Query Processing and Bulk Loading in Object Oriented Database Systems

A Project Report

Submitted in partial fulfilment of the

requirements for the Degree of

Master of Engineering

in

Computer Science and Engineering

by

M. Krishna

Department of Computer Science and Automation

Indian Institute of Science

Bangalore – 560 012

June 1996

Contents

Acknowledgements

I would like to thank my project guide, *Dr. Jayant Haritsa*, for his constant support and encouragement during the course of the project. His attention to detail, patience, understanding, and professionalism provided the inspiration to perform just that wee bit better everytime. The Database Systems Lab with its abundant supply of books and journals, the flexi-times, the seminars, the Pune trip and the numerous hours of discussions will never be forgotten.

I would like to specially thank Arun, who as a member of the DIAS project team, provided me with suggestions and constructive criticism which served to streamline ideas. His willingness to help at all times speaks volumes about him, as a person. I would also like to thank my other lab mates - Bhuvana, Binto and Ramesh for their help and enjoyable company. I would also like to thank the members of the Parasitic Work Bench team at Texas Instruments, India, for their valuable inputs.

I take this opportunity to thank my advisor during the first three years of my stay in IISc, *Prof. Priti Shankar*, who helped me take several important academic decisions.

Cheers to Ajax, Bubs, Bull, Cheens, Desp, Hins, Nals, Rajba, and Suthi who have made my stay in IISc a memorable one. Kudos to Anj, Ananth, Murls, Rishi, Rev, Sand, and Shyam for their unparalleled care, understanding and encouragement which has always helped me overcome the most difficult of times and given me memories that will always be cherished.

Finally, I would like to thank my grandparents, my parents and Laks for their love and encouragement, and without whom I would not be what I am.

Abstract

Object orientation is a paradigm that has revolutionised computing. Its influence on the area of database management systems is evident from the quantum of research in the area of Object Oriented Database Management Systems (OODBMSs) in the past decade. The OODBMS represents the convergence of many research areas that include artificial intelligence, programming languages, software engineering and traditional database systems. They overcome the modeling inadequacies of traditional databases at the same time reducing the "impedance mismatch" that existed between programming languages and database languages.

Query processing in object oriented databases is still in its infancy and the advantages gained by powerful modeling techniques cannot be exploited unless the system incorporates powerful query capabilities as well. In this project, we focus on the issue of query evaluation in object oriented databases. We have developed a query model that was used extensively in the construction of the query processing system. The query processing engine was integrated with DIAS, an object oriented database system storing parasitic information, as part of an ongoing database project sponsored by Texas Instruments. The DIAS project, thus, provided a test bed to test the capabilities of the query processing system. We have attempted to evaluate the system on real data provided to us by Texas Instruments.

We have also addressed the problem of bulkloading in object oriented databases as a complementary issue. The DIAS bulkloader built as a result of this evaluation is now operational and serves to move legacy data from files into the database.

Chapter 1

Introduction

1.1 Motivation

The late eighties and early nineties saw the emergence of problem domains dealing with inherently complex data. Application domains such as Computer Aided Design (CAD) for VLSI, Geographical information systems (GIS), and Biodiversity database systems are some of the many application domains which process inherently complex data. Relational database systems were initially used for data management in these applications but proved inadequate due to their lack of modeling power. Complex systems that are inherently hierarchical could not be captured naturally because the relational model represented information as records. Hence, it became necessary to flatten the hierarchy in order to map these systems into the relational model. This gave rise to inefficiencies in data management and subsequent performance problems. A complex entity had to be represented as a series of records and subsequently a series of algebraic operations had to be performed on the series of records to retrieve the object. Though relational systems had extremely efficient and well developed mechanisms for data retrieval, they could not be exploited due to the unnatural manner in which the complex entities were represented. The solution did not seem adequate and the search for innovative systems to handle complex problem domains resulted in the emergence of object oriented database systems.

Object orientation refers to the concept of representing entities with the preservation of associated semantics. This is accomplished by strongly binding the state and behaviour of a real-world or abstract entity into a single structure called an *object*. Object orientation provides various features that aid system development – *Abstraction* facilitates focusing on the essential aspects of an entity while ignoring other properties. *Encapsulation* hides the internal implementation details of an object from other objects. *Polymorphism* allows for multiple definitions of an operation. *Inheritance* allows sharing of state and behaviour between objects. *Aggregation* allows the containment of one object within another. Object oriented database systems, in addition to these features, provide *persistence* that enables the type and state of an object to transcend time and space. These features provide a powerful set of tools which can be used to model complex domains in a natural manner.

Thus, the object oriented database model addresses the issue of modeling of complex problem domains. However, they still lack *query processing* capabilities that compare in efficacy with its relational counterpart. This is because object oriented query processing throws up several new challenges as compared to relational database systems. The rich variety of added base types, parameterised types and user defined types introduce a heterogeneous environment necessitating specialised treatment according to type. Also, new types may have to be created at runtime since the result of a query may not belong to the existing classes that constitute the schema. Object oriented systems, in addition to associative access (as in relational systems), also provide for navigational access (i.e., pointer chasing). This leads to implicit joins in aggregation hierarchies requiring specialised techniques for efficient processing. Moreover, the scope of a query may include in addition to the referenced class, all classes belonging to the inheritance hierarchy rooted at that class. The possibility of method definitions being used to implement query operators leads to enrichment in the number of operators in the query language. However, the inability to accurately estimate the cost of methods makes the task of query optimization much more difficult. Finally, the presence of complex objects in object oriented systems requires more sophisticated access methods than those present in relational systems. Thus, query processing in such systems has been a focus of research in the past decade.

1.2 Goals and contributions

We first look at issues that need to be addressed during the development of an object oriented database system. The database designer needs to address three major issues – *object representation*, *object population* and *object manipulation*(as shown in Figure 1.1). Object representation refers to the problem of capturing the semantics of a problem domain with the help of the object model. Object population refers to populating the database and object manipulation refers to the query processing capabilities with the help of which users manipulate data. The object representation issue has been addressed and various methodologies developed [Rumb+91, Booc94]. However, object oriented systems lack established techniques for object population and object manipulation.

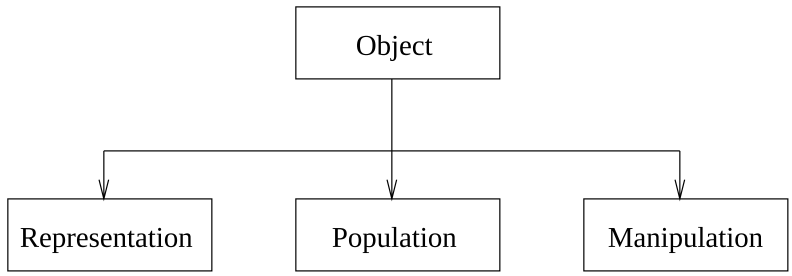


Figure 1.1: Database system issues

Thus, the aim of this project is to get a better understanding of the problems of query processing in object oriented database systems and to address some of the issues in building an object oriented query processing engine. Another aspect of the project is to address the problem of *bulkloading* in object oriented database systems. Bulkloading refers to the mass transfer of data from legacy files to the object oriented database and addresses the issue of data population.

As part of the project, we have evaluated the features of query processing systems in currently available object oriented database systems and identified issues that need to be addressed in building a query engine. We defined a query model to capture the semantics of operations performed during the processing of a query. The query model incorporated some features of an available query model [Kim89] and added new ones to capture additional semantic information of the domain such as simple associations in an object data model. We designed a generic query processing engine based on this query model that offers features such as ad-hoc declarative querying and schema independence. The design also incorporates specialised access mechanisms for the object oriented domain to provide efficient query processing.

In order to test the efficacy of the design, the query processing engine was implemented and integrated into an ongoing object oriented database project called DIAS (Database for interconnect analysis Systems). The DIAS database project is a joint venture between our institute and Texas Instruments, Inc. and aims at providing an object oriented solution to the problem of management of large volumes of parasitic data generated during the VLSI design process. The DIAS system, thus, provided a test bed for the query processing engine which in turn provided DIAS with a vehicle for efficient data management.

We have also dealt with the problem of bulkloading in an object oriented database. We have identified issues in bulkloading in an object oriented system and addressed them in the construction of a bulkloader for the DIAS database system. The DIAS bulkloader is now operational and serves to move large amounts of legacy data into the DIAS system.

1.3 Interface to DIAS

The DIAS database system is an object oriented database system that implements a parasitic data model [Arun96]. The DIAS system is composed of several components that include the dias kernel and the query processor. The bulkloader adds functionality to the system by providing mass-population capabilities. Figure 1.2 shows the architecture of the DIAS system. The contributions of our effort to the DIAS database project is shown with shaded boxes and amount to around 25000 lines of C++ code.

The DIAS database system uses the SHORE [Zwil+94] database server which provides object management and transaction related features such as concurrency control and recovery. The DIAS parasitic data model is implemented as part of the DIAS kernel. The DIAS kernel also implements access methods to provide associative access mechanisms for efficient retrieval and a schema manager that provides schema independence. The

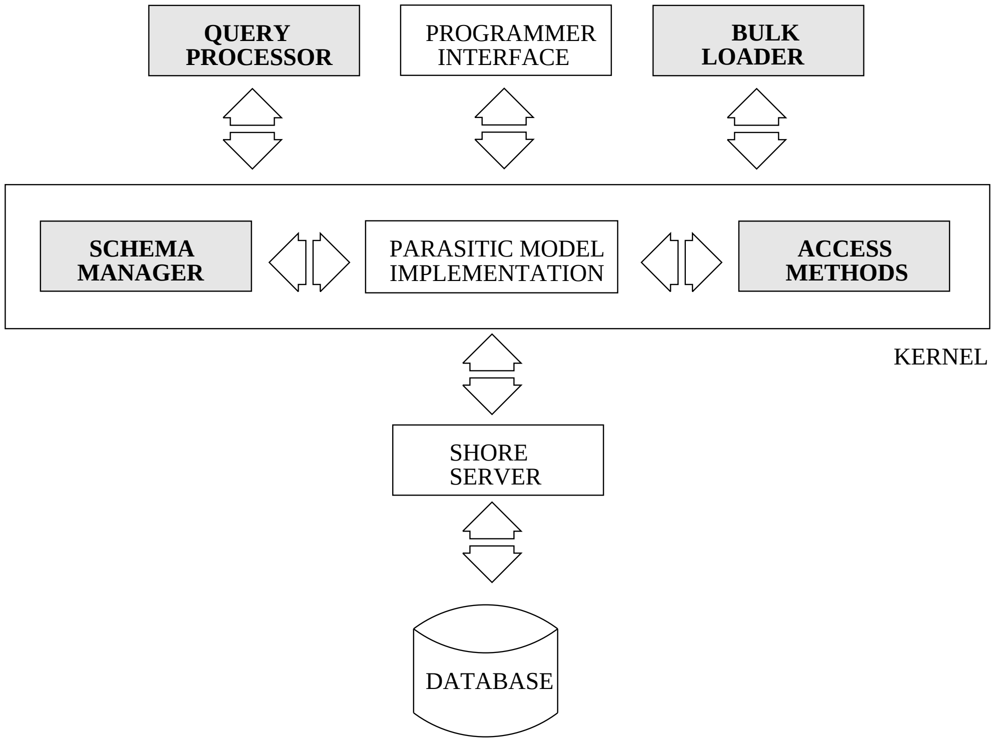


Figure 1.2: DIAS System architecture

query processor is built on top of the DIAS kernel and provides a declarative interface to query processing. DIAS also provides a programmer's interface called the Application programmer's interface (API) which provides an interface from which applications can be built. A bulk loader has also been built above the DIAS kernel which aids in moving legacy data from files into the database and also provides the facility of a textual dump. In the following section, we briefly introduce the query processing components and the bulkloader.

1.4 Query Processor

In this section, we present a brief overview of the query processor that is a core component of the query processing engine. The query processor was built using the framework provided by the query model incorporates common requirements of object oriented database systems (for example, a declarative query language). The architecture of the query processor is shown in figure 1.3.

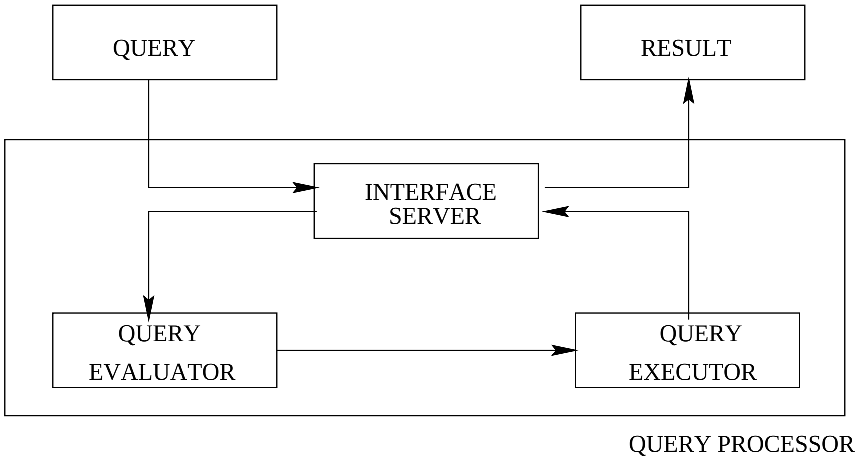


Figure 1.3: Query processor architecture

The query processor consists of an *interface server* which is essentially a manager module and interfaces with the external world. It takes in a query description by providing a query language interface. The interface server passes the language description to the query evaluator which translates the query into an parse tree through a series of operations and does a series of checks to ensure correctness of query. The evaluated query in its internal form is then used by the query executor to execute the query. The query executor generates the sequence in which the desired operations need to be performed, fetches the objects, evaluates them against the given predicate and passes them onto other operators for further evaluation. The query executor also includes runtime optimisations to provide efficient query processing.

1.5 Bulkloader

Bulkloading is the process of mass-populating a database with collected data. The bulkloading populates the DIAS database system with parasitic information available in a file based format. It is also capable of performing the reverse operation of dumping data from the database into the file in the specified format. The architecture of the DIAS bulkloader is shown in Figure 1.4.

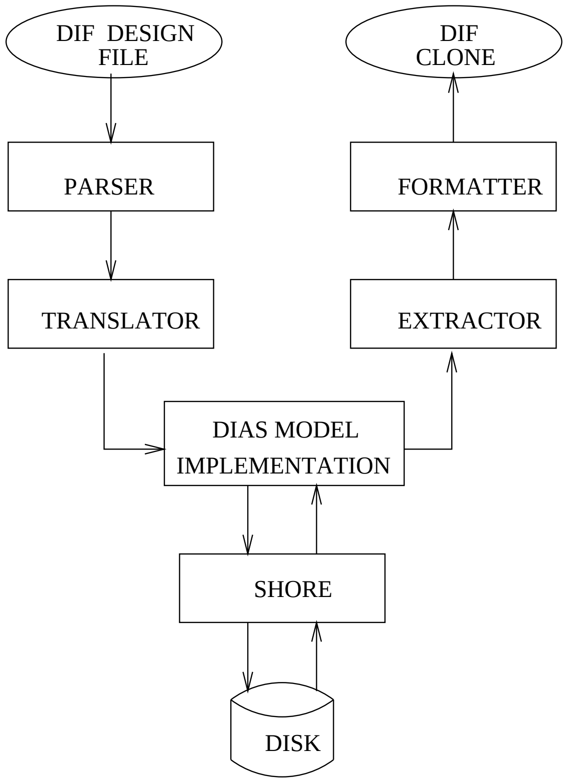


Figure 1.4: Bulkloader Architecture

The files containing the data are organised by a specified grammar and hence the bulkloader needs to understand the grammar. The parser extracts the information from the file and builds main memory data structures. The translator uses these structures to resolve object relationships and create the corresponding persistent objects with calls to the data model interface. The reverse process of the textual dump involves the persistent objects being read by the extractor from the model interface. The semantic information extracted is used to convert the relationships into a form compatible with the file format. The formatter reads this information and recreates the text file by incorporating grammar tokens and representing relationships in the file format. In both the cases, the model interface makes calls to the SHORE server which does the actual data manipulation on disk.

1.6 Schema Manager

The schema manager module of the query processing system is responsible for the management of the schema. It stores information about the classes and relationships in the schema. Semantic information linking an attribute of a class to the set of methods that operate on it are also stored. The schema manager uses this information to help the query processor in evaluating ad-hoc queries by providing the names of methods that have to be dynamically bound in order to retrieve specified objects from the database. It also aids the semantic evaluation stage of the query processor by providing schema information.

1.7 Access Methods

The access methods module supplements the access methods provided by SHORE with mechanisms that provide efficient data retrieval for queries directed at specific types of relationships. It provides the nested index and path index for queries directed at aggregation hierarchies and the hcC tree for queries directed at inheritance hierarchies. The B^+ tree provided by SHORE is used to index objects based on simple values. It is also used to index collections of objects. The access methods are invoked by the query executor using information provided by the schema manager.

1.8 Organisation of the Report

The rest of the thesis is organised as follows. Chapter 2 presents important aspects of query processing in representative object oriented database systems. The basic concepts of object orientation and issues in query processing are discussed in chapter 3. The query model is presented in chapter 4. Chapter 5 presents a survey of access techniques used for efficient retrieval in object oriented database systems. The design of the query processing system is discussed in chapter 6 and chapter 7 describes the implementation details. The design and implementation of the DIAS bulkloader is presented in chapter 8. We conclude in chapter 9 with the summary of our work and suggest possible avenues for future work.

Chapter 2

Related Work

The next generation database systems that have been prototyped use a variety of mechanisms to provide good performance. In this chapter, we discuss some of these systems and highlight their features. Object oriented database systems can be broadly classified into two types – *monolithic* and *extensible*. Monolithic object oriented database systems (simply, monolithic database systems) provide a complete end-user database management system, in that, they have a predefined internal structure. These systems aim at providing a data management solution that is independent of any particular problem domain. Monolithic database systems are also referred to as *database programming languages* in the literature. Extensible database systems, on the other hand, believe in providing a basic set of tools with the help of which a database system designer can build a database customised to a particular domain. Extensible database systems are sometimes called *database system generators*.

2.1 Monolithic database systems

This architecture represents database systems that extend existing programming languages, such as C++, to provide persistence, concurrency control and other database capabilities. These systems provide a query language which also operates in the application program environment sharing the same type system and data workspace. We now discuss some systems that fall under this category.

2.1.1 O_2

The O_2 database system [Deux90], a research prototype that has been transformed into a commercial product, was developed at GIP Altair, in Le Chesnay, France.It is object oriented database system based on a client server architecture. The goal of this project was to design and implement a next generation database management system. This system supports a set of database programming languages, CO_2 and $BasicO_2$, a set of user interface generation tools called LOOKS and a programming interface called OOPE. The structure of O_2 can be broadly classified into OOPE, LOOKS, the query interpreter, the schema manager, the object manager and the disk manager.

The query language implemented in O_2 is powerful at the same time being declarative in nature. It can access attributes or execute methods as part of its ad-hoc query facility. It is also possible to restrict operations to execution of methods when it is important to preserve encapsulation. The query interpreter has a complete set of operations on a wide range of parameterised types. The query language itself can be used anywhere in the CO_2 database programming language thus providing the user with a powerful system with the ability to embed a declarative query and use control constructs on the result. O_2 also implements access methods for efficient retrieval. The multi index organisation is used to index aggregation hierarchies while a variation of the CH tree is used to index inheritance hierarchies. These access methods are described in chapter 5. B-trees and hash indices are also provided to manipulate collection of objects.

2.1.2 ObjectStore

The ObjectStore database system [LLOW91] is a commercial object oriented database system developed by Object Design, Inc. The first version (version 1.0) was released in 1990. Its goals are to provide a tightly integrated language interface to reduce the impedance mismatch between programming languages and database programming languages apart from providing a unified programming interface for persistent as well as transient objects. Its goal is also to provide object access speed of persistent data equal to that of an in-memory dereference of a pointer to transient data. It incorporates various features such as clustering of frequently referenced objects.

