OR-MQL all relational views metadata view.

Name	Data Type	Description
OWNER	VARCHAR2(30)	Owner of the trigger.
TRIGGER_NAME	VARCHAR2(30)	NOT NULL Name of the trigger.
TRIGGER_TYPE	VARCHAR2(16)	When the trigger fires: BEFORE STATEMENT, BEFORE EACH ROW, BEFORE EVENT, AFTER STATEMENT, AFTER EACH ROW and AFTER STATEMENT
TRIGGERING_EVENT	VARCHAR2(216)	The DML, DDL, or database event that fires the trigger. For a listing of triggering events, see the create trigger statement in Oracle 9i SQL Reference.
TABLE_OWNER	VARCHAR2(30)	Owner of the table on which the trigger is defined.
BASE_OBJECT_TYPE	VARCHAR2(16)	The base object on which the trigger is defined: TABLE, VIEW, SCHEMA or DATABASE.
TABLE_NAME	VARCHAR2(30)	If the base object type of the trigger is SCHEMA or DATABASE then this column is NULL; If the base object type is TABLE or VIEW, this column indicates the table view name on which the trigger is defined.
COLUMN_NAME	VARCHAR2(30)	Name of the nested table column (if nested table) or else NULL.
REFERENCING_NAMES	VARCHAR2(87)	Names for referencing OLD and NEW columns from within the trigger
WHEN_CLAUSE	VARCHAR2(4000)	Must evaluate to TRUE for TRIGGER_BODY to execute.
STATUS	VARCHAR2(8)	When the trigger enabled (ENABLED|DISABLED).
DESCRIPTION	VARCHAR2(4000)	Trigger description.
ACTION_TYPE	VARCHAR2(11)	The action type of the trigger (CALL or PL/SQL)
TRIGGER_BODY	LONG	Statements executed by the trigger when it fires

Table A.9: OR-MQL all triggers view.

Name	Data Type	Description
OWNER	VARCHAR2(30)	Owner of the type.
TYPE_NAME	VARCHAR2(30)	Name of the type.
METHOD_NAME	VARCHAR2(30)	Name of the method.
METHOD_TYPE	VARCHAR2(6)	Type of the method.
PARAMETERS	NUMBER	Number of parameters with the method.
RESULTS	NUMBER	Number of results returned by the method.
FINAL	VARCHAR2(3)	(YES|NO) indicates whether the method is final.
INSTANTIABLE	VARCHAR2(3)	(YES|NO) Indicates whether the method is instantiable.
OVERRIDING	VARCHAR2(3)	(YES|NO) Indicates whether the method is over riding a sub type method.
INHERITED	VARCHAR2(3)	(YES|NO) Whether the method is inherited from a super type.

Table A.10: OR-MQL all type methods view.

Name	Data Type	Description
OWNER	VARCHAR2(30)	Owner of the type.
TYPE_NAME	VARCHAR2(30)	Name of the type.
METHOD_NAME	VARCHAR2(30)	Name of the method.
METHOD_NO	NUMBER	For an overloaded method, a number distinguishing this method from others of the same. Do not confuse the number with the object ID.
PARAM_NAME	VARCHAR2(30)	Name of the parameter
PARAM_NO	NUMBER	Parameter number(position).
PARAM_MODE	VARCHAR2(6)	Mode of the parameter(IN, OUT, IN/OUT)
PARAM_TYPE_MOD	VARCHAR2(7)	Whether this parameter is REF to another object
PARAM_TYPE_OWNER	VARCHAR2(30)	Owner of the type of the parameter
PARAM_TYPE_NAME	VARCHAR2(30)	Name of the type of the parameter
CHARACTER_SETNAME	VARCHAR2(44)	Name of the Character set

Table A.11: OR-MQL all method parameters view.

Name	Data Type	Description
OWNER	VARCHAR2(30)	Owner of the method type.
TYPE_NAME	VARCHAR2(30)	Name of the method type.
METHOD_NAME	VARCHAR2(30)	Name of the method.
CHARACTER_SET_NAME	VARCHAR2(30)	Character set.
METHOD_NO	NUMBER	For an overloaded method a number that distinguishes this method from the others.
RESULT_TYPE_MOD	VARCHAR2(7)	Whether the parameter is a REF to another object.
RESULT_TYPE_OWNER	VARCHAR2(30)	Owner of the return type.
RESULT_TYPE_NAME	VARCHAR2(30)	Name of the return type

Table A.12: OR-MQL all method results view.