Query processing in ObjectStore includes parsing the query from its query language definition into a parse tree. Path information is propagated up the tree to the nodes that represent the queries. During code generation, a function implementing a scan based strategy and another implementing an indexed strategy are associated with the nodes representing the queries. This allows for considerable flexibility at runtime. ObjectStore offers three interfaces - a C library interface, a C++ library interface and an extended C++ interface available with the help of ObjectStore C++ compiler. Using the Object-Store C++ interface ensures parsing and optimisation are done at compile time itself. Queries using the library interfaces are optimised at runtime. Access methods such as the B^+ tree, hash indices (for collections) and other improvised associative access mechanisms for hierarchies are used to provide associative access.

2.1.3 ORION

The ORION database system [KGBW90] is a result of a next-generation database project initiated as part of the Advanced Computer Technology (ACT) program at Microelectronics and Computer technology Corporation (MCC). The goal of this project is to design a next generation database system and to integrate programming languages and databases. Various versions of the ORION system have been implemented.

Query processing in ORION includes parsing of the query language and construction of the language tree. Different varieties of nodes are identified with each node and semantics attached. As pointed out in [KGBW90], traversal can be done in a bottom up fashion or a top down fashion. Depending on the nature of the query, either the top down or the bottom up traversal is used in query execution. ORION implemented the query model in [Kim89]. ORION extends LISP with object oriented capabilities, with database calls for navigational or query access and for the definition of data types. ORION also implements the CH tree index for class hierarchies.

2.2 Database system generators

This architecture represents systems that allow implementation of a database system that is tailored to specific needs. These systems do not have a specified conceptual and internal data models and these can be defined by the implementor. Conceptual data model refer to the query and representation capabilities of the system such as query syntax. The internal data model refers to the access methods and how the conceptual data schema can be mapped to them. Examples of database system generators are GENESIS, EXODUS and SHORE. We now describe some features of these systems.

2.2.1 GENESIS

The GENESIS extensible database management system [Bato+90] was a prototype system developed at the University of Texas at Austin. The goals of this project were to introduce the tool-kit approach to the construction of database systems and to provide the system with powerful components to aid in the construction of a full fledged customised database system in minimal time.

The GENESIS project analyses the challenges to query processing presented by the layered approach used in building a tool-kit system. As pointed out in [Bato+90], the layered nature of the system implies that known query processing algorithms must be decomposed into layers to clearly separate concerns among the various layers of the implementation. It is important to preserve this layered hierarchy in order to preserve the advantages gained by being able to *plug and place* components to form a customised database system. A query specified at the external level has to be mapped by the system to its internal layers. The GENESIS system defines several such mappings. For example, the *external-to-conceptual* mapping maps the resolved views and maps the external queries into their conceptual counterparts. The prototype supports B^+ tree and hash indices.

2.2.2 EXODUS

The EXODUS extensible database system [Care+90] was developed at the University of Wisconsin. The goals of this project were to facilitate the fast development of high-performance, application-specific database systems.

EXODUS allows the construction of compiled queries by providing a mechanism to specify queries which are then parsed into a tree. Rule based query optimisation optimises the query and produces output code in E programming language which is a persistent version of C++ [Stro94]. The E compiler is then used to compile the query into object code. The user can also build custom access methods which can be used during the query execution phase as is done in OSHADHI database system [Vidy95]. It can be seen that EXODUS provides compiled queries but does not provide a facility for interpreted queries. EXODUS provides a library of type independent access methods such as the B^+ tree, Grid Files [NHS84] and linear hashing [Litw80].

2.2.3 SHORE

The SHORE database system [Zwil+94] is a persistent object system developed at the University of Wisconsin and is a successor to the EXODUS database system. SHORE represents a merger of database and file system technologies and allows file based applications to continue to operate without being rewritten. The need for multilingual access to data, especially in large databases, has been recognised and a language neutral solution provided. SHORE has a symmetric peer to peer architecture and provides a unix-like name space and access control model. The design itself is scalable. It can run on a single processor, a network of workstations or a large parallel processor. SHORE provides B^+ tree and other spatial access methods.

2.3 Our work

In this section, we describe the differences between our approach and those of the systems discussed in the preceding sections. The SHORE system, in its current implementation, does not provide high level query processing capabilities such as a declarative query language. We have attempted to build a query processing engine on SHORE which provides query capabilities, at the same time, preserving the stated goals of SHORE such as providing system extensibility. The query engine uses a query language that implements a subset of OQL [Catt94], which is an evolving standard for object query languages and can be considered to be the object oriented version of the popular SQL. The query processing engine uses recently developed high-performance techniques for object access that were unavailable at the time the earlier systems were built. The engine has been developed using public-domain technology, which permits easy porting and enhancement of the system functionality. We have designed the engine based on a formal query model which aids in efficient query evaluation and execution. We also provide a facility for interpreted queries rather than compiled queries, in the current implementation, to provide flexibility at runtime.

Chapter 3

Query Processing

A query is a means of retrieving or updating information in the database. Query processing refers to the steps involved in executing the actions specified by the query. Query processing in object oriented database systems throws up new challenges due to their powerful modeling capabilities. In this chapter, we present some fundamental concepts of object orientation, evaluate the differences between traditional and object oriented systems, provide an insight into the general steps involved in query processing and identify major components that have to be designed while building a good query processing system.

3.1 Object oriented Concepts

The basic abstraction provided by an object oriented system is an *object*, which is an encapsulation of the state and behaviour of a real or abstract entity. The properties of all real world and abstract entities are captured in the abstraction that is an object. Thus, all query processing systems operating in the object oriented framework must deal with this abstraction. Thus, in this section, we review some of the fundamental concepts which form the basis for any object oriented query processing system.

- *Object Identifiers*

  The object oriented modeling paradigm suggests the modeling of every real world entity as an object. Each object has an unique identifier called an object identifier (OID) associated with it to differentiate it from other objects. Object oriented database systems provide objects with persistent and immutable identifiers: an object's identifier does not change even if the object modifies its state.

  The OIDs can be logical or physical in nature depending on the system requirements. Physical OIDs represent disk addresses of objects and have the advantage of having no administrative overhead in fetching an object once its identifier is known. However, movement of objects causes problems in the OID maintenance. It maybe necessary to update all objects having a reference to the object in question in the event of relocation of the object. Logical OIDs, on other hand, are independent of the object's actual location on disk and hence can be used irrespective of the object's location on disk. The disadvantage, however, is an extra level of mapping of logical OIDs to disk addresses.

- *State and Behaviour*

  Each object is characterised by its state and behaviour. The state of an object is described by a collection of *attributes* and their values. Attributes themselves maybe literals or composed of other objects. Furthermore, an attribute of an object may represent a collection.

  The behaviour of an object is characterised by a collection of *methods* that define the set of operations that are possible on the object. This representation of an object provides several advantages (for example, encapsulation).

- *Class*

  In object oriented terminology, a *class* groups various objects having the same structure and behaviour. *Instances* represent the state of various objects belonging to a class. Thus, a class can be thought of as a blueprint representing a set of similar entities in the database. Just as a relational database query was framed against a relation or a set of relations, an object oriented database query is framed against an object or a set of objects. A class maybe a primitive class (also called a literal). A primitive class (for example, integer) captures the existence of primitive types in the object oriented framework.

- *Inheritance*

  An object can inherit the structure and the behaviour of another object in addition to having its structure and behaviour. This relationship leads to a *specialisation* hierarchy (or sometimes called subtyping) wherein inheritance leads to specialisation of structure or behaviour or both. Inheritance provides extensibility which is an integral requirement of every complex system.

- *Association*

  An object interacts with other objects. This interaction defines a relationship between objects also called as an association. For example, the relationship between a *Student* class and an *Instructor* can be modeled as a simple association.

- *Aggregation*

  An aggregation is a complex association which represents the *part–of* relationship between objects. An object is allowed to contain an attribute whose domain is that of another object. This gives rise to nesting of objects resulting in *aggregation hierarchies*. It is common to find aggregation hierarchies in most object oriented systems since most complex objects represented by these system are naturally modeled this way.

3.2 Object oriented and relational query processing

In this section, we try to assess the fundamental differences between relational and object oriented query processing. The differences arise due to the rich modeling environment in object oriented systems.

- *Added Types*

  Relational systems provide a relation as the basic level of abstraction which is constrained by the first normal form to contain only literals. This homogeneity facilitated uniform treatment of all relations which was exploited to provide efficient processing. However, object orientation brings with it features such as added base types and user defined types. These added types prevent a uniform treatment of all entities thus necessitating specialised handling.

- *Formalisms*

  Relational systems had sound formalisms to express their queries. The operations and the operators were well defined and formed an exhaustive set of operators that the system had to deal with. Formal languages (for example, relational algebra) were able to capture the semantics of queries and defined operations among operators which helped in rule based algebraic optimisation. Object oriented systems lack a formal object oriented algebra which impedes the optimisation process. Most of the object oriented algebras that exist are mere extensions of relational algebra and do not take into account all operators that have been added due to added types and operations on them.

- *Result typing*

  The result of a query expression in these two systems differ in a fundamental way. In relational systems the result of an operation (for example, join) on a relation (or a set of relations) results in another relation which contains only primitive types. A relation can thus be viewed as a weak binding of primitive types and the relational schema a collection of such relations. The primitive types are thus used in the context of all schema definitions and are hence independent of any particular schema.

  However, in object oriented databases, the result of an equivalent operation on a class (or a set of classes) results in a type whose structure is dependent on the class (or classes) against which the query is directed and most importantly the result may be a user defined type which may in turn contain other user defined types.

The schema itself is a collection of all user defined types and as such the types are not universal in nature and are tied to particular schemas.

Operations on objects can cause new types to be created. Since such an operation is defined only at runtime, it requires dynamic typing, which is essentially creation of types at runtime. Some systems like ENCORE [ZM91], try to support this feature with the help of parameterised types used to contain intermediate types which have no static definition. Thus, it can be seen that there is a need for the creation of dynamic types, which are new types generated at runtime, unlike static types which are defined at compile time itself.

- *Traversal along aggregation hierarchies*

  The domain of any attribute of a relation in a relational system is that of one of the primitive types (due to the first normal form). However, in object oriented databases, a class can have attributes whose domain can be that of another class. This leads to aggregation relationships between classes. This relationship leads to what is called *implicit* join between classes where the join is based on object equality and is not specified explicitly by the user. This is in contrast to *explicit* joins which are similar to relational joins in that two objects are explicitly compared on the values of their attributes.

  Some systems support only implicit joins based on the argument that explicit joins are needed in relational systems to recompose entities broken down during the process of normalisation. Such a decomposition does not feature in object oriented systems since the models directly support complex objects. Relationships in object oriented systems are maintained with the help of references thus further reducing the necessity of explicit joins.

  It can be noted that the implicit join is different from the relational join in that the join is defined by the schema itself. Also, the implicit join implies a unidirectional join in that the join order is predetermined. Specialised techniques maybe required if the query is specified againist the direction of the join.

Aggregations also lead to recursive querying. This is not possible in relational systems since any relation contains only primitive types. In object oriented systems, a class can refer to itself either directly or indirectly thus causing the possibility of having *recursive* queries.

- *Parameterised types*

  Relational systems generally do not allow the presence of multi valued attributes which are eliminated during the process of normalisation. However, attributes of classes in object oriented systems can be single-valued or multi-valued thus necessitating specialised handling (for example, iteration).

  Also, the presence of parameterised types (sets, bags, lists) to represent multi-valued attributes imposes additional requirements on the type inferencing mechanism since operations can be performed on collections of different types.

- *Scoping and subtyping*

  An object oriented database may contain a hierarchy of classes which may represent various levels of specialisation. This hierarchy, also called inheritance hierarchy, leads to subtyping (specialisation). This property has a significant impact on query processing in object oriented systems and is totally absent in relational systems. The existence of subtypes necessitates the definition of the scope of any query directed at such a hierarchy rooted at a class. It becomes necessary to determine whether all instances of the class and its subclasses need to be retrieved or the scope should be limited to the class at which the query is directed at. The result of a query directed against a class hierarchy is composed of a heterogenous collection of objects as a consequence of scoping. Thus, elaborate type inferencing schemes need to be devised because universal operators in object oriented databases may now operate upon a heterogenous collection of objects.

- *Dynamic operator definitions*

  The possibility of method definition in object oriented database systems allows extensibility in the query language. A new operation can be introduced into a class by adding a corresponding method by which the predefined set of operators present in the query language can be augmented. However, this requires query languages to handle methods as well. Such a possibility does not exist in relational database systems where such an enrichment in the query language would require additions to the query processor itself.

- *Optimisation*

  In relational systems, query optimisation depends on the knowledge of access paths which essentially represents the knowledge about the physical representation of data. Cost based query optimisation evaluates various query plans based on cost of access and produces the best query plan. The encapsulation provided by object oriented systems requires cost estimation of methods which maybe difficult to estimate especially if the methods are implemented in a general purpose programming language. Also, storage information is obscured by the encapsulation.

- *Access Methods*

  Efficient query processing demands the necessity of specialised access methods. The presence of complex objects in object oriented systems necessitates more sophisticated access mechanisms than were present in relational systems.

3.3 Stages in Query processing

In this section, we describe the various stages in query processing. The stages described give a general overview of the steps implemented by any query processing engine, be it relational or object oriented. Thus, the stages outlined in this section help discuss the fundamental aspects of query processing that are necessary to build any query processing system.

Figure 3.1 shows the general stages in processing a query. This figure is similar to the one presented in [Grae93]. We believe that the stages hold in the context of both relational as well as object oriented query processing. The only difference being in the

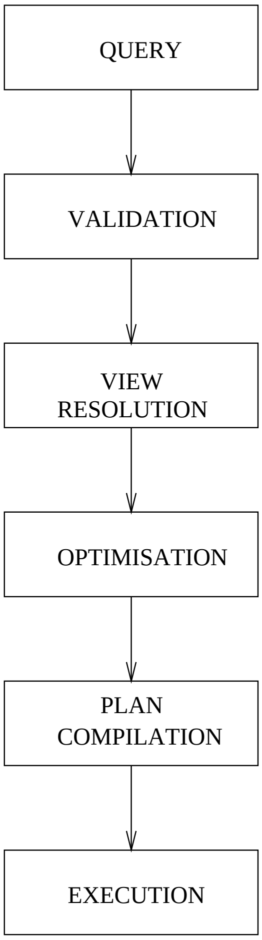


Figure 3.1: Stages in query processing

way each of the operations are performed in these two systems.

- *Query*

  The query itself is represented by means of a query language which is usually a high level declarative language. The query is then parsed and converted to a suitable internal form whose design is based on the operations that are required at later stages.

- *Validation*

  Next, the query is checked for errors. This process is usually accomplished with the help of schema information. This validation process helps to check if the referred entities are valid entities present in the schema. This step is important in preventing runtime errors which may occur as a result of a request for a non-existent entity.

- *View resolution*

  Views are macro functions used to define frequent operations (for example, a query that uses two relations in a relational system). Though physically separate, the relations can be viewed as a single logical entity as far as queries are concerned. Views are also used to restrict access to targeted entities in the schema.

  At the view resolution stage, a mapping of these logical entities is done to the actual storage structure of the entities. Object oriented systems may use views for restricting access to entire objects in a system.

- *Optimisation*

  Optimisation is a process that tries to finetune the system with respect to a performance measure. The performance measure may vary from one system to another depending on their constraints. Examples of performance measures include memory costs and response time.

  Optimisation involves generation of possible execution plans (also called query evaluation plans) for the given query and determining the best solution among them. The optimiser generally has a knowledge of the operators and their properties (for example, commutativity). The optimiser also considers available access methods while generating the best query plan.

- *Plan Compilation*

  Query evaluation plans generated by the optimisation process are converted to a form suitable for the query execution engine. This stage essentially involves the traversal of the operator tree by employing relevant tree traversal algorithms. The result of this phase could be machine code or even an interpreted language.

- *Execution*

  The Execution phase involves the actual execution of the query evaluation plan. The results of the execution maybe a formatted display or another target entity. Object oriented systems pose several problems due to the presence of typed objects and possibly new types generated at runtime as discussed in section 3.2.

3.4 Issues in Query processing

Efficient query processing involves addressing various issues which aid in increasing the performance level of the system. In this section, we attempt to identify major issues in query processing. We believe that such a process helps in finetuning the system. Figure 3.2 shows the major issues involved in building a query processing system. We briefly look at each issue.

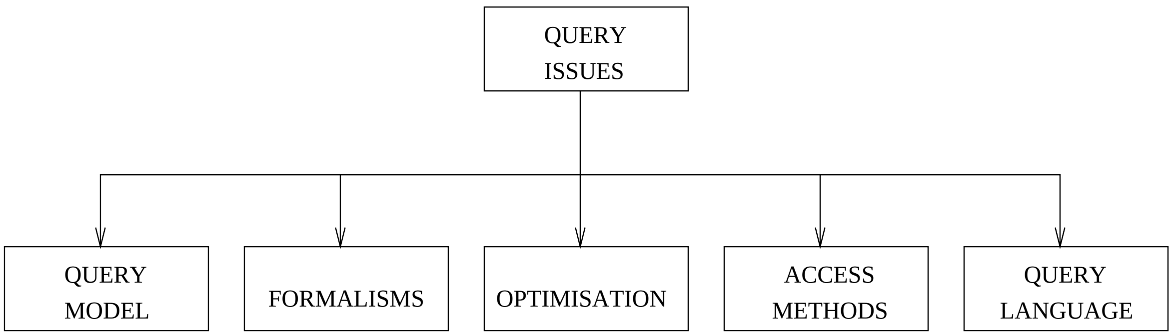


Figure 3.2: Issues in query processing

- *Query Model*

  A query model tries to capture the query processing environment in order to be able to tackle the complexities of ad-hoc query processing in advanced databases. This provides a formal tool to analyse and implement a query processing system. Query Modeling has become an important aspect of a query processing system in an object oriented environment due to the additional complexities that have to be handled by it. Many Query Models have been suggested in the literature [BKK88, Kim89, CM84, KKD89a] which address various query modeling issues in a query processing environment.

- *Formalisms*

  Formalisms aid algebraic optimisation, which uses algebraic properties to transform queries into semantically equivalent, but cheaper ones. Thus, formalisms aid the query optimisation process by providing certain algebraic transformations. Formalisms define operators and their properties at a logical level independent of any system. Thus, the algebra is sometimes referred to as a logical algebra and the operators defined as logical operators.

- *Optimisation*

  Optimisation itself is a mapping from the logical operators provided by the query language to the actual physical operators that implement these logical operators. The mapping process itself maybe complex since logical and physical operators frequently do not map directly into one another. For example, a duplication removal algorithm implements a part of the projection operator. Also, different systems may implement the same logical operators but maybe quite different in their physical implementation. For example, a system may implement hashing technique while another does not and hence the operation is quite different in these two systems. Due to these problems, the optimisation process is complex and is usually executed by supplying cost functions for the physical operators.

- *Access Methods*

  Access methods provide associative access and are an extremely important component of any query processing system. For example, a query processing system that implements a spatial access method is far superior to one that does not if the system is being used to store spatial data. In the object oriented domain, the selection of access methods to be implemented maybe domain dependent unlike in relational systems. For example, if a domain that is modeled has primarily aggregation hierarchies, the query processing system must implement an access method for aggregation after considering the semantics of data involved, their frequency of updates, etc.

- *Query Language*

  Navigational access is supported in all object oriented systems where objects can be retrieved by traversing a *graph* of objects. However, experience with relational systems has shown that high level, declarative and powerful query languages are an extremely important component. Thus, most object oriented systems provide a query language interface in addition to the navigational (program) interface. In reality, these two forms of access are complementary. A query may select a set of objects whose constituent objects may then be retrieved through navigational access.

  The design of a query language may not be simple. The query language itself, while providing a declarative interface, should have well defined and simple constructs. Relational systems saw the emergence of SQL as a standard. Standards of object oriented languages (OQL) are evolving [Catt94].

In this chapter, we discussed fundamental object oriented concepts, evaluated the differences in query processing in relational and object oriented systems, identified stages in a query processing systems and described the issues in query processing. We now address these issues in more detail.

Chapter 4

Query Model

A real world problem domain is captured with the help of an object oriented data model. Such a model disambiguates differences in interpretations and provides a single consolidated view with the help of various modeling constructs. Many such models have been proposed for object oriented modeling [Rumb+91, Booc94]. These models attempt to accurately capture real world problems in a representation that can be directly converted to a persistent model. These models provide the user with the power to accurately and naturally capture real world entities. Thus, these models address only representation issues. That is to say, a good object oriented model does not necessarily mean efficient data processing. To address the problem of efficient data processing, a query model is required which tackles issues related to data manipulation as opposed to data representation.

4.1 Overview

A Query Model is a tool to capture the working of any query processing system. It creates a basic framework on which algebraic formalisms can be built which can be used to finetune the system. Though it is derived from the constructs provided by the data model itself, it provides a concrete way to represent queries by capturing the semantics of a query in a deterministic fashion. Many systems provide an object oriented *shell* which essentially uses the power of the relational query model through a process of transformations [BKK88]. However, such an approach may prove counter productive in that such a transformation process may lead to inefficiencies. Object oriented databases thus require a query model that can preserve the advantages gained through powerful modeling techniques offered by such systems.

As pointed out in [Kim89], some query models [CM84, RS87] proposed in the context of object oriented systems neglect some of the features (for example, the effect of class hierarchy on queries). Some of the models also depend on value equality as opposed to equality based on OIDs. This is important in the context of defining the semantics of operators. We have chosen the model proposed in [Kim89], which is an improvement over the models suggested in [BKK88, KKD89a], as the basis and have attempted to add more functionality by introducing new constructs. We describe this modified model in this chapter.

4.2 The Query Model Representation

In this section, we describe the entire query model, its representation and operations on the graph itself.

4.2.1 Nodes and Edges

The query model is basically constructed as a graph of nodes and edges. The nodes represent classes (or parameterised types) while the edges represent the relationship between any pair of classes (or parameterised types). We further qualify the edges to be of three types – a composition (aggregation) edge, a generalisation (inheritance) edge and an association (reference) edge. We refer to the third type as a reference edge in order not to confuse the usage of terms since inheritance and aggregation are also forms of associations. The reference edge denotes the existence of a relationship between classes that are simple relationships that are neither aggregations nor inheritance. The graph itself is a directed (possibly cyclic) graph.

4.2.2 Direction of an edge

We define the *direction* of an edge in the context of the different types of edges. In the case of aggregation hierarchies, the direction of the edge is from the parent type to the type that is in its domain (we refer to this as the *aggregate* of its parent). In the case of inheritance hierarchies, the direction of the edge is in the direction of specialisation, that is, from the parent type to the subtype. It can be seen that the direction of the the aggregation and inheritance edges are determined by the schema itself. This definition is consistent with the one found in [Kim89]. We define the direction of a reference edge dynamically, that is, dependent on the query. In the case of the reference edge, the direction of the edge is so as to aid in the formation of a single directed graph for a query (for example, in the direction of an implicit join).

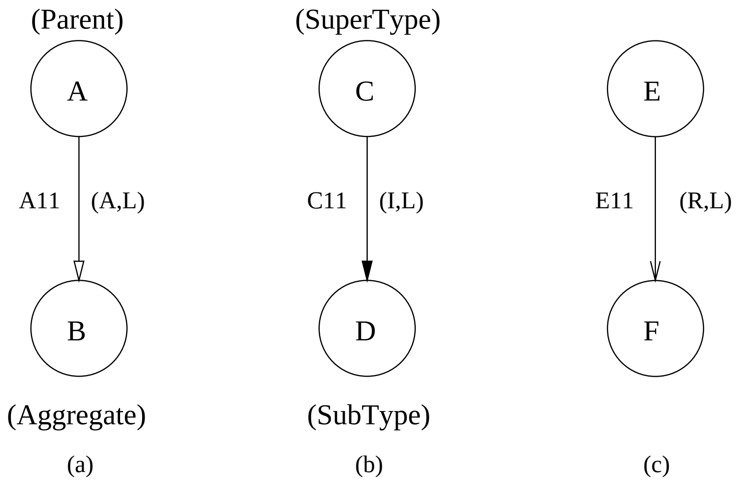


Figure 4.1: Nodes and edges in a query model

4.2.3 Notation

The graph constructs are shown in Figure 4.1. Figure 4.1(a) shows the representation of an aggregation relationship wherein a class A is composed of class B. Figure 4.1(b) shows the representation of a inheritance relationship wherein class C is a generalisation of class D. Figure 4.1(c) shows the representation of a relationship through a reference wherein class E is associated with class F through a reference. The direction of the reference is shown to be from class E to class F in this example.

Classes (or parameterised types) are represented by circles. The relationships are represented by the edges. The edge qualifying the aggregation relationship between two classes also has the name of the attribute from the parent class which is used in the implicit join (attribute A_{11} in the Figure). Primitive attributes which form ends of aggregation hierarchies are represented by their attribute name without an enclosing circle to differentiate them from non-primitive attributes. The attribute which is used as a reference to another object (attribute E_{11} in the Figure) is represented along the edge representing the relationship. In the case of an inheritance relationship, the attribute being inherited which is in the focus of an analysis is represented along the edge (attribute C_{11} in the Figure).

The symbol pair (also called *edge qualifiers*) (such as (A,L) in the Figure) qualifies the edge and denotes the possibility of complex relationships at the nodes of the graph. The first symbol in the pair represents the type of relationship that exists between the two classes related by the edge in question and the second denotes the existence (or absence) of a *dual* relationship at the parent node. The dual relationship refers to the possibility of a node being involved in an aggregation graph on one dimension and an inheritance relationship on another orthogonal dimension.

Symbol	Meaning
A	Aggregation
I	Inheritance
R	Reference
B	Bridge
L	Local
G	Global

Table 4.1: Symbols used in the query graph

The symbols used in the representation of the edges are summarised in table 4.1. While the symbols A,I,R and B signify the type of edge (arrowheads also depict the same information), the symbols L and G help in defining the scope of the query. Local scope is used to denote the target of the query is the class itself while global scope is used to denote the target of the query is the class hierarchy rooted at that class. We describe the various combinations and their implications on the query engine which will eventually operate on the query graph in the next few subsections.

4.2.4 Aggregation Edge Qualifiers

The symbol pairs (A,L) and (A,G) qualify edges which represent the aggregation relationship between two classes on the query graph. (A,L) implies that the parent node is not involved in a generalisation hierarchy and hence processing of this edge would be similar to a normal navigational access. (A,G) on the other hand, implies that the parent node is involved in a generalisation hierarchy and hence the explicit edges represented in the query graph maybe insufficient to express the semantics. In this case, the query graph can be thought of as having *virtual* edges connecting each of the nodes in the generalisation hierarchy to the aggregate node that the edge connects. A simple example is shown in Figure 4.2 which has classes A,B,C and F forming an aggregation hierarchy while classes C,D and E form an inheritance hierarchy. Virtual edges (represented by dashed lines in Figure 4.2) specify to the query engine the additional paths that it has to process due to the presence of relationships not indicated directly, but implicit in the query. It can be seen that these virtual paths convert generalisation hierarchies into purely aggregation hierarchies.

4.2.5 Inheritance Edge Qualifiers

The symbol pairs (I,L) and (I,G) qualify edges which represent the generalisation relationship between two classes on the query graph. (I,L) implies that the parent node is not involved in an aggregation either directly or indirectly (through a reference). (I,G) represents the fact that an entire aggregation has been inherited and hence requires the generation of virtual edges to capture the semantics of the query.

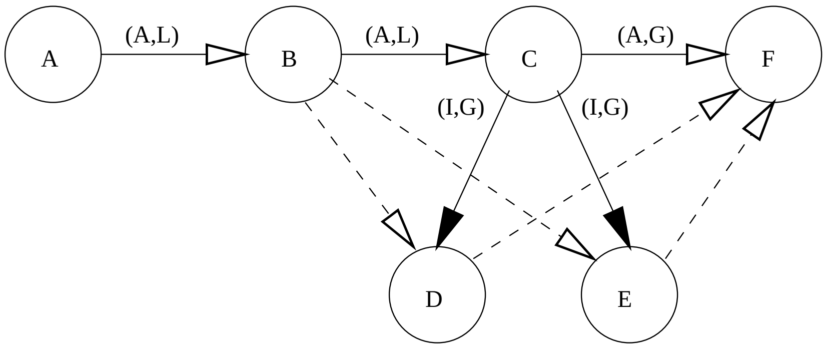


Figure 4.2: Virtual edges in a query graph

4.2.6 Reference Edge Qualifiers

The symbol pairs (R,L) and (R,G) qualify edges which represent a general association between two classes. (R,L) signifies a simple association, while (R,G) signifies that the parent node of the reference may be a leaf node of an aggregation hierarchy or a root node of an inheritance hierarchy. If the referenced node is also part of a hierarchy, then the reference itself can be called a bridge. Hence the bridge is a special case of a reference which connects two hierarchies creating what the query engine would look at as one *virtual* hierarchy. Such an edge is represented by the pair (B,G). An example of a virtual hierarchy is shown in Figure 4.3. The inheritance hierarchy formed by the classes A,B and C is linked to the aggregation hierarchy formed by the classes D,E and F with the attribute L forming the leaf.

The concept of the bridge is extremely useful in defining virtual hierarchies which provide considerable performance improvements if access methods look at them as such. It can be easily seen that a bridge can be involved in creating a virtual hierarchy in four different ways – two aggregation hierarchies may be linked, two inheritance hierarchies may be linked, or an aggregation hierarchy may be linked to an inheritance hierarchy or vice versa.

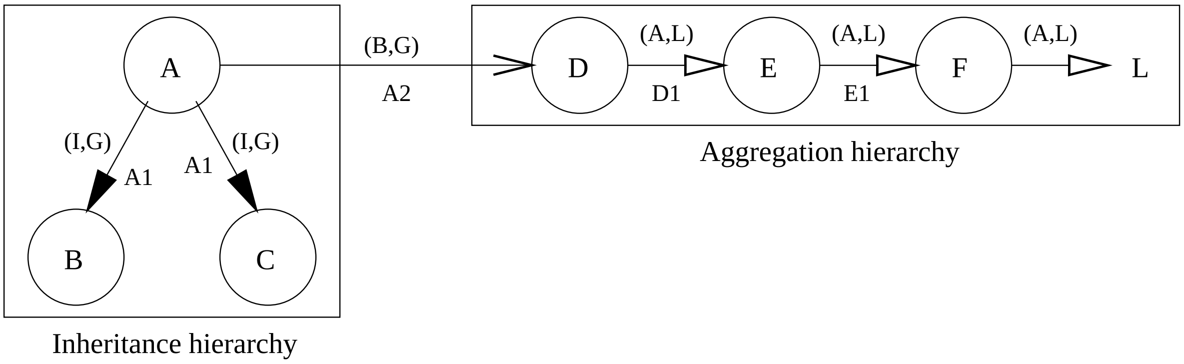


Figure 4.3: Virtual hierarchy

4.3 Definitions

This section defines some terms which are used in the context of query graphs.

1. **Node Compatibility**

   Two nodes are said to be *node compatible* when either both are of the same type (class) or they participate in subtype-supertype relationship with each other.

2. **Adjacency**

   Two nodes are *adjacent* if they are related with each other by a single edge. In the query model, three types of adjacency can exist depending on the type of the edge. Thus, adjacency due to aggregation is called *aggregate adjacency*, adjacency due to inheritance is called *subtying adjacency* and adjacency due to a reference is called *referential adjacency*.

3. **Edge Equivalence**

   Two edges of different query graphs are said to be *edge equivalent* if and only if they connect the same nodes and are of the same type and scope.

4. **Component**

   A *Component* of a query graph is a set of adjacent nodes and their connecting edges in the query graph. This definition of a component of a query graph helps in understanding a query graph as a set of such components. A node that is involved in relationships with two or more nodes would participate at the most in that many number of components. A *basic component* is one that has two adjacent nodes and its connecting edge. A Component is also called a subgraph.

5. **Complete path**

   A path is a sequence of classes involved in an aggregation relationship. A *complete path* in the data model is one that is maximal in the sense that no more nodes can be added to the path while preserving the semantic meaning implied by the data model. A *root* node is the node at the beginning of a path (not necessarily a complete path) while the *leaf* node is the node at the end of a path.

6. **Path Expression**

   A *path expression* is defined as a series of nodes of the query graph built on a path in the data model defined as

   $$C_1.A_1.A_2 \ldots A_n \ (n \geq 1),$$

   where C_1 is a type of the root node in the query graph, A_1 is an attribute of class C_1 represented by a node which is *aggregate adjacent* to the node represented by C_1, A_i is an attribute of class C_i represented by a node which is aggregate adjacent to the node represented by A_{i-1}, such that C_i is the domain of attribute A_{i-1} of class C_{i-1}.

   It maybe noted that the definition of path expression implies the constraint that path expression maybe only along the implicit joins defined in the schema and captured by the query graph. The concept of path expression is important since most query languages use some form of path expression to imply navigational access through an aggregate of objects.

   The length of the path is the number of edges along the path in the query graph specified by the path expression. The domain of the path is the domain of attribute A_n of class C_n.

7. **Path Instantiation**

   A path instantiation refers to the sequence of objects obtained by the instantiation of the path.

   A path instantiation is said to be *complete* if it contains an instantiation of the root node of a complete path as well as that of the leaf node in the path.

   A path instantiation is said to be a *left-partial* instantiation if the path instantiation does not include the root node in a complete path. Similarly, a path instantiation is said to be a *right-partial* instantiation if the path instantiation does not contain the node representing the domain of the path (leaf node). These definitions of path instantiations are consistent to the definitions found in [FMV94].

   It can thus be seen that a query graph may represent a complete, left or right partial instantiation of a path.

8. **Forward Traversal**

   Forward Traversal of a query graph implies the traversal of the query graph along a path instantiation in the direction of the path, that is, along the direction of the implicit joins. Thus, forward traversal refers to the traversal of a path in the query graph which starts at the root node of the graph and ends at the leaf node.

9. **Reverse Traversal**

   Reversal Traversal of a query graph implies the traversal of the query graph along a path given by the path expression against the direction of the path. Such a traversal implies the traversal from a leaf node of the path to a root node specified by the path expression.

4.4 Operations on the query graph

This section describes some basic operations on the query graph. Then, we can define the operations on the query graph as follows. These operations are used by the query processor.

1. **Union**

   The union of two components is obtained by merging compatible nodes and equivalent edges to form a resultant component. The total number of nodes and edges in the resultant component is reduced by the number which is equal to the sum of the nodes and edges in the two source components reduced by the number of merges. If the two components are *disjoint*, that is, the components have no compatible nodes (and hence equivalent edges), the resultant component consists of both the input components without any merging. The total number of nodes and edges is thus the sum of the nodes and edges of the two input components.

2. **Intersection**

   The Intersection of two components is obtained as a component having compatible nodes and equivalent edges from both the input components. If both the nodes are incompatible, the result is a NULL component.

3. **IsSubComponent**

   A component x is said to be a subcomponent of another component y if and only if all the nodes of x are compatible each with atleast one node of y, and all the edges in x are equivalent each with atleast one edge in y.

   A set of components A is said to be a subset of another set of components B if all the components of A are subcomponents of at least one component of B.

4.5 Benefits

The benefits of a query graph representation of a query are many and are analogous to the benefits obtained with the help of a data model.

- A query model helps in formalising the representation of a query. It also helps in capturing various components (for example, a path expression) involved in a query expression in the form of a uniform graph model. The operations defined on the query graph help not only in building a query graph from a query language but also during evaluation and semantic checking.

- Building a query processing engine in the background of a query model helps in exhaustive consideration of various combination of query constructs. This is an extremely important ingredient especially in ad-hoc query processing where the user, while having enormous expressive power, can generate a variety of combinations which the query processing engine should be able to handle.

- The query model is used to construct a query graph at runtime, besides it can be used at design time as a simple representation for analysis. It can also be seen that the model itself is directly implementable since the symbols can be directly mapped to corresponding data structures.

- Apart from query representation, the graph has the ability to express scoping in a simple way. Scoping is implemented with the help of virtual edges and hence provides a mechanism wherein local and global scoping is allowed by the system dynamically.

- Semantic checking can now be implemented as graph constructs thus providing an abstraction. For example, if an explicit join is attempted by the user against a query processing system that does not support such a join, the query graph constructed for the query would have two root nodes. A simple check of uniqueness of the root node is sufficient to catch this condition.

- The concept of a bridge and virtual hierarchies help in identifying such hierarchies which can be used in systems to provide specialised handling. It can be seen that such hierarchies can be identified in the static data model itself, but the query graph repeatedly instantiates the paths along such hierarchies thus creating statistical information (for example, the number of times a bridge is instantiated) which could be used to build certain indices dynamically to speed up access.

- The query graph provides a single view of the entire query, that is, the semantics of the predicate set are correlated. For example, this helps in deciding the objects along a path that need to be stored which will be used to produce the final result.

In this chapter, we described the query model which is extensively used in the construction of the query processor. We described the query model representation and the various constructs. Our contribution includes the model structure definition, a symbolic representation with the ability to represent scoping and references, the concept of virtual edges and virtual hierarchies, added definitions to aid in the development of formalisms and the definition of operations on the query graphs.

Chapter 5

Access Methods

In object oriented database systems, query processing involves traversing complex structures represented by the object oriented query model. For example, object oriented query processing systems need to support queries on objects forming a hierarchy. The hierarchy itself maybe a result of specialisation or aggregation and queries based on each of these hierarchies need to be processed differently. The traditional access methods such as B^+ tree [Come79], which gave relational database systems performance benefits by providing associative access to data, are not powerful enough to deal with complex data. Thus, the search for methods to deal with the properties of complex data has led to a variety of access techniques appearing in the literature.

In this chapter, we attempt to classify these access methods and provide a survey of access methods that are specific to a classification but independent of any problem domain.

5.1 Classification of queries

In this section, we classify queries based on the various associations that can be identified in any object oriented model. The classification provides a direct mapping to the type of queries or a combination of them that the database system maybe called upon to handle. Queries can be broadly classified as shown in Figure 5.1.

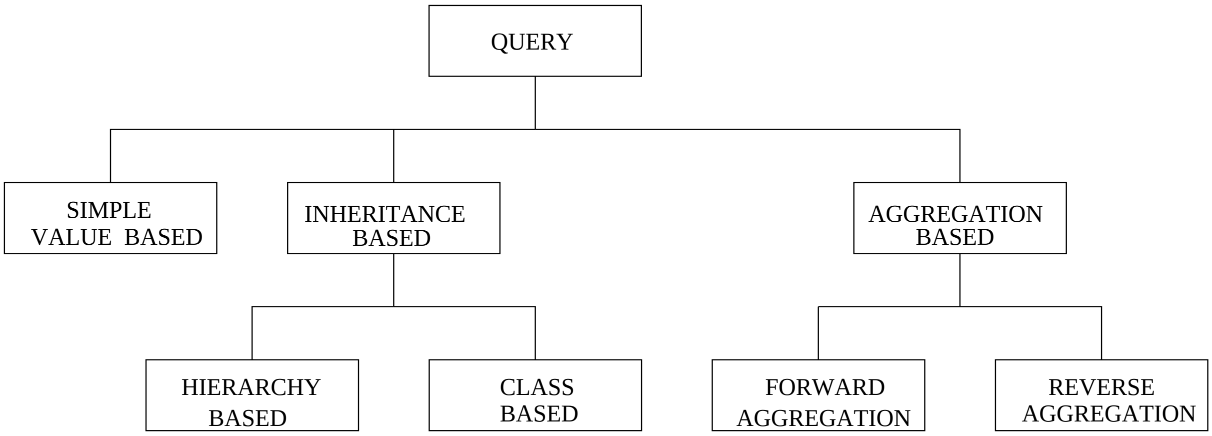


Figure 5.1: Classification of Queries

- *Simple Value based queries*

  Simple Value based queries are queries that essentially have simple literals or base types in their search predicates. It can be seen that all queries in relational database systems fall under this category due to the absence of complex relationships (for example, hierarchies). Thus, techniques used in the context of relational database systems can be exploited for simple value based queries.