Column Name	Data Type	Description
TYPEA	VARCHAR2(50)	The first type in the relationship
TYPEB	VARCHAR2(50)	The second type in the relationship.
CARDINALITY	VARCHAR2(15)	The cardinality between the two types.
REFTABLE_NUM	NUMBER	Number to reference cardinality definition table.

Table A.13: OR-MQL high level metadata view.

Column Name	Data Type	Description
OWNER	VARCHAR2(50)	The owner of the schema where the types are defined.
TYPEA	VARCHAR2(50)	The first type in the cardinality relationship
TYPEB	VARCHAR2(50)	The second type in the cardinality relationship
DEPTA	VARCHAR2(8)	Whether or not the type of the attribute A was found deep or shallow.
COLLECTION_TYPEA	VARCHAR2(50)	Name of the collection type for A if the attribute is a collection
CARDINALITY	VARCHAR2(10)	The cardinality relationship of the type.
COLL_TYPEB	VARCHAR2(12)	Werther the collection of type B is a NESTEDTABLE or VARRAY.
DEPTB	VARCHAR2(8)	Whether or not the type of the attribute B was found to be deep or shallow.
COLLECTION_TYPEB	VARCHAR2(50)	multimedia

Table A.14: OR-MQL low level metadata view.

Index	TypeA Many-type	Cardinality	TypeB-Many_type
1		one-one	
2		zero-one	
3		one-zero	
4	SHALLOW	many-zero	
5	NESTED	many-zero	
6	VARRAY	many-zero	
7		zero-many	SHALLOW
8		zero-many	NESTED
9		zero-many	VARRAY
10		one-many	SHALLOW
11		one-many	NESTED
12		one-many	VARRAY
13	SHALLOW	many-one	
14	NESTED	many-one	
15	VARRAY	many-one	
16	SHALLOW	many(N)-many(M)	SHALLOW
17	SHALLOW	many(N)-many(M)	NESTED
18	SHALLOW	many(N)-many(M)	VARRAY
19	NESTED	many(N)-many(M)	SHALLOW
20	NESTED	many(N)-many(M)	NESTED
21	NESTED	many(N)-many(M)	VARRAY
22	VARRAY	many(N)-many(M)	SHALLOW
23	VARRAY	many(N)-many(M)	NESTED
24	VARRAY	many(N)-many(M)	VARRAY

Table A.15: OR-MQL cardinality of Objects Lookup Table

Name	Type	Description
PARENT	VARCHAR2(50)	Parent type in the inheritance relationship.
CHILD	VARCHAR2(50)	Child type in the inheritance relationship.
INSTANTIABLE	VARCHAR2(5)	Whether or not this type is instantiable (true||false).
FINAL	VARCHAR2(5)	Whether or not this type is final (true||false).

Table A.16: OR-MQL Inheritance Metadata.

Name	Type	Description
UPPER_BOUND	VARCHAR2(30)	The maximum size.
ELEM_TYPE_MOD	NUMBER	The modifier of the collection.
ELEM_TYPE_OWNER	VARCHAR2(30)	Owner of the type upon which the collection is based.
ELEM_TYPE_NAME	VARCHAR2(30)	Name of the element type in the string.
LENGHT	NUMBER	Maximum lenght of character string elements.
PRECISION	NUMBER	Decimal point precision of a number
CHARACTER_SET_NAME	VARCHAR2(44)	Name of the character set.

Table A.17: OR-MQL varray collection metadata.

Name	Type	Description
ELEM_TYPE_MOD	NUMBER	The modifier of the collection.
ELEM_TYPE_OWNER	VARCHAR2(30)	Owner of the type upon which the collection is based.
ELEM_TYPE_NAME	VARCHAR2(30)	Name of the element type in the string.
LENGHT	NUMBER	Maximum lenght of character string elements.
PRECISION	NUMBER	Decimal point precision of a number
CHARACTER_SET_NAME	VARCHAR2(44)	Name of the character set.

Table A.18: OR-MQL nested table metadata.

Appendix B

Object-Relational Metadata Query Language (OR-MQL)

The keyword "names" can be placed after any of the selection keywords so as to return the names of the entity. If the keyword "names" is omitted the details of the entities will be returned. If the owner keyword is not specified the queries will be implemented on the account of the user who is logged in.

1. [select] schema names // returns the list of all database schema names in the schema repository
2. [select] types [names] ["owner"]
3. [select] cardinality [owner][[.typea[.typeb]] or. schema_name]
4. [select] lowlevelcardinality [owner][[.typea[.typeb]] or .schema_name]
5. [select] carreftable // reference metadata for cardinality relationships
6. [select] inheritance ["owner"]["type"] or ["owner"].["schema"] // asking a for a name from inheritance information will not return useful information.
7. [select] attributes [names] ["owner"]["type"]
8. [select] methods [names] ["owner"]["type"]
9. –omitting names keyword and select keyword–
10. parameters ["owner"]["type"]["method"]
11. results ["owner"]["type"]["method"]

Tables and Views

1. tables ["owner"]
2. ObjectTables ["owner"]
3. relationalViews ["owner"]
4. objectViews ["owner"]
5. triggers ["owner"] or ["owner"][".view"] or ["owner"]["."table"]
6. constraints ["owner"][".view"] or ["owner"]["."table"]
7. collection_varrays ["owner"]
8. collection_nestedT ["owner"]

Role Metadata

1. role_subSchema
2. root ["role_subSchema"]
3. rootAttributes ["role_subSchema"]["root"]
4. rootMethods ["role_subSchema"]["root"]
5. rootRoles ["role_subSchema"]["root"]
6. roleAttributes ["role_subSchema"]["root"]["role"]
7. roleMethods ["role_subSchema"]["root"]["role"]

Appendix C

Role Extensions

In this section access to role metadata is presented. Explanations of the fields extracted from the underlying views is provided together with descriptions of the SQL statements.

The role system modelled is implemented as five types. The structure of the types is included in the extensions and extra fields are also provided. This section describes the fields retrieved from the role metadata views, the SQL statements used to retrieve them and the schema query language definitions.

The role extensions are illustrated with a human resource examples and illustrations. We illustrate a root role person and two assumable roles student and employee and their respective characteristics.

Root Roles

Table C.1 describes the fields retrieved from the `SYS_ROLEVIEW_OBJTAB`. This table describes the root role in our role system. The root roles basic structure is defined with methods and attributes. This basic structure can be extended to include new behaviour and characteristics by assuming new roles. The roles that can be assumed by the root roles are stored in the variable `ROLEVIEW_LIST`.

The fields from *table C.1* are retrieved using four `select` statements, *listing C.1, listing C.2, listing C.3,* and *listing C.4.*

When discerning the structure of an extended O-R schema and examining the structure of root roles it is necessary to retrieve the metadata in *table C.1.* The query in *listing* C.5 illustrates the `root_role` schema query. When querying multiple role sub-schemas the `role_subSchema` variable is used to distinguish which sub-schema to query. Including

Name	Data Type	Description
ROOT_NAME	VARCHAR2(30)	Name of root role.
ROOT_OID	NUMBER	ID and primary key of role.
ORACLE_TYPE	REF	The type of the role.
SUPERTYPE	VARCHAR2(30)	The supertype of the role.
ROLEVIEW_LIST	NESTED TABLE	List of the roles supported by the root role.
ATTRS_LIST	NESTED_TABLE	List of root roles attributes.
METHOD_LIST	NESTED_TABLE	List of root roles methods.
NUM_ATTRS	NUMBER	The number of attributes.
NUM_METHODS	NUMBER	The number of methods.
NUM_ROLES	NUMBER	The number of roles which can be assumed.