- *Inheritance based queries*

  To a query processor, a query based on the inheritance relationship between classes, poses a number of problems. It maybe necessary for the query processor to recognize all objects with a specified value for an attribute which is at the base of an inheritance hierarchy. If this attribute has been inherited by other objects, it becomes necessary to identify all such objects as well. Thus, any access method that deals with *inheritance based queries* needs to be able to answer queries based on attribute values which are part of a hierarchy. These queries can thus be called *inheritance based class hierarchy* queries. It also maybe a requirement in some systems to be able to answer pinpoint queries which are directed at a particular class in the inheritance hierarchy. These queries can be called *inheritance based single class* queries.

Thus, in general, the query processor in an object oriented system must be capable of handling both hierarchy based queries as well as single class based queries targeted at inheritance relationships in the data model.

- *Aggregation based queries*

  An aggregation hierarchy is rooted at one object and a *forward* (top down) traversal of the hierarchy is required to extract an object at the leaf of the hierarchy in the absence of any access method. It can be seen that a path exists for the navigational retrieval of the required object. Navigational retrieval maybe expensive especially when object accesses are expensive. A potentially more difficult problem is to have a reference to an object at the leaf of the hierarchy and then attempt to find the root object of the hierarchy which requires a *reverse* (bottom up) traversal of the aggregation hierarchy. This problem is further amplified in systems that do not maintain reverse references along an aggregation hierarchy.

5.2 Simple value based queries

Simple value based queries in object oriented databases are similar to the queries seen in the context of relational database systems where queries were based on a single attribute or key value. In this section, we briefly highlight the characteristics of some of the popular indexing mechanisms used in relational databases systems and which can be exploited in the context of object oriented systems.

5.2.1 B^+ tree

The B^+ tree [Come79] is a n-ary tree which indexes all the records with the help of a balanced tree structure. The nodes of the tree are stored on disk and hence to access a record would require disk accesses proportional to the height of the tree. Being a balanced n-ary tree, it has rules associated with it for inserting or deleting records from the index. Since the organisation of data indexes is similar to the principles of the binary tree, it can be seen that the data appear in a sorted manner as the leaf nodes are traversed from left to right if the leaf nodes are chained. This property of the B^+ tree makes it extremely useful for range queries wherein data records whose indexed attributes fall within a certain range are required by the query. Since most commercial applications have such requirements, the B^+ tree is widely used in relational database systems.

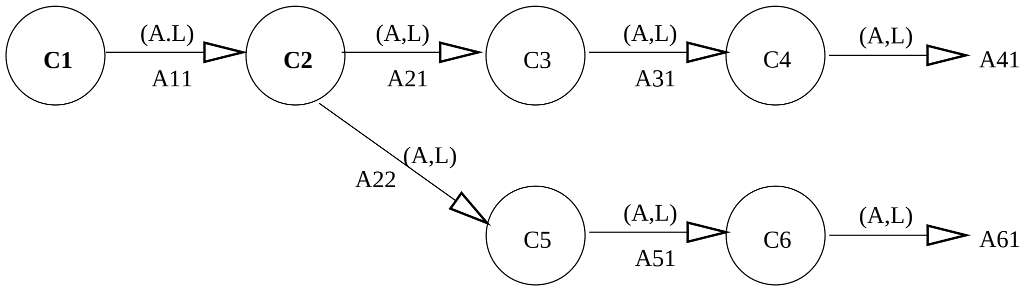
5.2.2 Extendible hashing

Extendible hashing [Fagi+79] is another method used to index data records. In this method, attributes on which the records are indexed are hashed onto buckets which in turn point to disk addresses. The hash structure itself is capable of dynamically changing in size in proportion to the size of the database without the need to change the hashing function thus avoiding problems associated with static hashing. Also, only a single disk access is required to retrieve the indexed record. However, hashing in general destroys the order of data and hence it is cumbersome to obtain a sorted order using the hash table. Thus, this structure is unsuitable for range queries and its utility remains limited to exact match queries.

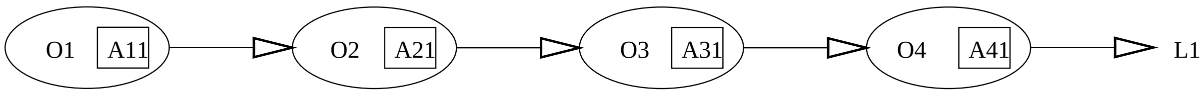
5.3 Aggregation based Access Methods

In this section, we present a collection of access methods to address the problem of handling aggregations in an object oriented model. We survey nested indices, path indices, multi index organisation and multi index with path configuration. A detailed survey of some of the access methods maybe found in [FMV94].

A sample query graph is shown in Figure 5.2(a) which will be used for the purpose of illustration in this section. It shows the class relationships that exist which form the aggregation hierarchy. Figure 5.2(b) shows a path instantiation $O_1.O_2.O_3.O_4.L_1$ of the path $C_1.A_{11}.A_{21}.A_{31}.A_{41}$ where O_i is an instance of C_i, $i = 1, \ldots, 4$, and L_1 is a literal. A path in the query graph which is of interest is depicted with a representation similar to the one used in [BK89]. The problem, is therefore to be able to retrieve all objects of type C_1 given a value for O_5. Normally, this requires an exhaustive search, but the presence of



(a)



(b)

Figure 5.2: Sample query graph and a path instantiation

access methods facilitates efficient retrieval of the required objects.

5.3.1 The Nested Index

Structure

The nested index [BK89] addresses the problem of traversals along a complete path instantiation by providing a *direct* link between the root and the leaf node of the path instantiation. The nested index was originally designed to facilitate queries that require reverse traversals of query graphs, but can be adapted to facilitate forward traversals as well. The nested index basically uses the leaf node of an aggregation hierarchy (which is usually a literal) as the key for a search and maps this value to the objects at the beginning of all path instantiations that result in having the required value for the leaf node. This idea of a nested index is best represented pictorially as shown in Figure 5.3.

It can be seen that the key value is the value of O_5 and the index contains the set of objects that contains this value through an aggregation hierarchy. The Figure 5.3 shows

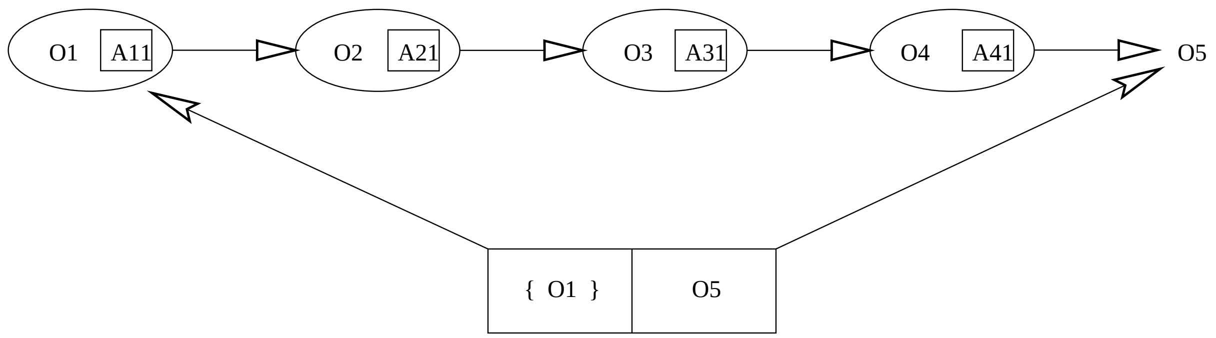


Figure 5.3: Nested index

only one element of the set, that is O_1. The number of path instantiations that lead to attribute A_{41} having value O_5 determines the cardinality of the set.

The collection of such nodes representing the associations between the leaf and root of path instantiations can be maintained in a B^+ tree with key value as the leaf nodes' value. This organisation wherein a B^+ tree is used to organise the nodes gives additional power in that range queries can also be addressed by taking advantage of this organisation and the capability of a B^+ tree to preserve order. The leaf node of such an organisation can be modeled as a node containing the record length, key length, key value, the number of elements in the set containing references to the root nodes of path instantiations and the list of OIDs of the objects which form the root nodes of the path instantiations.

Insertions and Deletions

Insertions are handled by updating the entries in the leaf node wherein the new OID of the root object of a path instantiation having the key value for the leaf object is inserted into the list of OIDs and the number of elements of the set updated. If a nested index node does not exist on an insertion, a new node is created and appended at the appropriate location in the B^+ tree. Deletions can similarly be handled by deleting the corresponding OID of the root object from the list and updating the count of elements in the set. The node itself is deleted in the case where the cardinality of the set becomes zero. Notice that if the path is of length one, the nested index organisation becomes identical to the simple B^+ organisation used in most relational database systems.

Partial Instantiations

The nested index structure also stores objects at the root of right partial instantiations apart from storing objects of complete path instantiations. The right partial instantiation, by definition, does not have the leaf object of the path in its instantiation and hence is represented by a NULL key value. Thus, all instances of the class at the root of a path are mapped by the nested index organisation. However, from the index it is not possible to determine the nature of the right partial instantiation in that it is not possible to determine the actual length of the complete path instantiation of which it forms a subset. The nested index organisation does not record any left partial instantiations since it stores only the root nodes of a path instantiation which by definition is absent in left partial instantiations.

Retrievals and Updates

It can be seen that nested index facilitates speedy retrievals. However, updates cause a problem. Updates refer to an operation where the value of an attribute (which could be an object reference in the aggregation hierarchy) is changed. In this case, one forward traversal is made to ascertain the leaf value with the original attribute value, another is made to ascertain the leaf value with respect to the new attribute value and a reverse traversal is made to obtain the object reference (OID) of the object at the beginning of the path instantiation. The nested structure is then updated using the result of these traversals. It can be seen that reverse references need to be maintained for this configuration to facilitate reverse traversals. Thus, the nested index configuration is most suitable for systems which primarily serve data retrieval requirements and where data updates are rare.

5.3.2 The Path Index

Structure

The path index [BK89] addresses the problem of traversals along a complete path instantiation by providing *direct* links between the objects of the path instantiation. The path index, like the nested index, uses the leaf node of an aggregation hierarchy as the key for a search and maps this value to all path instantiations that result in having the said value for the leaf node. This idea of a path index is shown in Figure 5.4.

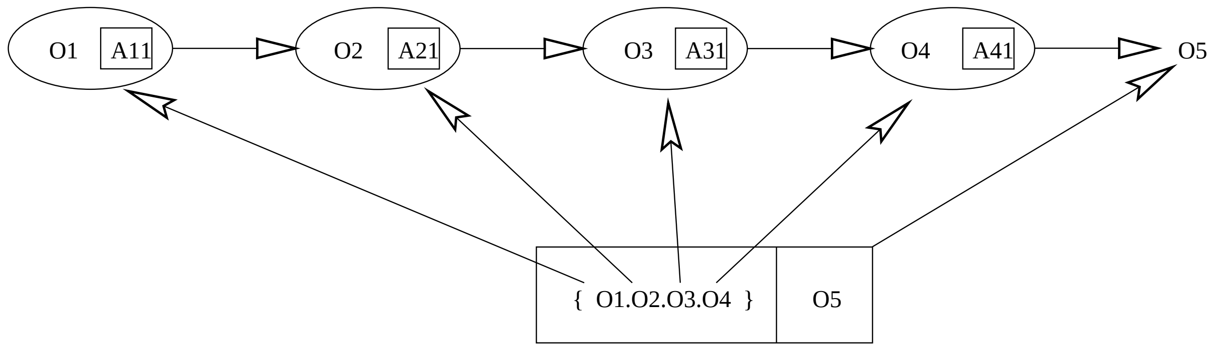


Figure 5.4: Path index

The key value used to map the path instantiations is the value of O_5 and the index itself contains all path instantiations that lead to the value of O_5 (The Figure 5.4 shows one such path). It can be seen that the path index contains a set of paths each path identified by a set of references (OIDs).

The collection of such nodes mapping the key values to the path instantiations can be maintained in a B^+ tree with key value as the leaf nodes' value (as in the case of the nested index). The leaf node in such an organisation can be modeled as a node containing the record length, key length, key value, the number of path instantiations and the list of path instantiations. A possible leaf node structure is shown in Figure 5.5. It maybe necessary to order the list of OIDs in each path instantiation depending on the implementation.

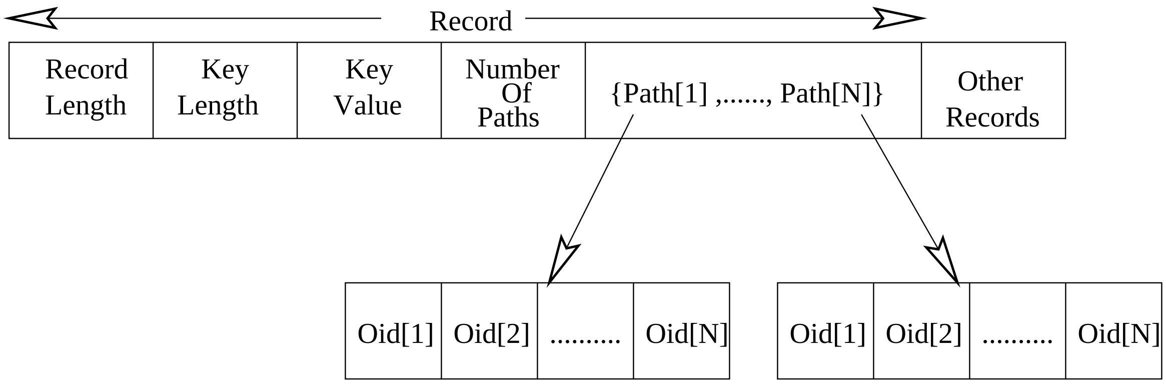


Figure 5.5: Leaf node in a path index

Insertions and Deletions

Insertions are handled by first finding the OIDs that contribute to a path instantiation to be inserted and then inserting this set of OIDs into the set of path instantiations. The count of path instantiations and other parameters are then updated. If a path index node does not exist on an insertion, that is, a path instantiation for the given mapping key value does not exist, a new node is created and appended at the appropriate location in the B^+ tree according to the rules of the B^+ tree insertion. Deletions are similarly handled by deleting the set of OIDs representing the path instantiation and decrementing the count of the path instantiations. The other parameters are also updated. If the set of path instantiations for a given key value becomes zero as a result of a deletion of a path instantiation, then the node itself is deleted from the B^+ tree following the rules for deletion in a B^+ tree. As in the case of the nested index, if the path is of length 1, the path index organisation becomes identical to the simple B^+ tree organisation used in most relational database systems.

Partial Instantiations

The path index, unlike the nested index, stores left and right partial instantiations in addition to storing complete instantiations. A NULL key value specifies a right partial instantiation while a left partial instantiation can be detected by the cardinality of the set of OIDs in a path instantiation. The path index, holds the OIDs of all the objects involved in a path instantiation and hence can be used to answer queries against nested objects with a single index lookup. It can be seen that this is not possible in a nested index. This retrieval of the entire path instantiation enables a special kind of operation known as *path projection*. This operation can be used to project certain objects from a path instantiation instead of the whole path instantiation in order to address queries that require such an arrangement. This operation itself can be done dynamically depending on the nature of the query and the nature of the path instantiations (left partial, right partial or complete).

Retrievals and Updates

Updates in a path index do not possess the disadvantages faced by the nested index configuration. Two forward traversals are necessary to determine the key value with respect to the new path. and with respect to the old path. Since the entire path instantiation is available in the index node itself, the root node of the path instantiation can be determined from the index itself. It can be seen that no reverse traversal is required in order to update the index. Thus, this configuration can be used in database systems that do not maintain explicit reverse references as well. The main disadvantage of the path index configuration is in the fact that the leaf nodes now contain more information about every path instantiation thus reducing the number of such records that can be stored in a node. Thus, it maybe necessary to scan more leaf nodes than if a nested index were maintained in order to retrieve information. In summary, retrievals in the path index is *more costly* than the nested index while updates are *cheaper*.

5.3.3 Multi index

Structure

The multi index [MS86] is an organisation that essentially breaks a path into its basic components and uses a separate index (for example, a B^+ tree) to maintain all path instantiations of each basic component. Therefore, given a path $C_1.A_1.A_2 \ldots A_n$ $(n \geq 1)$, traversing n classes $C_1C_1C_2 \ldots C_n$, a multi index is defined as a set of n simple indices (called *index components*) $I_1, I_2, \ldots, I_n$ where I_i is an index defined on $C_i.A_i, (1 \leq i \leq n)$. All indices $I_1, I_2, \ldots, I_{n-1}$ are identity indices, that is, they have OIDs as key values. This definition is similar to the one found in [FMV94]. The last index, I_n, may or may not be an identity index depending on the domain of the attribute A_n. Thus, the multi index tries to provide an index at every basic component of the path thus indexing the entire path and hence the all path instantiations. Each of the indices are generally B^+ tree indices but could use other techniques as well depending on the nature of the path instantiations. A sample multi index organisation is shown in Figure 5.6.

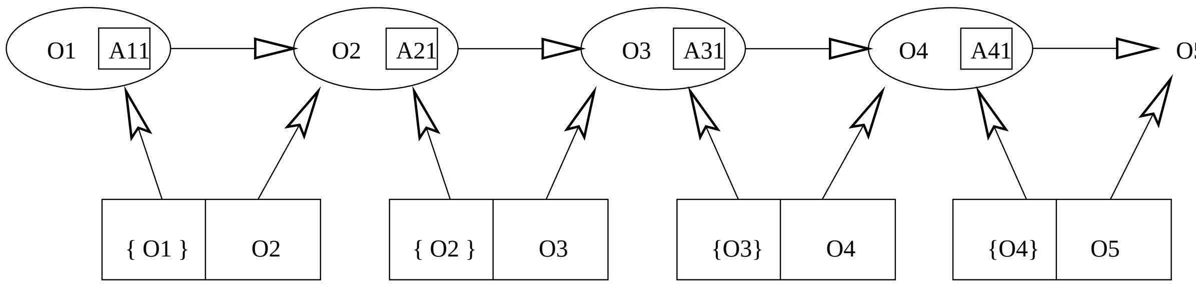


Figure 5.6: Multi index

Thus, a lookup requires reverse traversal of a number of indices equal to the path length of the path instantiation. Consider the single path instantiation as shown in Figure 5.6. Performing a lookup involves traversing the index keyed on value O_5 and obtaining the object references (OIDs) to instances which satisfy the criterion. For each OID in this set obtained (in this case, only O_4) a lookup is make in the adjacent index to obtain the set of references satisfied by the lookup. A similar process is followed backwards until the root node of the path instantiation is encountered. Now, the set of instances obtained from the root index gives the resultant set of all objects that satisfy the criterion of having the value equal to that in O_5.

Insertions and Deletions

Insertions of a path instantiation are handled by updating the entries along all the indices along the path to accomodate the new set of values. Thus, the insertion operation requires the insertion of a value in a number of indices which is equal to the path length of the path instantiation. The deletion operation is similar in that, it requires the deletion of the corresponding entries in all the indices that have information about the path instantiation being deleted. Thus, deletion also requires traversal of a number of indices equal to the path length of the path instantiation being deleted.

Partial Instantiations

The multi index organisation stores information about both left and right partial instantiations in addition to complete path instantiations. Right partial instantiations are recorded with the help of a NULL key value mapped to instances of the last class in the path. Left partial instantiations, however, require no such special identifier and are accomodated by the respective indices which span the partial instantiation.