Table C.1: The fields retrieved for the root role.

```
SELECT COUNT(*) num_attrs
FROM sys_root_objtab c, TABLE( c.root_attribute) (+) p
GROUP BY c.root_name, c.root_oid;
```

Example C.1: The NUM-ATTRS field.

```
SELECT COUNT(*) num_methods
FROM sys_root_objtab c, TABLE( c.root_method) (+) p
GROUP BY c.root_name, c.root_oid;
```

Example C.2: The NUM-METHODS field.

```
SELECT COUNT(*) num_assumable
FROM sys_root_objtab c, TABLE( c.roleviewlist) (+) p
GROUP BY c.root_name, c.root_oid;
```

Example C.3: The NUM-ROLES field.

```
SELECT root_name,root_oid, oracletyperef,
supertype, roleviewlist, root_attribute, root_method
FROM sys_root_objtab
```

Example C.4: Direct select from SYS-ROOT-OBJTAB.

the `names` keyword means only the names of the `root_roles` will be returned, omitting names will return the details in *table C.1.*

```
[select] root [names] OWNER.(ROLE.SUBSCHEMA)
```

Example C.5: Schema query for root-roles.

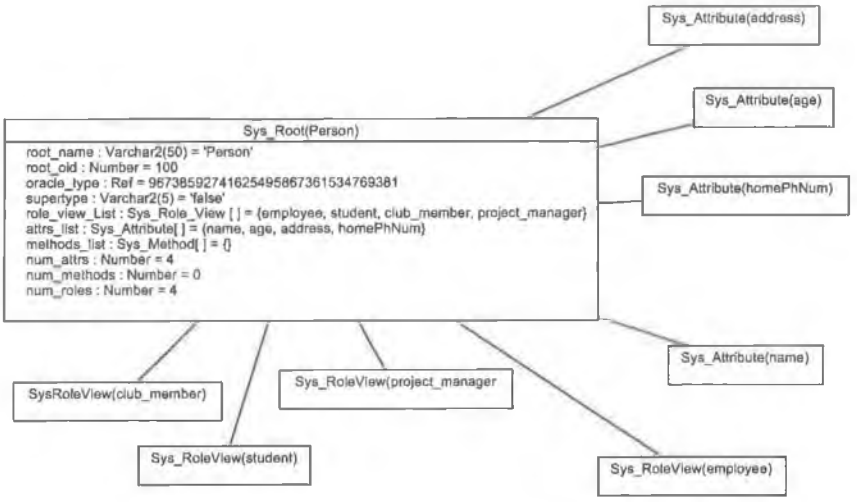

Figure C.1: Root role metadata.

Name	Data Type	Description
ROLE_NAME	VARCHAR2(30)	Name of the assumable role.
ROLE_VIEW_ID	NUMBER	The ID of the assumable role.
NUM_ATTRS	NUMBER	The number of attributes.
NUM_METHODS	NUMBER	The number of methods.
IS_MULTIPLE	VARCHAR2(3)	Is it a multiple role?
ROOT_OID	NUMBER	The root roles identification
ROLEVIEW_METHODS	NESTED_TABLE	List of roles attributes.
ROLEVIEW_ATTRIBUTES	NESTED_TABLE	List of roles methods.

Table C.2: Fields for assumable roles.

A human resource example is used to illustrate role metadata. The example comprises a root role, its structure and the roles it can assume. After extracting the fields from *table C.1* the structure is illustrated in *figure C.1*. This roles basic structure is described by four attributes `name`, `age`, `address` and `homePhNum`. It can assume for roles `employee`, `student`, `club_member` and `project_manager`.

Assumable Roles

The `SYS_ROLEVIEW_OBJTAB` table holds all the assumable roles for the root roles. The retrieved fields described in *table C.2* provide the information for the metadata interface. The fields in this table are taken from `SYS_ROLEVIEW_OBJTAB` table and `SYS_ROOT_OBJTAB` table. Three *select* statements are needed to extract the fields in *table C.2; listing C.6, listing C.7* and *listing C.8.*

When discerning the structure of an extended O-R schema and examining the structure of assumable roles it is necessary to retrieve the metadata in *table C.2.* The query in *listing*

```
SELECT COUNT(*) num_attrs
FROM sys_roleview_objtab c, TABLE( c.roleview_attribute) (+) p
GROUP BY c.roleview_name, c.roleview_id;
```

Example C.6: The NUM-ATTRS field.

```
SELECT COUNT(*) num_methods
FROM sys_roleview_objtab c, TABLE( c.roleview_method) (+) p
GROUP BY c.roleview_name, c.roleview_id;
```

Example C.7: The NUM-METHODS field.

```
SELECT roleview_name, roleview_rid, root
ismulitple, roleview_attribute, roleview_method
FROM sys_roleview_objtab
```

Example C.8: Direct select from SYS-ROLEVIEW-OBJTAB.

C.9 illustrates the `root_role` schema query. The `root` variable is used to distinguish a root-role. Including the `names` keyword means only the names of the assumable roles will be returned, omitting names will return the details in *table C.1*.

```
[select] rootRoles [names] OWNER.ROLE_SUBSCHEMA.ROOT
```

Example C.9: Schema query for roles.

The root role of the human resource example is `Person`. The structure of a `Person` can change as he receives education and eventually gets a job. As these changes occur, the structure of the `Person` role also changes as it assumes new roles. *Figure C.2* illustrates the fields extracted from *table C.2* for the human resource example. There are four assumable roles, `employee`, `student`, `club_member` and `project manager`. Two are shown in detail, `employee` and `student`. The basic structure of a `student` has `college`, `faculty` and `modules` attributes. The basic structure of an `employee` has a `salary`,`work phone number` and `job` description. The combined structure of the root role and the assumed roles give the state of the whole entity at any moment in time.

Role/Root Attributes

The SYS_ATTRIBUTE_OBJTAB holds the metadata about the attributes of the root roles and the assumable roles. *Tables* C.3 *and* C.4 describe the fields retrieved for use

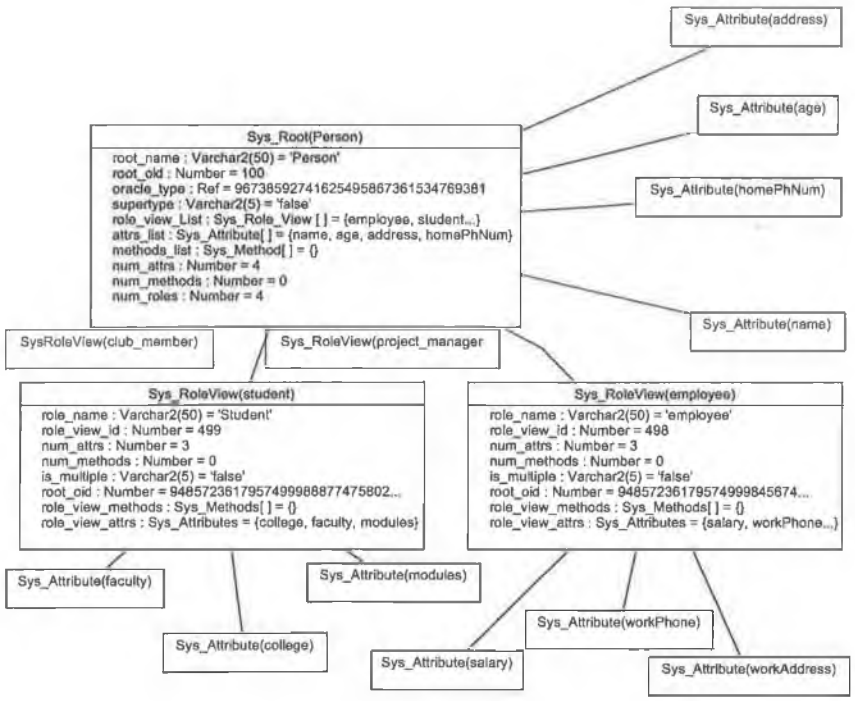

Figure C.2: Roleview metadata.

Name	Data Type	Description
ROOT_ATTR_NAME	VARCHAR2(30)	The name of the attribute.
ROOT_ATTR_ID	NUMBER	The id of the attribute and the primary key.
ROOT_ORACLE_TYPE_REF	VARCHAR2(30)	Reference to the base type of the attribute.
ROOT_OID	NUMBER	The object identifier of the owning role.

Table C.3: Root attribute fields.

in the metadata interface. To retrieve the fields for attributes, two select statements are used, *listing C.10* and *listing C.11.*

```
SELECT p.attr_name,p.attr_id
p.oracletype_attrref, c.root_oid
FROM sys_root_objtab c,
TABLE(c.root_attribute) p;
```

Example C.10: The attributes fields select statement (Root table).