Retrievals and Updates

Updates in the multi index organisation are the least expensive among the organisations discussed. This is because it requires updation of only a single simple index unlike in the cases of the path and nested index which required traversals along the path instantiation. An object updation thus is done *locally* in the sense that it is local to the basic component of the path in question and is independent of the other components that make the path. However, it can be seen that retrievals are costly due to the necessity of traversal of a number of indices equal to the length of the path instantiation in question. It may also be noted that this organisation facilitates queries that require *reverse traversal*.

5.3.4 Multi index with path configuration

Structure

To motivate the necessity of a path configured multi index, we summarise the characteristics of the nested, path and multi index. Nested and Path indices have high update costs especially if the path indexed is of length greater than 3. However, they have low retrieval costs. In contrast, the multi index has high retrieval costs while having low update costs. Thus, it can be seen that an intermediate solution can be obtained by splitting the path into subpaths and if these subpaths are optimal for the given path, the lowest cost solution can be obtained.

An algorithm for finding the optimal configuration can be found in [Bert94]. This algorithm takes as input the frequency of various operations (insertion, deletion, retrieval) for the classes along the path besides taking into account whether reverse references are maintained along the path. Then the algorithm outputs the optimal subpath configuration for the given path and also assigns the type of index (nested, path or simple) that should be assigned to each subpath. Most importantly, the algorithm decides if some subpath does not require any index. The algorithm also identifies circumstances under which the absence of indices along a path may turn out to be an optimal configuration. This happens when there is minimal sharing of subpaths between various paths and reverse references exist along the path. Thus, reverse references maybe the best solution in systems that possess the above characteristic and are constrained in terms of space which is a primary requirement in order to be able to build indices.

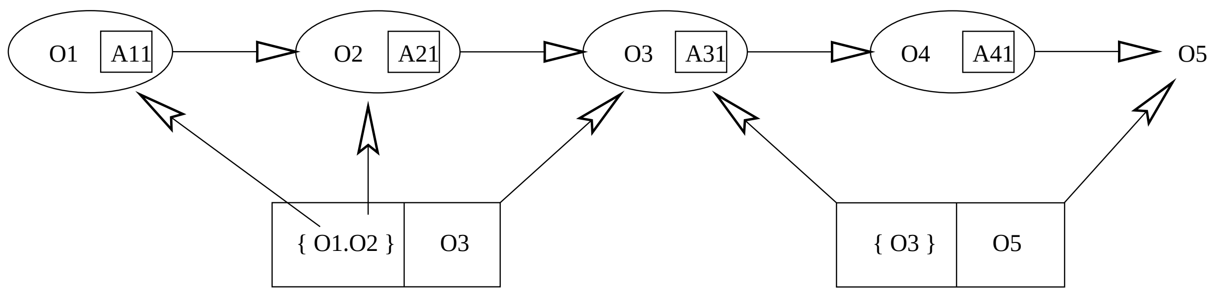


Figure 5.7: Multi index with path configuration

For the purpose of illustration, Figure 5.7 shows indices allocated to subpaths as a possible result of application of the algorithm to a path having a higher frequency of operations on the entire right partial instantiation represented by the subpath $C_1.A_1.A_2.A_3$. The subpath represented by $C_3.A_3.A_4.A_5$ has a low frequency of operations on the entire subpath. Thus, it can be seen that the right partial instantiations mentioned in this example are indexed by a path index while the left partial instantiations are indexed by a nested index.

Insertions and Deletions

Insertions require addition of the subpaths of the path instantiation into the respective indices allocated for the path instantiation. Deletions, similarly, require the subpaths (of the path instantiation being deleted) to be deleted from the corresponding indices.

Partial Instantiations

This multi index organisation may store left partial or right partial instantiations depending on the path organisation. They differ from the simple multi index organisation in that all nodes along a path may not be indexed. Thus, the entire partial instantiation may not be stored.

Retrievals and Updates

Retrievals are cheaper than in the multi index case since the number of indices that need to be traversed in the retrieval is smaller in this case. The retrieval operation itself is similar to the one in the multi index case where the key value for the search is used to obtain the list of OIDs of instances (along path instantiations that map to this key value) from the index at the end of the path instantiation. This list is then used on the relevant index at this portion of the path instantiation to obtain OIDs that further aid in the *backward* construction of the path. This process is continued till the root of the path instantiation is obtained. Updates are cheaper than in the case where only a nested or a path index has been used. This is because any update requires forward (and possibly reverse) traversal only on the subpath indexed and not on the entire path instantiation. Thus, it can be seen that the multi index with path configuration provides a structure wherein a tradeoff is obtained with respect to retrievals and updates. The main disadvantage of this organisation is that, to obtain a optimal path configuration, statistical data is required which may not be readily available or may not be a precise indicator of the frequency of operations.

5.4 Inheritance based access methods

In this section, we survey techniques used to address the a problem orthogonal to the problem of indexing aggregation based queries. The data model allows a hierarchy of classes to be involved in such a relationship. Since an attribute of a class C is inherited by all classes in the hierarchy rooted at C, the scope of a query directed at the class C is, in general, an inheritance hierarchy rooted at C. An example of such an inheritance hierarchy is represented as a query graph shown in Figure 5.8.

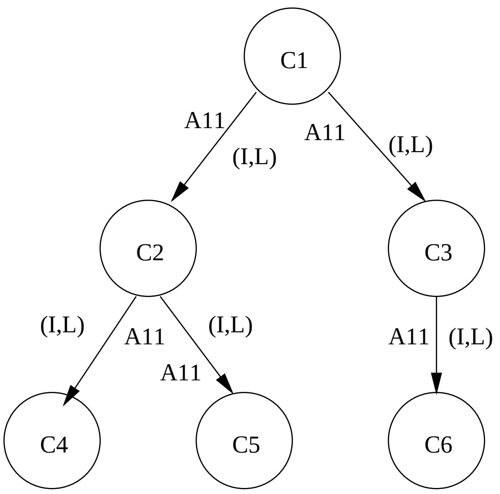


Figure 5.8: A query graph of an inheritance hierarchy

It can be seen from the figure that an attribute A11 of the class C1 is inherited by all subclasses of C1 and subsequently their subclasses. A solution to index such an organisation of instances is to maintain a separate index for instances of each class. This organisation is called a *multi-level single-class* index. However, this organisation does not take into account the relationships that exist among the objects and hence more sophisticated mechanisms exist some of which are surveyed here.

5.4.1 The CH tree

Structure

The CH tree [KKD89b] or the class-hierarchy tree is an index organisation that attempts to maintain an index on an attribute for all instances of all classes in a class hierarchy rooted at a particular class. Thus, unlike in the single-class index, one index is maintained for an entire inheritance hierarchy.

The CH tree is essentially a modification of the B^+ tree. The internal nodes of the CH tree are similar to that of the B^+ tree. The leaf nodes differ from that of the B^+ tree in that they attempt to store information about an entire hierarchy. The structure of a leaf node of the CH tree is shown in Figure 5.9.

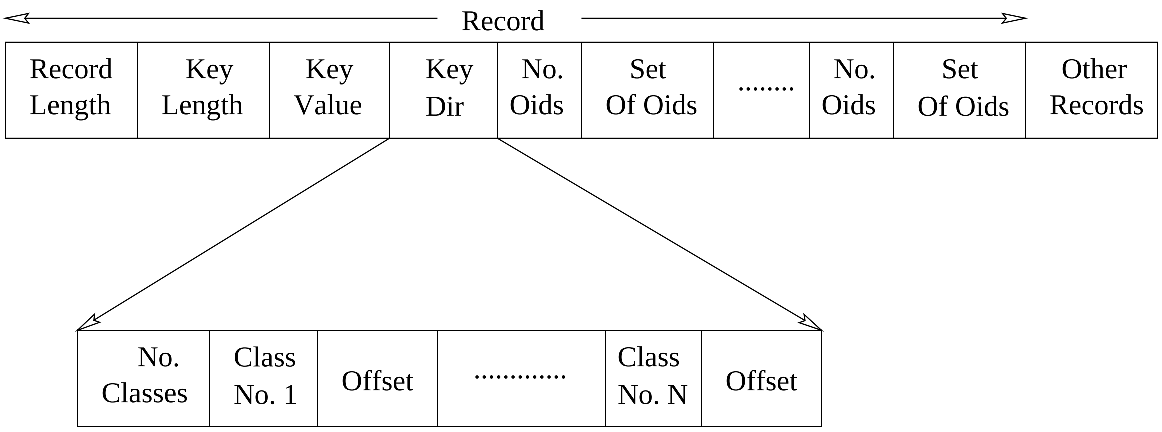


Figure 5.9: Leaf node in a CH tree

The leaf node consists among other information a key directory. The key directory contains the number of classes that contain objects with the key value in the indexed attribute. For each such class, the class identifier and the offset of the list of OIDs of objects of the class are specified. The OIDs are themselves grouped according to their types. This organisation facilitates queries directed at particular classes in the hierarchy as well.

Insertions and Deletions

Insertions are handled by inserting the OID into the corresponding list if one exists. A new list is created if no list is found. If the size of the index record becomes greater than that of an index page, additional pages are allocated to it and an overflow page pointer is maintained in the first page (not shown in the figure). Deletions are done by deleting the corresponding OID from the list. It can be seen that this organisation facilitates deletion of all instances belonging to a particular type as well. Merging of index pages can be done if the size of an index record is greater than an index page before a deletion and becomes less than an index page after the deletion.

Retrievals

Performance studies based on a cost model [KKD89b] have shown that the CH tree performs better than a single-class index if the number of classes in the hierarchy is at least two. In terms of disk space used for the storage structure itself, a conclusive result could not be found. The CH tree organisation itself is uniform in its treatment of queries against class hierarchies and against single classes in the class hierarchy. Queries that are directed against a particular class in the class hierarchy follow the same retrieval procedure as those for retrieving an entire hierarchy. Hence, the index page containing the list of OIDs for the class in question is obtained along with all the other classes in the hierarchy which is an overhead for this nature of queries. Hence, in general, CH trees perform efficiently against queries directed at the hierarchy itself and may not be a cost effective solution for queries directed at particular classes in the class hierarchy.

5.4.2 The H-tree

Structure

The H-tree [LOL92] attempts to alleviate some of the disadvantages of the CH tree. The H-tree organisation tries to preserve the superclass-subclass relationship while trying to improve the performance for single-class queries. A H-Tree structure is maintained for each class of a class hierarchy and the trees are nested according to their superclass-subclass relationship.

The H-tree itself is a modification of the simple B^+ tree index. The internal nodes of the H-tree differ from that of the B^+ tree in that they contain pointers to nodes of H-trees of subclasses in addition to storing key and the corresponding node pointers. The leaf node itself is similar to that of the B^+ tree node. The internal node of an H-tree is shown in Figure 5.10.

No Keys	B1	K1	.....	Kn	Bn+1	No Of Nodes	Set Of Pointers To Nodes

Figure 5.10: Internal node in a H tree

The internal node has all the fields that can be seen in a B^+ tree and in addition to those is present a count of the number of references to corresponding nodes of other H-trees representing the subclasses and the pointers themselves. Two classes involved in a superclass-subclass relationship would thus have two such trees allocated for them with added associations among the trees to facilitate hierarchy traversals. The internal node of the tree representing the superclass would have a direct link to a corresponding internal node in the tree representing the subclass. These links thus create a hierarchy of index trees. The range of values that fall under the subtree rooted at the referenced node can be determined from its parent node. This extra reference is generally avoided by maintaining the minimum and maximum values of the node along with the reference in the superclass tree node itself.

Insertions and Deletions

Creation of the H-tree begins with the creation of the tree component for the most specialised class and so on till the most generalised class. Insertions are made in a manner similar to the B^+ tree and nodes may split which may require readjusting in the set of pointers it contains in addition to readjustment in the set of pointers in the superclass tree node that references this node. Deletion causes similar readjustments if it results in merging of nodes.

Retrievals

Retrieving instances along a hierarchy involves complete traversal of the index maintained on the class at the root of the hierarchy and partial traversal of indices maintained on the subclasses taking advantage of the links provided. Queries directed at single classes are handled by referencing the corresponding index for the class.

This organisation essentially improves on the single-class index by providing links between the trees of classes involved in a superclass-subclass relationship. While the single-class index made no special provision for class hierarchy queries, the H-tree provides such an access which only requires partial traversal of trees for the subclasses involved. The H-tree removes the disadvantage of the CH tree which read and discarded information about irrelevant classes (in a query directed against a single class). For such queries, only the relevant class tree is accessed and hence no irrelevant information is read. However, for queries on entire hierarchies, the H tree organisation may need to make more accesses than with the CH tree.

5.4.3 The hcC tree

Structure

The hcC tree [SS94] is an index structure proposed which can handle both single-class based as well as hierarchy based queries efficiently. The hcC tree is a modification of the B^+ tree index. The internal nodes are very similar to those of the B^+ tree except for an additional bitmap at each node giving advance information and to prevent an unnecessary traversal of the tree.

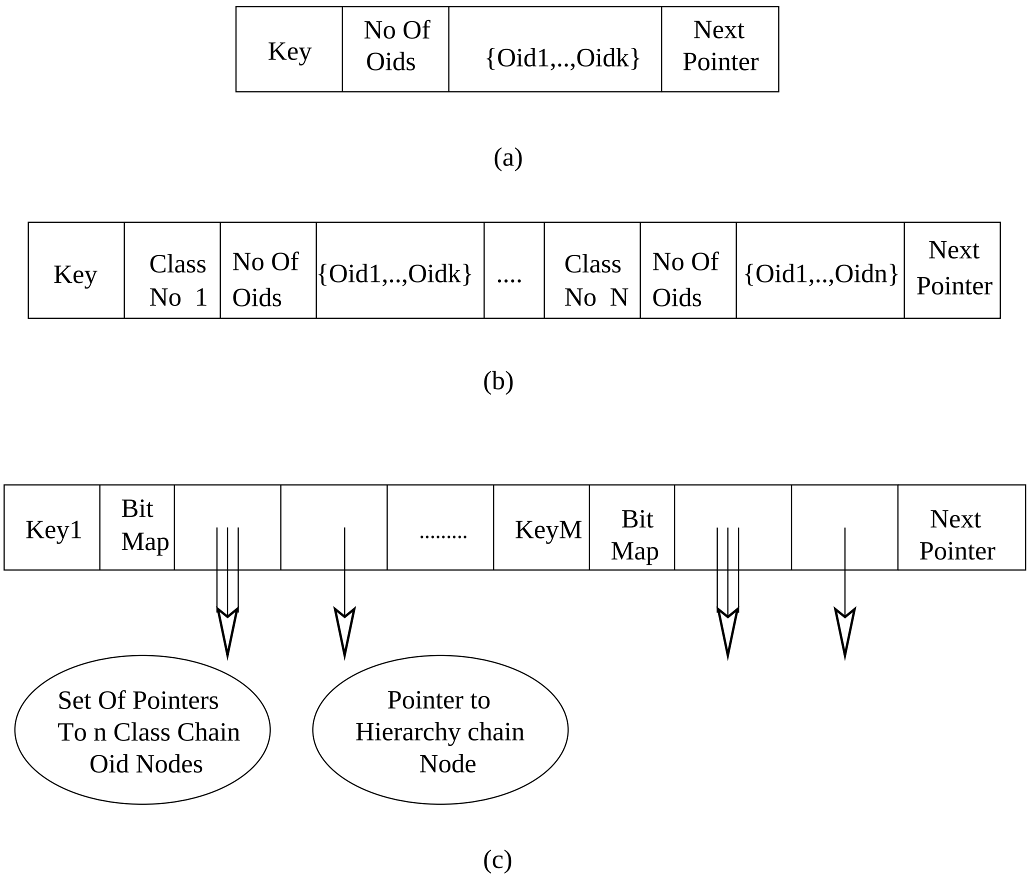


Figure 5.11: Nodes in a hcC tree

This bitmap stores information on whether a particular class has any instances at all. If not, a complete traversal of the tree is not required to find a null answer. The leaf node of the tree maintains the key value, the bitmap and a set of n+1 pointers. Of these n+1 pointers, n pointers point to the instance list of classes that make the class hierarchy and one pointer points to the instances that make the hierarchy itself. The nodes that the n+1 pointers point to are called OID nodes. OID nodes representing single class instantiations are called *class-chain* OID nodes while the OID nodes representing the hierarchy are called *hierarchy-chain* OID nodes. These OID nodes themselves form the last level in the hcC tree and corresponding OID nodes are chained together. That is, the OID nodes representing a class C are all chained together and all the hierarchy nodes are chained together as well. The Figure 5.11(a) shows a class-chain OID node, Figure 5.11(b) shows a hierarchy chain OID node and Figure 5.11(c) shows a leaf node of the hcC tree.

Insertions and Deletions

Insertions into the hcC tree involve inserting the value at the class-chain OID node as well as the hierarchy-chain OID node. Deletions cause a similar operation where the entry is deleted from the class-chain as well as the hierarchy-chain node. The advantage of the hcC tree is that it facilitates both single class queries as well as hierarchy queries. Both types of queries require the traversal of one index tree (unlike the H trees) and retrieval of the corresponding hierarchy or a subset of the classes in the hierarchy.

Retrievals

The performance results obtained in [SS94] showed that the H-trees, in general performed poorly against the hcC tree in the context of hierarchy based queries while the CH tree performed poorly in the context of single class based queries. Thus, it can be seen that the hcC tree provides a efficient mechanism in systems which encounter both types (hierarchy and single class based) queries.

Chapter 6

The Query Processing System

The query processing system has been designed to cater to any object oriented framework running with SHORE. It consists of three components – the query processor, the schema manger and the access methods. These components work in tandem to provide the query processing capabilities. Queries are modeled with the query model described in chapter 4. It maybe noted that the design is generic in nature and the implementation is independent of any particular data model representing a domain.

The query processing system has been used in the construction of the DIAS system, which aims to provide efficient data management for the vast amount of data generated during the parasitic analysis phase of the VLSI design process. The DIAS system project is being executed at the database research lab at our institute and aims at reengineering the data storage structure of parasitic data at Texas Instruments. Thus, the DIAS model implementation can be viewed as an application which uses the query processing system and hence we use it as an example while describing the query processing system and its components.

6.1 System Architecture

In this section, we describe the query processing architecture and its relation to other components of the system. Though the architecture is the DIAS system architecture, we expect all the components to exist in any system that uses the query processor. Figure 6.1 shows the architecture of the DIAS system relevant to the discussion and is a reproduction of a portion of Figure 1.2. The components that are part of the query processing system are shown as shaded boxes. We briefly describe the components DIAS system in this section.

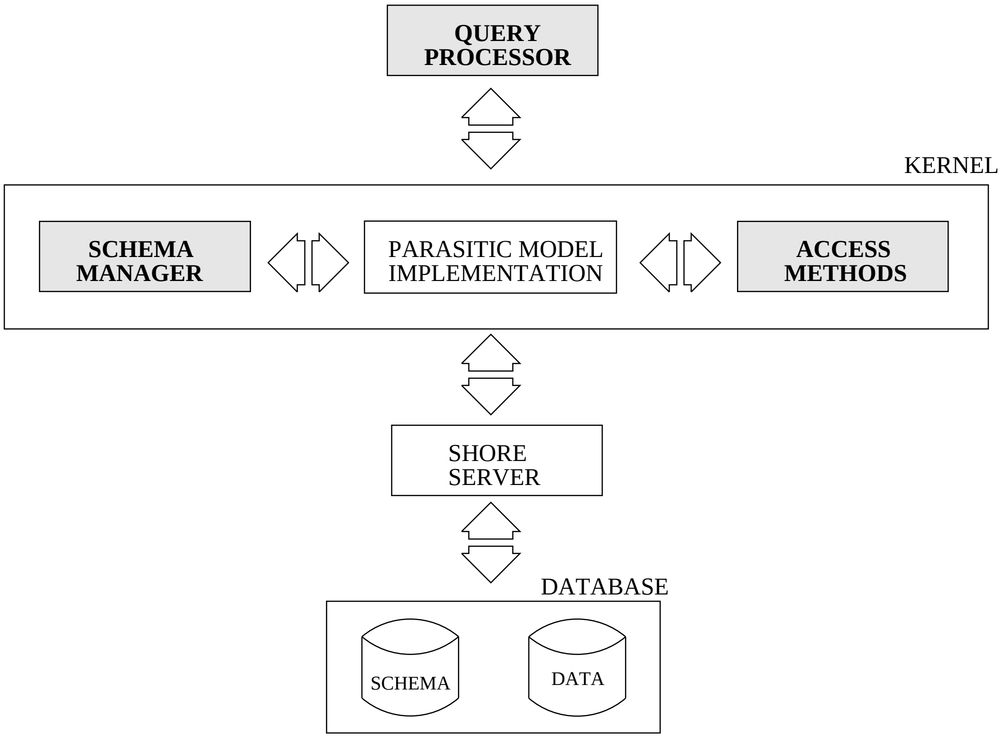


Figure 6.1: DIAS System architecture

SHORE forms the lowest layer of the system on which the DIAS data model is implemented. The data model implementation contains interfaces (classes) declared in the SDL interface provided by SHORE and the method definitions in a language binding provided by SHORE (SHORE provides a C++ language binding). The Schema manager is also built using the SDL interface, and maintains information about the types used by the object oriented model, their attributes and their methods. The schema manager is responsible for providing the data model independence of the query processing system. Access Methods are implemented in a similar manner (using the SDL interface) which can be accessed by routines implemented in the model itself or directly by the query processor. The data model implementation, the schema manager and the access method implementation form the core of the system and is called the *kernel*.

The query processor provides a declarative query language interface through which a user can query the database. The query processor translates this query with the help of the schema manager. The query is then executed by the query processor with the help of the model interface using access methods and making runtime optimisations. The methods of types in the object schema are used to query the database and such calls are made through the SHORE server. The database itself is shown to have a schema part and a data part. The schema manager interacts with only the schema while the method definitions of the model access only the data part of the database. These interactions are made possible by the SHORE server. This organisation prevents the user from interfering with the metadata (schema) through queries. The sections to follow discuss the design of each of these components in detail.

6.2 SHORE Server

The lowest layer of the DIAS system consists of SHORE (Scalable Heterogeneous Object Repository), a persistent object system developed at the University of Wisconsin [Zwil+94]. SHORE forms the lowest layer of the system and is responsible for object management and transaction related features such as concurrency control and recovery. The Value Added Server (VAS) uses these features of the Storage Manager and presents an interface targeted at specific client applications. DIAS uses two such VASs: the SHORE Value Added Server (SVAS) and the SHORE NFS server. The SVAS is useful for applications using the object oriented database features mainly while the latter is useful for applications relying on the file system features of SHORE.

SHORE was chosen as the backend engine for the following reasons: First, SHORE supports both file system and object oriented features, thereby ensuring that legacy application tools can continue to operate in the new environment. Second, it allows for databases built by an application written in one language to be accessed and manipulated by applications written in other object-oriented languages as well. This feature is especially important in the VLSI CAD domain where tools are implemented in a variety of languages. Finally, Shore provides application independent features such as support for transactions and object management.

6.3 DIAS data model

In this section, we give a brief idea about the design components in a circuit and then discuss the data contained in the DIAS data model. A design consists of *cells* and *interconnects*. Cells are essentially pre-characterised entities, that is, all information about them is readily available and hence needs no further attention. Interconnects form the portion of the circuitry that establish connectivity between various cells. A cell itself can include other cells (for example, a CPU design contains the design of a ALU) and so on thus forming a hierarchy. A cell which is at the lowest level of hierarchy in a design is generally referred to as a *leaf cell* or a *library cell* (for example, a NAND design). The port of a cell forms its input or output interface.

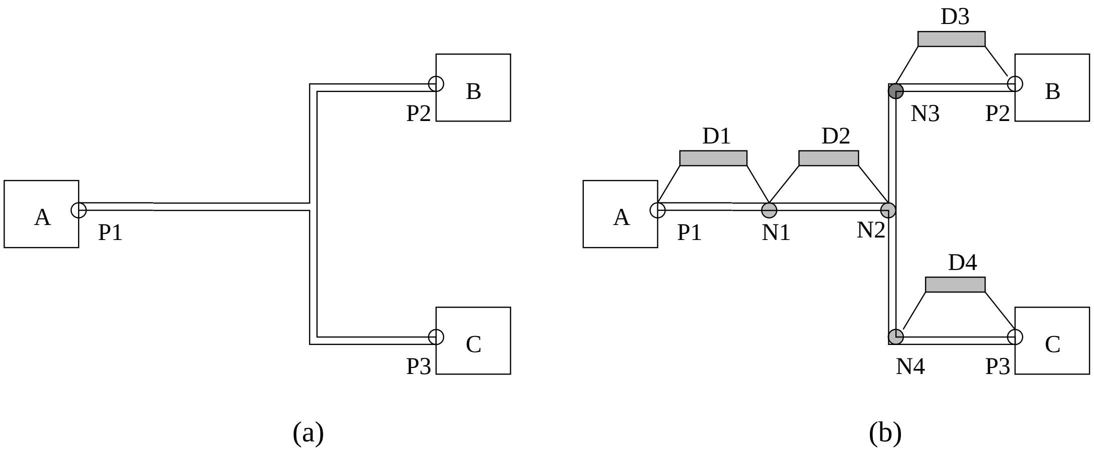


Figure 6.2: A sample design with parasitics

Parasitic effects are incorporated into a design in a three stage process which involves extraction of the parasitics from the circuit netlist, modeling them with a suitable model (for example, the RC-tree model) and incorporating them in the original circuit for analysis. The result of this exercise results in parasitic nodes and parasitic devices being incorporated into the original netlist. Parasitic nodes are logical points on the interconnect that serve as a endpoint for parasitic devices which are logical components introduced to model the effects of parasitics. Figure 6.2(a) shows the connectivity between cells A, B and C. The interconnect is represented by closely spaced parallel lines. Figure 6.2(b) shows the interconnect after parasitics have been taken into account. Parasitic nodes are represented by filled circles (N1,N2,N3,N4) while parasitic devices (D1,D2,D3,D4) are represented by filled boxes. Parasitic nodes which also form the ports of a cell are shown with unfilled circles (P1,P2,P3).

The data can be broadly classified under the following categories.

- *Netlist data*

  The netlist shows the electrical connectivity of the circuit. Design entities such as cells, ports, etc fall under this category.

- *Layout data*

  The layout related information of the design elements and the parasitic data fall under this category. Data stored include layout geometries, length and width of various metal layers, etc.

- *Characterisation data*

  The library cells used in the design have characterisation information. These are used in subsequent calculations for a design using them. The data include the output capacitance for different ports of a cell, etc.

- *Parasitics data*

  The DIAS system mainly deals with parasitic data. For each signal in the circuit, details about the interconnect model are stored in the database. Also, using the interconnect model, the values obtained for different parameters of interest such as slew at the output terminal, the delay due to the parasitic elements are stored in the database.

The DIAS data model captures the parasitic domain and uses the well known Rumbaugh notation to depict the model. The entire model is given in Appendix B. The interested reader may refer [Arun96] for a detailed discussion on the development of the DIAS data model.

6.4 Schema Manager

The schema contains information about all the types in the model. It reconstructs the schema defined by the model. This is necessary even though SDL constructs type information only because SDL has no knowledge of the semantics that link the attributes to their methods in a class. This information is important for ad-hoc querying as will become evident soon. The reconstructed schema is stored as a sequence of records with each record describing information related to an attribute of a class. Thus, the total number of records in the schema is equal to the sum of all attributes of all classes in the object model. A schema record template is shown in Figure 6.3.

User AttrName	System AttrName	Class Name	Collection Type	Set of methods	Other Information

Figure 6.3: A Schema Record

The record contains attribute name as defined in the object model by the designer of the system and the attribute name as referred to by a user of the system (these are represented by *System AttrName* and *User AttrName*, respectively, in Figure 6.3). The User AttrName allows the user the flexibility of using aliases to names of attributes of classes. It also contains the class to which the attribute belongs (referred to as *Class Name*). The *Collection type* of the attribute gives information about the collection that it qualifies. The attribute name may qualify a set of objects, or maybe a reference to another object or may refer to a base type. The *Set of Methods* field contains a set of method names that can be used to perform operations on the attribute. Some other information such as maximum attribute width may also be stored for purposes of formatting, etc.

The user can specify the attribute list and other information in the form of a table and an automated tool called the *schema loader* transforms this textual information and stores it in the schema section of the database. The Schema Manager reads this information from the database at runtime and stores it in a hash table structure in main memory to reduce overheads. The schema manager also updates information modified at runtime (for example, the maximum attribute width changing as a result of an insertion). The schema manager also maintains a query graph of the entire schema (which is necessary for semantic checking purposes at later stages) using the query model described in chapter 4.

6.5 Query Processor

This section describes in detail the construction of the query processor. Figure 6.4 shows the architecture of the query processor. The QUERY and the RESULT are input to and output from the query processor respectively.

The query processor takes as input the query in the form similar to OQL [Catt94]. The interface server transfers the query to the query evaluator which parses the query, creates a corresponding parse tree and evaluates the query for semantic inconsistencies with the help of the schema manager. The query evaluator discards the query if an error is found and informs the interface server. If the query is semantically correct, the query evaluator transfers the parse tree to the query executor which generates a flow, optimises it, and then executes it. The results of the query are transferred to the interface server which then displays the results to the user in a specific format. We now describe each of these components in greater detail.

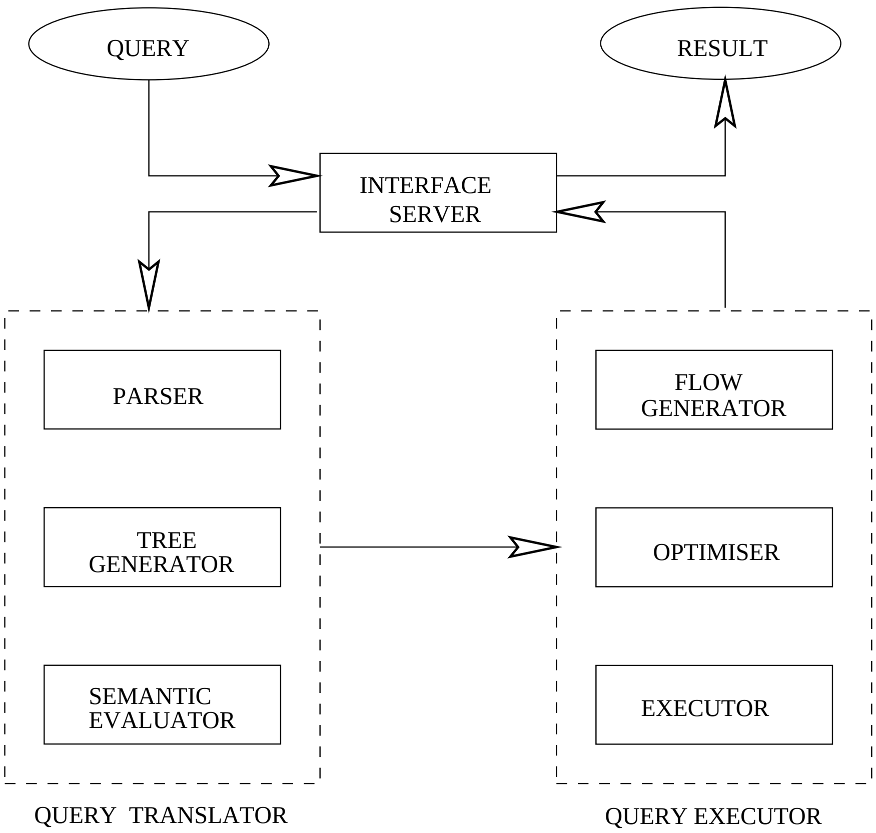


Figure 6.4: Query Processor Architecture

6.5.1 Query Language

We describe the query language used to represent a query and highlight some of its capabilities. The language specification is inspired by OQL and implements a subset of it. The query language built was however found sufficiently powerful for data manipulation in the parasitics database. In the following discussion, we refer to a path expression as *complex* if either its length is greater than 2 or if the length is 2 and the leaf node is not a base type, else we call the path expression *simple*.

A *predicate* is a triple of the form $< PathExpression, operator, value >$. The *operator* could be one of the relational operators $(>, <, >=, <=, ==, !=)$. The *value* could be one of integer, double or string. A predicate is *complex* if the path expression contained is complex, else the predicate is said to be *simple*. The language itself can be broadly defined as shown in Figure 6.5.

```
SELECT <PathExpression>, ... , <PathExpression>
FROM   <TopLevelObject> in <TopLevelClass>
WHERE ( <expression> OP ... OP <expression> );
```

Figure 6.5: Query Language Syntax

The language consists of the *SELECT* operator which performs the object projection operation, which is essentially extracting attributes from the objects. The *FROM* operator is used to define the toplevel class name which represents the type of the root node of the path expression. In DIAS, the toplevel class has been implicitly assumed for the sake of convenience. Thus, it is enough to specify the instance name without specifying its class. However, the *WHERE* operator is used to define predicates that the resultant objects of a query must satisfy. The *WHERE* clause itself can be complex and is abstracted in the figure as series of *expressions* separated by the *OP* operator (which could be an *AND* or *OR*). The *expression* can contain simple or complex predicates or both and an arbitrary number of them can be linked with the *AND* and *OR* operators. The *expression* clause can also contain other *SELECT-FROM-WHERE* operators thus forming a nesting hierarchy. Thus the *expression* maybe deep in two orthogonal directions – one dependent on the number of predicates, and the other dependent on the number of nesting level of expressions. An example query is given below and its specification is given in Figure 6.6.

QUERY

List the names of signals in the design FpuDesign which have a low to high transition delay greater than 20nS, and have greater than 4 terminals associated with it, and the estimated prelayout wire capacitance is less than 0.5pF.

For the following discussion, the path expressions in this query are referred to by symbols as shown below.

- P1 : *Signals.Sources.Timings.Transitions.LHValue*
- P2 : *Signals.NumTerminals*

```
SELECT Signals.NetName
FROM   FpuDesign
WHERE ( Signals.Sources.Timings.Transitions.LHValue > 20
              AND  Signals.NumTerminals > 4
              AND  Signals.PreWireCap   < 0.5
      );
```

Figure 6.6: An Example Query Specification

- P3 : *Signals.PreWireCap*

All the predicates contain the toplevel object (in this case, *FpuDesign*) implicitly specified for user convenience. With this observation, it can be seen that the predicates are all complex since the the path length of the path expression is greater than 2 in all cases. The grammar for the language can be found in Appendix A.

6.5.2 Interface Server

The interface server module is intended to provide a user-friendly interface. This module takes the input specification and transfers it to the query evaluator. If the query evaluation results in the query specification being found incorrect, the query evaluator signals the server to report the error. The interface server is also responsible for tasks such as interpreting the path expressions in the top level *SELECT* clause and applying them to the result objects of the query. The interface server, thus, iterates through the result objects of a query, and formats the output. The iteration process itself may need to traverse query graphs. This is necessary because dynamic typing is not implemented and hence object projection cannot be done by the query executor itself. The interface server, while performing the tasks mentioned, can be viewed to be a place holder for a more sophisticated interface in the future.

6.5.3 Parser and Tree Generator

The Parser module of the query evaluator is responsible for the parsing of the input query specification and the creation of a parse tree. The query language grammar itself is specified as input to a parser generator tool (yacc) and a parser is generated. The lexical analyser to feed the parser is generated using a token specification specified to the lexical analyser generator (lex). Thus, at runtime, the lexical analyser inputs tokens read from the query specification to the parser, which ascertains the correctness in syntax of the specification. Thus, syntax checking is done as part of the parsing operation. As the parser scans the input query specification, it builds internal data structures which transforms the input query into an internal format. The path expressions specified in the *SELECT* clause are stored as a set for each level of nesting. The information of the top level object name is also stored for each level of nesting in the query. The predicates which the resultant objects should satisfy are stored in the form of a parse tree. The parse tree is built by a series of operations which essentially entails building subtrees for each predicate and merging them with the corresponding operators for each level of nesting in the query. The parse tree built for each level of nesting is then combined into a single parse tree by a series of merge operations preserving the semantics expressed by the query. Thus, at the end of the parse operation, the nesting in the query specification is converted to a tree structure which enables easier processing of the query. However, the information about the level of nesting is maintained to aid semantic checking, propagating results up the tree, etc. Figure 6.7 shows the parse tree generated for the predicates in the query specification of Figure 6.6.

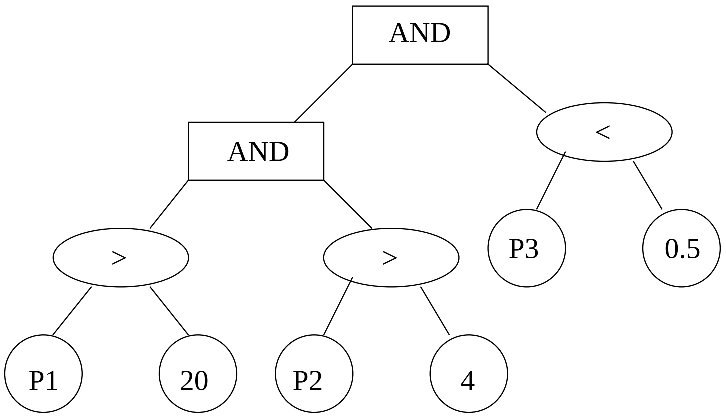


Figure 6.7: An Example Parse Tree

The Parse tree consists of three types of nodes. Leaf nodes (represented by rectangles) store either path expressions or values. The internal nodes represent operators and they could in turn be either the comparison operators (represented by ellipses) or logical operators (represented by rectangles). This differentiation is made because of the inherent difference in their implementations.

6.5.4 Semantic Evaluator

The semantic evaluator is responsible for evaluating the correctness of the query. The semantic evaluator module builds a query graph for its semantic checking. The query graph is built in stages. For each leaf node representing a path expression in the parse tree, a query graph is constructed. The query graph formed by the union of the query graphs of all the path expressions represents the query. The query graph constructed in this way for the example query specification of Figure 6.6 is shown in Figure 6.8.

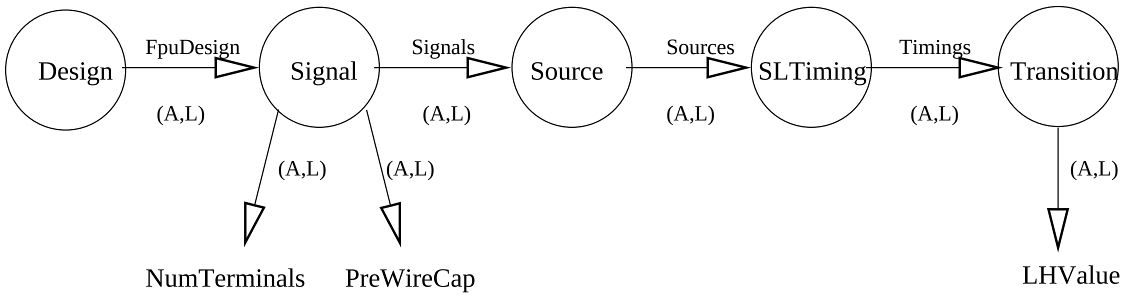


Figure 6.8: Query graph for specification in Figure 6.6

Once the query graph is constructed, semantic evaluation is initiated. Some example evaluation operations process are listed below.

- The paths specified by the query are checked for consistency with the schema. This verification is done by checking whether this graph is a subgraph of the query graph maintained by the schema manager. If it is not a subgraph, it implies that there exists classes that are referred whose definition do not feature in the schema.