When discerning the structure of an extended O-R schema and examining the structure of roles and root-roles it is necessary to deduce the structure of the attributes. The query in *listing* C.12 illustrates the root/role attribute schema query. Including the names keyword means only the names of the attributes will be returned, omitting names will return the

Name	Data Type	Description
ROLE_ATTR_NAME	VARCHAR2(30)	The name of the attribute.
ROLE_ATTR_ID	NUMBER	The ID of the attribute and the primary key.
ROLE_ORACLE_TYPE_REF	VARCHAR2(30)	Reference to the base type of the attribute.
ROLEVIEW_RID	NUMBER	The object identifier of the owning role.

Table C.4: Role attribute fields.

```
SELECT p.attr_name,p.attr_id
p.oracletype_attrref, c.roleview_rid
FROM sys_roleview_objtab c,
TABLE(c.roleview_attribute) p;
```

Example C.11: The attributes fields select statement (Assumable Role Table).

details in *table C.3* or *C.4* depending on the query. Specifying the `role` variable will return the attributes for the role which belongs to the named root, otherwise the attributes of the root will be returned.

```
[select] Attributes [names] OWNER.ROLESUBSCHEMA.ROOT[.ROLE]
```

Example C.12: Schema query for roles or root roles attributes.

Attributes are what describe the current state of a role or a root role. For a root role that has assumed new roles the combined attributes fully describe the state of the entity at that moment. *Figure C.3* includes the metadata from *table C.4* and *C.3* which illustrates attribute metadata for this example.

Root/Role Methods

The SYS_METHOD_OBJTAB holds the metadata about the methods of the root roles and the assumable roles. *Tables C.5* and *C.6* describe the fields retrieved for use in the metadata interface. To retrieve the fields for methods two select statements are used, listing C.13 and listing C.14.

When discerning the structure of an extended O-R schema and examining the structure of roles and root-roles it is necessary to deduce the structure of the methods. The query in *listing* C.15 illustrates the root/role method schema query (square brackets '[]' indicate an optional parameter). Including the names parameter means only the names of the attributes will be returned, omitting names will return the details in *table C.5* or *table C.6*

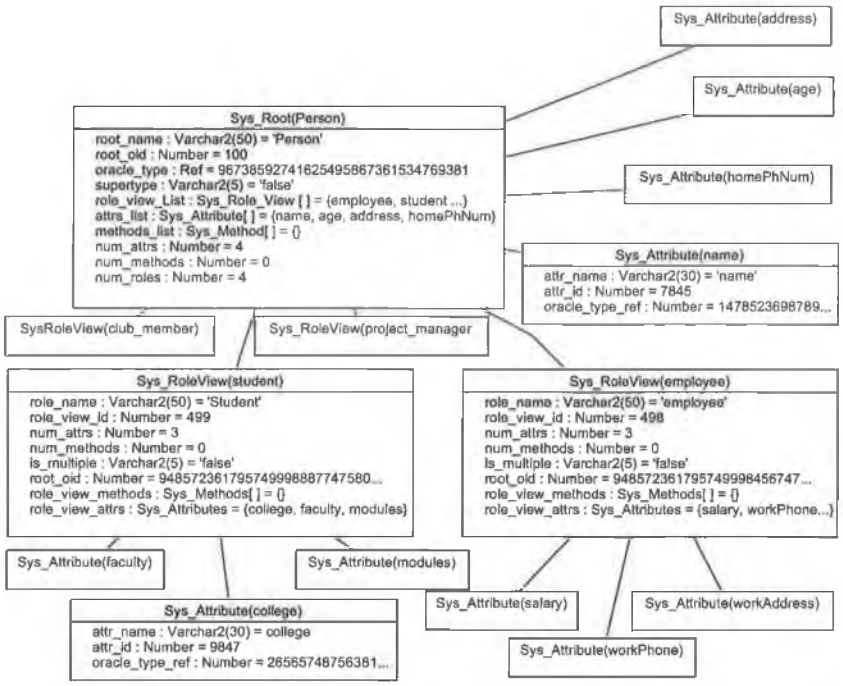

Figure C.3: Role attributes metadata.

Name	Data Type	Description
ROOT_METHOD_NAME	VARCHAR2(30)	The name of the method.
ROOT_METHOD_ID	NUMBER	The ID of the method and the primary key.
ROOT_ORACLE_TYPE_REF	VARCHAR2(30)	Reference to the base type of the method.
ROOT_OID	NUMBER	The object identifier of the owning role.