- Explicit joins are disallowed by the query processor in its current implementation. Attempted explicit joins are detected by examining the query graph constructed for uniqueness of the root node of the subgraph representing a particular level of nesting. If two root nodes exist for a subgraph constructed for the level of nesting, an explicit join error is signaled to the interface server.

- A query graph for the path expression in the *SELECT* clause is constructed. If the intersection of this graph with any of the subgraphs representing the predicates in the query result in a null graph, a warning is generated for a possible semantic error wherein a constraint clause has been specified that has no effect on the result.

- The left hand sides in the predicates are allowed to have only path expressions and the right hand sides are allowed to have only base values. This is ensured by examining the predicates. Type mismatches between the left and right hand side of a predicate are also detected.

- Though the query language grammar allows for representation of multiple path expressions to be expressed in the *SELECT* clause, the system enforces only a single path expression to be specified due to the absence of dynamic typing.

- Relational operators are defined on only the base types and hence results of nested statements can only be base types in the current implementation.

6.5.5 Flow Generator

The flow generator module is responsible for implementing the order of evaluation of operators in the parse tree. The generator, thus, decides on the associativity of operations depending on the rules built into it. Normally, the predicates are evaluated left to right, but this order may need to be modified depending on the availability of operands. This situation is especially true if the parse tree is left deep or right deep which occurs when nesting of queries is involved. Thus, to summarise, the flow generator is responsible for deciding the order of evaluation of the subtrees (a subtree here is used to depict one operator node and its immediate children). In the current implementation, the flow generator generates the subtrees by detecting subtrees in a postorder fashion. Thus, for the parse tree of Figure 6.7, the subtree representing the predicate $< P1,>,20 >$ is evaluated first, followed by the subtrees representing $< P2,>,4 >$. The subtree rooted at the AND operator that is the root for both the predicates is then evaluated followed by the subtree representing the predicate $< P3,<,0.5 >$. The subtree at the root node of the parse tree is evaluated last.

6.5.6 Optimiser

The optimiser is responsible for speeding up operations at the internal nodes and currently concerns itself mainly with the logical operators. The optimiser thus deals with subtrees whose root is a logical operator. The operator uses logical OIDs of the operands, which are usually collections of objects. The optimiser first sorts the OIDs in the respective sets in the operand nodes using a sorting technique such as quicksort using the logical OID as the key to sort on. Once the lists are sorted, the optimiser implements the logical operation on the logical OIDs itself without requesting for the objects on disk. This operation on surrogates instead of the actual objects themselves can be implemented in main memory itself thus speeding up the evaluation of the query. The AND operator is implemented by performing a surrogate intersection of the sets that form the operands and the resultant set of identifiers are propagated up the tree. The OR operator is implemented by performing a surrogate union operation and the resultant set of identifiers propagated up the tree.

6.5.7 Executor

The executor module actually uses the other modules and executes the query. Each subtree generated by the flow generator is used to execute a portion of the query. The executor uses the optimiser to perform the optimisation operations when logical operators are involved. The executor uses the query graph constructed by the semantic evaluator to evaluate path expressions. The query graph is traversed node by node with method information supplied by the schema manager. Since the query graph nodes can contain parameterised types, the executor has to to be able to traverse portions of the query graph multiple times. Thus, the executor maintains information about the portion of the path to traverse repeatedly. For the example graph shown in Figure 6.8, the node representing the class *Signal* can be considered to be a set of *Signal*, since the attribute Signals is implemented as a set. Thus the executor has to repeatedly traverse the subgraph rooted at *Signal* in order to fetch the required values for all the signals in the design. The value fetched is then compared using the relational operator specified in the subtree. The query graph may also contain references which have to be handled as a separate case by the executor. These references however, do not cause repeated traversals of the query graph by themselves. Access methods are used by the executor if available for the path in consideration. The executor also decides from the select expression, the object in the path expression that needs to be stored as the result and which should be propagated up the tree. This is determined by marking the schema graph at the point at which the class representing the object to be propagated exists. In the case of the example of Figure 6.6, it can be seen that the marker would be placed at the node representing *Signal* in the query graph since these objects are required by the select expression to project the relevant attribute. This resultant object itself is determined by an intersection of the query graphs representing the select expression and the predicates at a given nesting level. The intersection gives the maximal path that is common to both the select expression and the predicates. The resultant objects obtained are transferred to the interface server which handles the output to the user.

6.6 Access Methods

The DIAS kernel implements some of the access methods described in chapter 5. The access methods were chosen after analysis of the pattern of possible queries that DIAS would be called upon to handle. In the current implementation, paths that need to be indexed have to be identified at compile time itself.

Simple Value Based access methods

To answer queries based on simple values (literals) the B^+ tree index provided by SHORE is used extensively. The trees were built on explicit attributes in the domain as well as some parameterised types such as sets used in the implementation. Since exact match queries were not expected to be frequent, extendible hashing was not implemented.

Aggregation based access methods

The DIAS model contains a number of aggregation hierarchies some of which are virtual hierarchies created through references in the model. The presence of virtual hierarchies are taken into account while deciding the paths to be indexed. For the DIAS domain, updates were expected to be infrequent, while retrievals were expected to be frequent. This is so because entire designs are loaded and then used by various tools to analyse them. Therefore, the bottleneck was expected to be in the retrievals. Hence, the nested index was chosen. However, the path index was also implemented as some paths in the graph required retrieval of intermediate nodes as well. It was felt that the path index would also cater to updates in the future in the absence of reverse pointers being maintained along paths in the database.

Inheritance based access methods

The hcC tree was implemented to address the issue of inheritance based queries in the DIAS database. This was done since the hcC tree performs well against both single class and class hierarchy queries which is a primary requirement since queries could be directed at a single class or a hierarchy in the DIAS domain.

We looked at the design of the query processing system in this chapter. Specifically, we looked at the interactions that existed among the query processor, the schema manager, the model implementation and the access methods module. In the next chapter, we discuss the implementation of the query processing system.

Chapter 7

Implementation Details

In this chapter, we discuss some implementation issues involved in the development of a query processing system.

7.1 Schema Manager

The schema manager has been implemented with the SDL interface provided by SHORE. The schema information consists of a set of records, each record maintaining information about an attribute. The SDL declaration for the schema is shown in Figure 7.1.

The Figure only shows the declaration of the schema interface and does not show the declaration of the SchemaRecord interface and other subsidiary interfaces required for the schema manager.

The schema loader takes as input a text file specification of the schema and loads the schema with the information. The loader consists of a parser which is generated by yacc whose grammar is trivial and hence not given. The loader does certain checks for consistency in the input and then enters the data into the schema by making calls to the SHORE server.

```
interface Schema
{
     protected :
                 attribute set<SchemaRecord> Records;
                                // set of SchemaRecords
     public    :
                 void Construct();
                                // the constructor
                 void AddRecord( in ref<SchemaRecord> Rec );
                                // add record to schema
                 ref<SchemaRecord> GetRecord( in short Num );
                                // get a record from schema
                 short GetNumRecs();
                                // the number of records
};
```

Figure 7.1: SDL declaration of the schema

7.2 Parse Tree

In this section, we describe the parse tree construction. The parse tree is constructed as the parser scans the query specification and is constrained by having no knowledge about the predicates that have not yet been read and hence about the nesting levels. Thus, the parse tree construction is done in stages for every level of nesting and then combined into a single parse tree while unrolling the nesting. The intermediate parse trees are stored in table structure called the *Parse Table*. Figure 7.2 shows a snapshot of a portion of the parse tree in construction. The figure also shows the contents of a node in the parse tree. The parse tree for every level is constructed by maintaining their constituents on a stack. The figure shows a list implementation of the stack.

The first column in the parse table contains a flag pair that is used to denote the state of each record in the table. The first flag, called the OperandFlag, denotes the operand that the parser has not yet seen for the current record entry. This could be the left operand (L), the operator (C) or the right operand (R) of a predicate. The second flag, called StatusFlag, denotes the status of the record entry. This denotes whether the

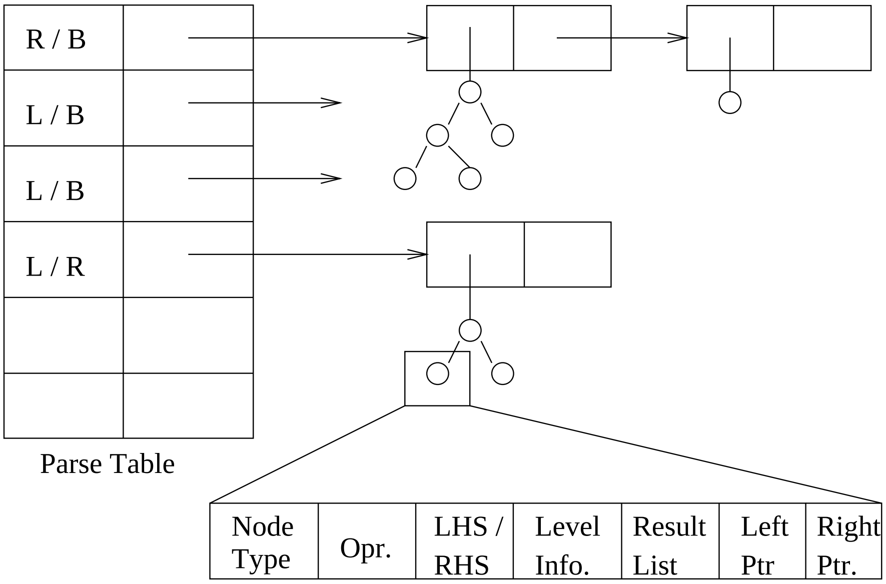


Figure 7.2: Snapshot of Parse tree under construction

record has a complete subtree for the current level (in which case it is said to be ready for a merge of levels and denoted by R) or an subtree that is waiting for a an operand to be evaluated at a higher nesting level (in which case it is denoted busy (B)). The second field contains a list of subtrees representing constructs seen by the parser until this stage. The example is of a query with a nesting level of 4. The figure, thus, shows the subtree at level 4 to be complete while that of levels 2 and 3 are waiting for a left operand, while that of level 1 is waiting for a right operand. At this point, the merge of levels 3 and 4 is done using the flags at level 3. A left node is inserted for level tree with the subtree from level 4 and parsing continues to find the operator for this operand and its right sibling. A merge of nodes at level 3 is done to get the subtree for level 3 and it is then inserted at the appropriate location at level 2. This operation continues till the parse is complete at the end of which only one tree at level 1 exists which is the desired parse tree for the query.

7.3 Query graph

The query graph representation had two main entities, the node and the edge. Thus, the implementation of the query graph contains two types of structures, one to represent the node and the other to represent the edge. The structure representing the node contains the class name, and a set of pointers to edges that it connects. The structure representing the edge has pointers to two nodes and has attribute and the symbol information as present in the query graph. This organisation, while being simple, facilitates the graph operations such as union, intersection and subgraph detection. These operations then represent simple traversal of these data structures and manipulation of the structures. Virtual edges and virtual hierarchies are detected with the help of the edge qualifiers and are incorporated into the query graph at the appropriate place. Scoping in queries is implemented by the traversal along the virtual edges.

7.4 Subtree generation

The Flow generator described in section 6.5.5 generates subtrees which are then used in the execution. The subtree generation algorithm is described in this section. The tree itself is built as a template, and has its corresponding tree iterator which is responsible for returning one node of the tree at a time following some order. The flow generator incorporates postorder traversal of the tree. This iterator is implemented in a non-recursive way which is necessary in order to be able to store the state of the traversal between consecutive calls to the iterator. The generator is thus responsible for generating subtrees rooted at an operator that have both its operands *ready*. For this purpose, a stack is used to contain the nodes. Figure 7.3 shows the function to compare two nodes while Figure 7.4 shows the algorithm used for subtree generation. The PUSH and POP represent the familiar stack operations, while the NEXT-TREE-NODE represents the iterator operation.

```
Function CompareNodes: args Node1, Node2
begin
        if Node1.parent = Node2.parent // if parent of the two nodes
             Marker = NEXT-TREE-NODE   // are equal, the next node in
             PUSH Marker               // the post order traversal is
        else                           // the parent node and a subtree
             PUSH Node1 Node2          // is generated, else the nodes
             Marker = NONE             // are stacked till the siblings
        return Marker                  // are found.
end
```

Figure 7.3: Node comparison for siblings

7.5 Optimisation

The optimisation in the logical operations is performed by the implementation of the result list as a linked list. The linked lists representing the sets of the two children of the subtree being evaluated are sorted on the basis of their logical OIDs. The AND and OR operators are implemented using the union and intersection operation, respectively, on the linked lists. The resultant object's OIDs are stored in the result list of the root of the subtree and propagated up the tree. A further optimisation is done if the operator is an AND. In this case, when the subtree to be evaluated has a parent as the operator AND and its left tree has been evaluated, the subtree evaluation process is done on the objects obtained from the left tree evaluation.

7.6 Execution

The execution of the subtree can be broadly classified as those that need traversal of the query graphs and those that do not. The first set of subtrees form the leaf nodes of the tree and hence contain path expressions that have to be traversed in order to retrieve objects. The second set of subtrees essentially have the objects at their nodes and need only to operate on them. We describe the evaluation of a subtree involving the query graph traversals. Consider the query graph example of Figure 6.8. The *Signal* class node

```
Algorithm GenerateSubTree
begin
        if NumStackedNodes >= 2            // Compare the nodes on
             Marker = CompareNodes POP,POP // the stack. This condition
             if Marker != NONE             // is necessary to check
                  return Marker            // for internal nodes forming
        Node = NEXT-TREE-NODE              // a subtree.
        if Node = NONE
             return NONE                   // the termination condition
        PUSH Node
        while TRUE
             Marker = CompareNodes POP, NEXT-TREE-NODE
             if Marker != NONE             // Iterate till a subtree is
                  break                    // found by finding two
        return Marker                      // consecutive nodes with the
end                                        // same parent.
```

Figure 7.4: Subtree generation algorithm

represents a set and so does the *SLTiming* class. The number of path instantiations that need to be traversed due to these two sets can be seen to be the product of the instances of these two sets. Thus, the query executor has to be able to dynamically traverse all paths instantiations which implies that it should be able to generate all the instantiations along any query graph. It can be seen that the generation of all the instantiations is a correctness issue. To perform this operation, the query traversal is implemented by maintaining markers at each set valued node and generating paths exhaustively. The algorithm shown in Figure 7.5 accomplishes this by iterating over the members of a set while using recursion to traverse the query graph whose length can vary with each query.

The algorithm shows the core of the execution steps executed on a single subtree. The information about the methods to be called to obtain the relevant information at each stage is obtained from the schema manager. Once the methods are obtained, they are executed by dynamically binding them. The result of the method execution is then used for further processing.

```
Function TraverseGraph: args Marker
begin
         while Marker != NULL
               if Type = BASE
                    break
               else if Type = REF
                    Use the REF to get next object using dynamic binding.
                    Store the result object.
               else if Type = SET
                    Num = Cardinality of set using dynamic binding.
                    Set Marker to current node.
                    for I:1 to Num
                         Get Object[I] using dynamic binding.
                         TraverseGraph Marker
                         Restore Marker
         Handle Base type comparison
         Compare query graph of select expression with that of predicate.
         Store the relevant object of the current path in result list.
end
```

Figure 7.5: Query graph traversal algorithm

7.7 Heterogeneity

The problem of handling a heterogeneous collection of objects by a query was also addressed. This problem was overcome by including an abstract base class from which all other classes inherit. This inheritance provided the answer to referring to any object in the database with the reference to the base class. Thus, the result list consists of a list of references to the base class. Since references to the inherited classes are compatible with those of the base class, the list itself could consist of heterogeneous collections of objects. This inheritance also allowed for the dynamic binding of methods wherein they were declared virtual in the base class. Thus, it was possible to operate on objects in the result list, wherein dynamic binding of methods was used for data manipulation.

7.8 Other Details

The query processing system has been built in the C++ programming language. The schema manager and the access methods module use SDL for the declarations and C++ binding provided by SHORE to define the methods. Template classes, which are container classes for objects of other types, are implemented for the linked list, the tree and the stack. Extensive use of operator and function overloading has been made throughout the implementation. Dynamic binding features of C++ were employed to provide ad-hoc query capability which uses runtime binding with information about the methods being obtained from the schema manager.

Chapter 8

Bulkloading in OODBMS

8.1 What is BulkLoading?

Bulkloading is the process of mass-populating a database with collected data. As more and more applications move to object oriented systems, bulk loading in object oriented systems has assumed importance. Bulk loading techniques available in conventional database systems cannot be used since the techniques rely on properties found in those systems which are absent in object oriented database systems (for example, a primary key in relational databases). Thus, new techniques need to be evolved to provide efficient bulk loading in object oriented database systems. Some algorithms maybe found in [WN94].

The data in question maybe in any user defined format and maybe textual or binary in nature. The bulkloader thus needs to have a knowledge of the input format and the corresponding database format to represent the information. Sometimes, bulkloading also involves simultaneous creation of certain indices for efficient access and may also incorporate semantic data checking depending on its sophistication. Though the term bulkloading suggests a unidirectional movement of data from files into the database, the term is also used to mean the reverse process of restoring a data file from the database.

8.2 Necessity

- Object Oriented Database users are users whose problem domains are complex thus needing powerful modeling techniques in order to capture the domain in a natural and efficient manner. Until recently, in the absence of object oriented database systems, users had to use conventional systems for their data management which proved inadequate for a variety of reasons. However, vast amounts of important data were generated in this period which cannot be discarded or regenerated. Hence, it becomes important for such users to be able to move all the legacy data into the object oriented systems that is better suited for the problem domain.

- Most of the current object oriented database systems do not provide an explicit bulk loading facility and database population is done with the help of statements in the database manipulation language itself. This is a good solution for database population of small volume data but needs more specialised loading mechanisms if the data to be loaded runs into megabytes of objects of data. Certain optimisations to access data can be done which may speed up subsequent query processing (for example, indices maybe built on large set valued types automatically). Additionally, a load utility can group certain operations, such as integrity checks, to dramatically reduce their cost for the load [Moha93].

- Users of object oriented database systems sometimes need textual dumps of data or portions of the data. This maybe done for automated debugging which operate on existing file based formats. It maybe impossible in the short term to create automated debugging facilities on the database itself and also due to the fact that many such tools may require to be rewritten.

- Objects may need to be reclustered for performance reasons. There may not be an efficient way to recluster online especially if the database uses physical object identifiers which map to disk addresses. In such cases, it might be easier to dump the objects in the order to be reclustered and as such reclustering can be done on a reload.

- Users may feel the necessity to ascertain the accuracy and correctness of the data moved to the OODBMS. This especially gains importance during development and initial installation of an OODBMS. A textual dump often serves the purpose in a simple and convincing manner which otherwise would have to be ascertained by queries on portions of the database.

8.3 Issues in bulkloading in OODBMS

- *Surrogate identifiers*

  Objects are referenced with the help of object identifiers which are independent of the semantics of the data contained in the objects. This is in contrast to the situation in relational databases where each tuple is identified by a primary key and foreign keys are used to depict relationships. Such keys are part of the data itself and as such are used during bulk loading to generate relationships. However, in OODBMSs, relationships are represented through object identifiers. Thus, the relationships in the data which is to be loaded must be represented by alternate means since object identifiers are generated by the database at object creation time only. Thus, such relationships are represented with the help of *surrogate* identifiers which essentially are pointers to objects in a relationship. These pointers themselves maybe represented as strings or integers.

- *Inverse relationships*

  Inverse relationships refer to the presence of bidirectional relationships between objects. This implies a dependency from an object to another and necessitates update of one if the other is updated. Such relationships are often created by OODBMS and as such needs to be handled during the bulk loading of data. The problem in handling such relationships is that, during sequential creation of objects from the data file, one object involved in the relationship may need to be created which references another object not yet seen by the bulk loader. The object not seen yet also needs to be intimated about the relationship. This maybe solved by immediately creating an instance of the object not seen and updating only its inverse relationship. However, this leads to performance problems.