Table C.5: The root method fields.

Name	Data Type	Description
ROLE_METHOD_NAME	VARCHAR2(30)	The name of the method.
ROLE_METHOD_ID	NUMBER	The ID of the method and the primary key.
ROLE_ORACLE_TYPE_REF	VARCHAR2(30)	Reference to the base type of the method.
ROLEVIEW_RID	NUMBER	The object identifier of the owning role.

Table C.6: The role method fields.

```
SELECT p.method_name,p.method_id
p.oracletype_methodref, c.root_oid
FROM sys_root_objtab c,
TABLE(c.root_method) p;
```

Example C.13: The root method select statement.

```
SELECT p.method_name,p.method_id
p.oracletype_methodref, c.roleview_rid
FROM sys_roleview_objtab c,
TABLE(c.roleview_method) p;
```

Example C.14: The role method select statement.

depending on the query. Specifying the `[.role]` attribute will return the methods for the role which belongs to a particular root (`.root`), otherwise the methods of the root will be returned.

```
Methods [NAMES] OWNER.ROLE_SUBSCHEMA.ROOT[.ROLE]
```

Example C.15: Schema query for roles or root-roles methods.

Role Sub-Schema

Table C.7 describes the fields retrieved from SYS_SUBSCHEMA_OBJTAB table for use in the metadata interface. *Listing C.16* is the SQL statement that is used to retrieve the fields for the sub-schema.

```
SELECT schema_name, root
FROM sys_subschema_objtab
```

Example C.16: The SUBS-CHEMA select statement.

When discerning the structure of an extended O-R schema and examining the structure of a sub-schema the metadata in *table C.7* needs to be retrieved. The query in *listing C.17* illustrates the role sub-schema query (square brackets '[]' indicate an optional parameter).

```
[select] role_subSchema
```

Example C.17: Schema query for role-subSchema.

Name	Data Type	Description
SCHEMA_NAME	VARCHAR2(30)	The name of the schema.
ROOT	REF	Reference to type SYS_ROOT.

Table C.7: The SUB_SCHEMA fields.

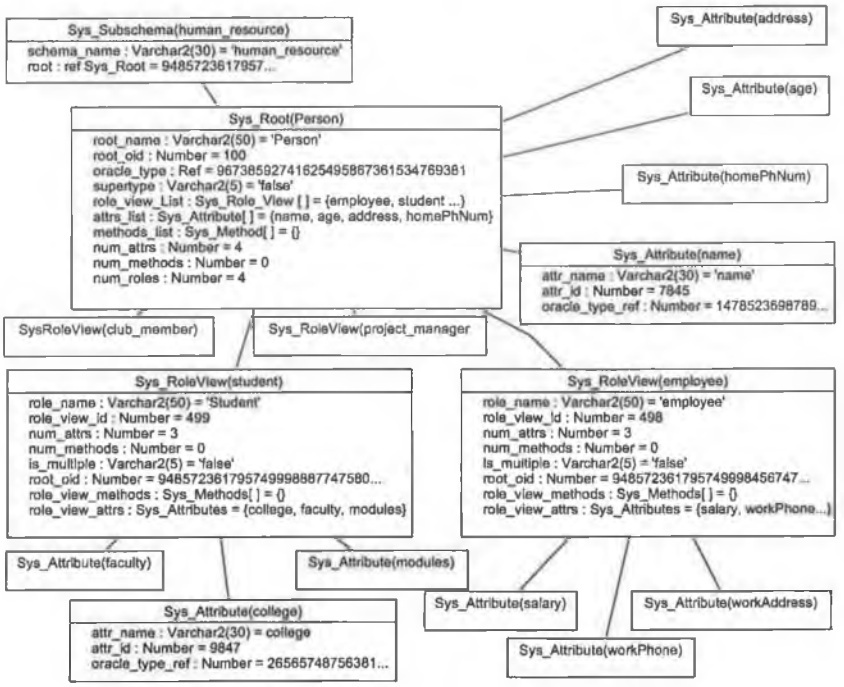

Figure C.4: Sub-schema metadata.

Subschema points to one root role. For the human resource example the root role is Person, as everything in the sub_schema must be of type Person. This is illustrated in figure C.4.