- *Circular relationships*

  Objects maybe involved in a circular relationship which maybe a direct or indirect relationship between two objects. A special case of a direct circular relationship is an inverse relationship. Apart from resolving surrogate identifiers as in the case of inverse relationships, the bulk loader must be able to break cycles while scanning the input data.

- *Implicit relationships*

  Coupled with the above problems, there may also be some text based data files that store data having a definite and well defined structure wherein relationships are represented implicitly. For example, the levels of nesting in the representation of one object with respect to another may define a containment relationship between the two. Such relationships are referred to as implicit relationships. Implicit relationships also include implicit inverse and implicit circular relationships.

- *Memory Constraints*

  The bulk loading of an OODBMS is not a value adding activity in a commercial setup. Hence, it maybe an activity forced to run in parallel with other value adding activities (for example, a SPICE analysis of a VLSI circuit). Being a low priority job, it maybe forced to limit memory usage in order not to slow down other memory intensive jobs. This puts the additional burden on the design of the bulk loader to be able to perform efficiently with limited memory availability. Thus, the bulk loader may require direct control on the buffer management policies of the OODBMS.

Some loading algorithms which resolve surrogate identifiers and inverse relationships can be found in [WN94]. The algorithms deal with the problem of surrogate identifiers and inverse relationships using one or two passes of the data file. These algorithms deal with the problem in one of three ways: The two pass algorithms create the objects in one pass and populate them in another pass. The one pass algorithms, on the other hand, use todo-lists wherein object identifiers of objects involved in relationships with objects not seen yet are stored and updated at a later stage. One pass algorithms also exploit logical object identifiers which can be obtained from the database storage manager before the actual object is created. Variants of this updation process exist taking into account various parameters like clustering. However, these algorithms do not take into account the presence of implicit relationships.

8.4 The DIAS BulkLoading Architecture

The DIAS bulkloading architecture is discussed in this section. The problem is to populate the DIAS database system with parasitic information available in a file based format. The files containing the information are textual in nature and employ a definite grammar structure for compact representation. The architecture of the DIAS Bulkloader is shown in Figure 8.1. We discuss the components of the bulkloader in this section.

8.4.1 DIF Data File

The input text file containing the parasitic information, also called Delay Interchange Format (DIF) file, contains parasitic information represented by a well defined grammar. It is mainly divided into two sections - the *CELL* section and the *DESIGN* section. Each DIF file represents information about a particular design.

The CELL section gives the list of all library cells used in the construction of the design and their associations and characteristics. A portion of a sample DIF file has been extracted and shown in Figure 8.2. Though the figure is by no means complete, it serves to demonstrate various aspects of the DIF file. The figure shows a portion of the CELL section.

In the DIF file, actual data is generally preceded by tokens which serve to qualify the data. This has been done due to the presence of variable length data against every such qualifier and hence making it difficult to make the qualifier implicit. Examples of tokens

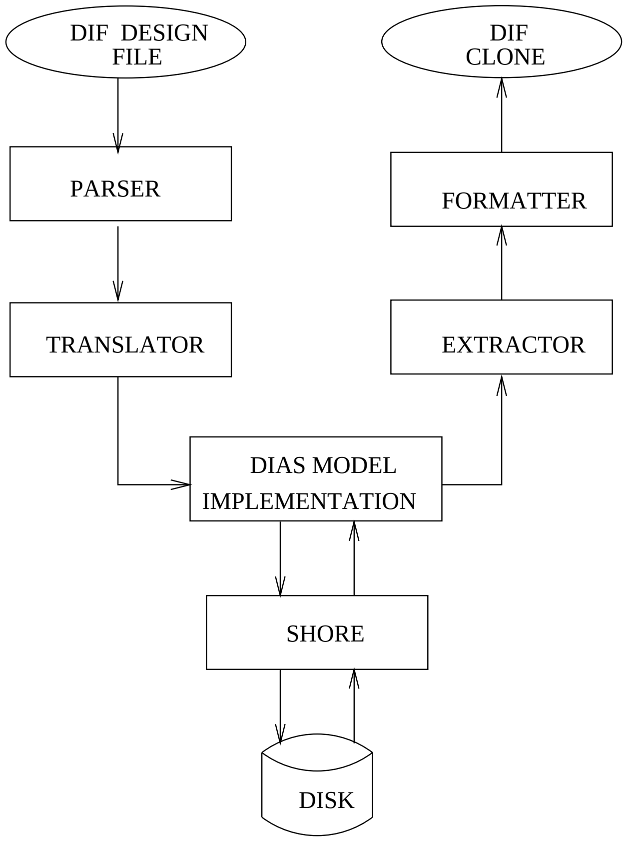


Figure 8.1: Bulkloader Architecture

as seen in Figure 8.2 are CELLS which denotes the CELL section, PORTS which denote the ports of a cell. The NAME token denotes a name or names and depending on context could denote the name of a cell, port, or other named objects. The CAP token denotes values following it are capacitance values whose units are to be read from a portion of the DESIGN section. The DIRECTION token qualifies the type of port.

Each cell is identified by the occurrence of a NAME at the topmost level of nesting. The name string itself serves as a surrogate identifier for any future references to the cell object. Each cell contains information characterising it. In this example, only the PORT information has been shown. The fact that cells are associated with ports is represented by a implicit relationship defined simply by the level of nesting. Each port also has a name which is identified by the occurrence of the NAME token and is differentiated from the NAME token of the cell name again by the level of nesting. Associated with each port is information about the port such as its direction (for example, an input port), and the loading capacitance represented here as a triplet of values which represent the minimum, nominal and maximum values. The existence of many ports is recognized again with the level of nesting involved when the NAME token is read. The cell and port names are usually defined by the designer during the design phase itself.

```
(CELLS
  (NAME "S3and2_a_BC"
    (PORTS
      (NAME "in[0]"
        (CAP 0.031823 0.031823 0.031823)
        (DIRECTION INPUT))
      (NAME "in[1]"
        (CAP 0.031823 0.031823 0.031823)
        (DIRECTION INPUT))
      (NAME "out"
        (DIRECTION OUTPUT))))
  (NAME "S3inv_a_CC"
    (PORTS
      (NAME "in"
        (CAP 0.025934 0.025934 0.025934)
        (DIRECTION INPUT))
      (NAME "out"
        (DIRECTION OUTPUT)))))
```

Figure 8.2: Sample DIF structure

The DESIGN section, which is not shown in Figure 8.2 contains information about the design itself. This section contains information about various parameters used in the design (for example, the units used for capacitance values). It also contains all the signals in the design. Each signal contains information about the terminals involved, and the *rctree* associated with it which is represented by a series of parasitic nodes and the parasitic values between consecutive nodes. Each signal also contains information about the timing delays associated with each source terminal and also the timing delays between each source and load pair. The representation is similar to the CELL section in that nesting is used to specify implicit relationships both inverse and circular. The surrogate identifiers used can be both names (for example, signal name) and numbers (for example, source port number).

8.4.2 Parser and Translator

The parser itself consists of a lexical analyser and a parser generator built using lex and yacc tools respectively. The lexical analyser feeds the parser generator with tokens and the parser generator does semantic checking using the defined grammar and produces the corresponding C file to be used in the parsing. During parse, the data is extracted from the given DIF data file and stored in main memory data structures. The main memory data structures were used to facilitate a certain amount of parallel development with the DIAS API and also to have control on the order in which the data objects are clustered on disk which is now the order in which objects are converted to persistent objects.

The *translator* module, which forms the core of the bulk loader, actually does the semantic evaluation and conversion from surrogate identifiers to object references. It includes the handling of implicit inverse and circular relationships. The module actually uses information gleaned from the level of nesting of an object to create the corresponding references in DIAS. The major issues addressed by the translator module are dealt with in the next section.

8.4.3 DIAS API and SHORE

The DIAS Application Programming Interface (API) was successfully used and hence tested in the bulk loading process. Though the DIAS API was used extensively, transaction management and other transaction related issues were handled in the bulk loader itself to facilitate reconfiguration, if it becomes necessary, as for example, change of working environment with decreased main memory. The main memory data structures were converted to persistent objects as specified by the DIAS Object model and the necessary references and links created in the process. Some system maintained links not existing in the data but required by DIAS for efficiency reasons were also created in the translation. SHORE was used to ultimately create and maintain the object repository.

8.4.4 Extractor and Formatter

Building elaborate printing mechanisms was required for the reverse process of extracting data from the database and printing it in the format from which it was initially loaded. This required the creation of surrogate identifiers as it existed in the original DIF file. Thus, it was necessary for the bulkloading system to understand which of the attributes actually are used as surrogate identifiers (for example, SignalName in the SIGNAL class). In other cases where no attributes represented the surrogate identifier, and where an integer was used to specify the surrogate, the system had to generate the identifier under the constraint that it match the surrogate identifier in the original DIF file. The problem was handled by using an ordering mechanism on the non-attribute surrogates which was then used in the reverse process to extract, by position, the surrogate to be used for a particular object.

The inverse and circular relationships had to be reconstructed as implicit inverse and circular relationships in the data file. Thus, inverse and circular relationships had to be identified and the technique of using levels of nesting used to specify such relationships. Most importantly, it required the understanding of *not* outputting systems maintained references for efficiency which was not part of the original DIF file.

The formatter module actually converts the semantic information that was recreated by the extractor module and takes care of formatting. Formatting here includes depiction of the nesting levels and adding the necessary tokens to qualify the data obtained. It was important to recreate the original DIF file without error either in a representation of a character or in the relative position of a attribute with respect to another. This was necessary to facilitate future comparison of the original and the clone DIF files to easily ascertain the correctness of the entire process of bulkloading.

The output of this module was helpful in evaluating the correctness of data loaded into the database and also to evaluate performance.

8.5 Handling Bulkloading Issues in DIAS

The issues raised in section 8.3 have been handled effectively in the construction of the DIAS Bulkloader.

The problem of surrogate identifiers was handled during the translation of main memory data structures into persistent objects. The absence of explicit forward references through surrogate identifiers meant the loader had to handle references only to existing objects. Since references to objects created exist, creating back references posed no problem. The DIF file, in general, did not exhibit explicit inverse references.

Though explicit forward references and explicit inverse references were not encountered, a large number of implicit references, some of which were nested, existed which had to be transformed into object references. Implicit relationships were translated using a virtual stack of object identifiers which formed top level objects at different nesting levels. The implicit relationships most often carried an implicit inverse relationship and hence the inverse references also had to be created. This required updating an object already existing thus causing the necessity of a read operation to fetch the object. The other alternative is to pin the objects at different levels of nesting that are currently active. However, with the nesting levels not being deep, it was felt that performance levels will not be significantly affected without pinning objects to memory.

The presence of implicit forward and inverse relationships create implicit circular relationships which are broken taking advantage of the fact that nesting brings about the relationships. That is to say, the recursion is broken at the object at the top most level of the nesting.

Memory requirements and constraints are taken care of with the help of a user modifiable resource file which controls the extent to which the bulk loading problem maybe broken into depending on current system constraints. Though the bulk loading process can normally be done in single long transaction, it has been implemented as a series of transactions in the DIAS bulkloader and hence users can control the length of the transaction thus having a virtual control over the buffer pages used by the database. Users can specify the intervals at which translation should take place into persistent objects or vice versa and also the intervals (in number of signals) at which the transaction should commit thus flushing its buffer pages.

8.6 Bulkloading Performance

The DIAS Bulkloader was used to populate DIAS with various designs obtained from tool generated DIF files. DIF files in the range 50kB to 12MB were loaded which created databases sizes varying from 300kB upto 80MB. A sparc20 machine running solaris 2.4 having 64MB main memory was used in the evaluation of the bulk loader. The periodicity of transaction commit was fixed at 100 signals.

Design	FileSize(MB)	Terminals	Signals	Load Time	Dump Time
Design1	0.6	150	58	0:10	1:05
Design2	1.3	542	212	2:09	4:08
Design3	2.0	945	349	3:24	8:52
Design4	3.1	1583	542	7:16	16:11
Design5	5.2	2830	889	14:00	30:10

Table 8.1: Load and dump times for varying design sizes (in minutes:seconds)

In this experiment, we investigated the time taken to load data from legacy text-based design files into the DIAS database, and the time taken to subsequently dump (retrieve) the entire design contents from this database.

We evaluated the load time and dump time for different designs that represent varying degrees of complexity. These results are presented in Table 8.1, which shows the designs, the size of the text files that contain their parasitic information, the number of terminals, the number of signals and the corresponding load and access times. From these results, we observe that the load and access times increase in an approximately linear fashion with design complexity, thereby indicating that the DIAS system scales reasonably well. It maybe observed that the access times are higher than the corresponding dump times which is due to asynchronous I/O in the cases of writes.

Chapter 9

Conclusion

The modeling inadequacies of relational systems saw the emergence of object oriented database systems which came with a rich set of modeling constructs. These powerful constructs also brought with them new challenges to query processing in these systems. Query processing in object oriented systems is still being understood. In this project we tried to understand some of the problems and developed a system that deals with them. We conclude by summarising our contributions, describing the special features of the query processing system and possible avenues for future work.

9.1 Summary

In this project, we dealt with query processing in object oriented systems in general and the DIAS system in particular. We identified the differences between query handling in relational and object oriented systems with a view to understand the additional complexity introduced by object orientation. We identified issues that need to be addressed while designing a query processing system which were broadly classified under query model, formalisms, optimisation, access methods and a query language. We concentrated on the issues of modeling queries, providing a query language and providing associative access with the help of access methods.

A query model was then defined which incorporated some features of the model by [Kim89]. The model was extended to include simple associations and a means to tackle scoping in queries directed at inheritance hierarchies. The query model was used to build query graphs and operations on them. The semantics of the query were translated to operations on the query graph. A query language on the lines of OQL was also developed to provide the user with a declarative query interface for ad-hoc querying. The grammar for the query language was written taking into consideration basic necessities of object oriented systems.

A detailed survey of access methods was made to evaluate the available associative methods of access. The survey included the nested index, path index, multi index and multi index with path configuration for queries directed at aggregation hierarchies. The CH tree, H tree and the hcC tree were analysed for providing associative access for queries directed at inheritance hierarchies. B^+ trees and extendible hashing were analysed as methods to aid simple value based queries.

Using the query model and some of the access methods, a query processor was built. The main aim of the query processor was to provide declarative query processing capability to any application running on SHORE. DIAS can be considered to be an application using the query processor. The query processor consisted of various modules interacting with each other to provide ad-hoc query capability. They included the parser, the tree generator, the semantic evaluator, the flow generator, optimiser and the executor. Schema independence was provided by the schema manager which maintained the application's schema information. The access methods implemented were the nested index, path index and the hcC tree.

As an orthogonal problem, we also addressed the problem of bulkloading in object oriented database systems. Several issues in bulkloading in object oriented systems were addressed in building the bulkloader for the DIAS system to load the database with real data from design tool generated files.

9.2 Special Features

- The query processing system is built with the idea of providing a general query interface for applications built using SHORE. The system is, thus, dependent on SHORE but independent of any particular schema. This allows the system to be used on any application built using SHORE. This is facilitated by the schema manager.

- The system provides a OQL-like query language interface and provides ad-hoc query capability. The query interface implements basic features of OQL, and is sufficiently powerful to express queries with multiple constraints. Nesting of queries is also permitted. The system also handles parameterised types in an efficient manner.

- The query processing system uses the query model to model queries thus providing a concrete means to express the semantics of a query. The query model helps in the representation of some of the operations of query processing as a combination of operations on the query graph.

- The query system implements specialised access methods. B^+ trees provided by SHORE are used to evaluate simple value based queries and also to index collections of objects. The nested index and path index are used to facilitate queries on aggregation hierarchies while the hcC tree is used to index inheritance hierarchies. The system also provides surrogate intersection as a means of optimisation while evaluating queries to achieve a significant speedup.

9.3 Future work

Aggregate Operators

The query system, while providing ad-hoc query capabilities, does not implement aggregate operators such as min, max, avg, etc. These operations are well defined for the base types but may not be defined on user defined types. Also, the query system does not implement sorting and grouping which could be used to implement the aggregate operators. Grouping is easily accomplished by first sorting the input objects. The sort operation itself maybe costly in the absence of B^+ tree indices with chained leaf nodes maintained on the attribute used as the key for the sort. The sort operation in databases is generally implemented by merging where the input data are written into initial sorted runs and then merged into larger and larger ones till no more merges are possible. Aggregate operators may return scalar values or a set of scalar values. The set of scalar values results from grouping similar items and has to be handled.

Methods in the query language

The query system currently has a facility to express queries based on attributes specified in the schema. Methods are not captured by the query language and subsequent modules in the system. This additional feature of being able to bind methods to the language would help in processing information within the query itself. This requires extensions to the query language and additional information about methods to be stored in the schema manager. Handling the results of such a method call is an issue that needs to be addressed. Also, type checking becomes more complex.

Dynamic typing

Dynamic typing can been implemented which enables the creation of new types at runtime. This facilitates storing of the results of a query in the database which is useful if they were to be used as operands in a query at a later stage. Dynamic typing involves the creation of new types with exactly the same properties that they might have if they were statically created. This implies preservation of properties such as encapsulation. A possible implementation could be on the lines of dynamic typing built in ENCORE [ZM91]. Another issue to be addressed is the insertion of the resultant type in the schema which should preserve the properties of relationships that already exist. For example, it is necessary to ensure that insertion of a new type in an inheritance hierarchy would actually preserve the semantic meaning of the hierarchy and inherit the attributes of its ancestors.

Updates

The query system in operation currently is primarily a retrieval engine. A useful extension would be to provide ad-hoc update facility. However, several issues need to be addressed in its implementation. Versioning may be necessary if updates are allowed. Stricter type checking is required to preserve the semantic consistency in the database. The query language needs to be extended and so also the schema manager. The complexity of methods to be handled by the schema manager increases and hence the sophistication of the schema manager.

Formalisms

Formalisms need to be defined for operations on objects and the subject of optimisation of queries needs to be considered in more detail. A rule based optimisation process based on cost functions for plan generation would be an asset to the system. This would require definitions of operators, their properties and the relationship with the operands.

Compiled queries

The query system does not provide for the incorporation of compiled queries into the system. A useful extension would be to build a optimiser generator on the lines of the EXODUS optimiser generator. This facility would provide the system with prechecked, optimised queries which results in faster response time which is a requirement in most commercial applications. This requires an accurate estimation of cost of execution of methods. The query system now needs to understand the cost functions and incorporate them during the optimisation phase. The Volcano optimiser could be used to build such a system.

Performance

The performance of the query processing system needs to be evaluated. Performance metrics need to be identified, representative query patterns generated and existing systems identified for comparison.

Parallel execution

Parallel query execution is another area which could improve the performance of the system considerably. Though SHORE does not provide for parallel execution in its current version, we feel that a certain measure of parallelism can be built into the system with the help of other public domain tools currently available such as Parallel Virtual Machine (PVM) [Sund90].

Thus, we believe that the project served to understand the area of query processing in object oriented database systems and provide a closer look at the problems involved. This project brought together ideas of various researchers in the field and attempted to provide a efficient query processor based on these ideas. The query processing system that resulted from that effort is sufficiently powerful as a retrieval engine. With the incorporation of the ideas presented above, we believe the query system is a step towards addressing the query processing issues in object oriented database systems.

Bibliography

[Arun96] N. Arun, "Database Management for VLSI interconnects", *MSc(Engg) Thesis draft, CSA Dept, Indian Institute of Science*, in preparation.

[BK85] D. Batory, W. Kim, "Modelling Concepts for VLSI CAD Objects", *ACM Trans. on Database Systems*, March 1985.

[BK89] E. Bertino, W. Kim, "Indexing techniques for queries on nested objects", *IEEE Trans. on Knowledge and Data Engineering*, 1989.

[BKK88] J. Banerjee, W. Kim, K. Kim, "Queries in object oriented databases", *Proc. of the 4th Intl. Conf. on Data Engineering*, 1988.

[BZ87] T. Bloom, S. Zdonik, "Issues in the design of an Object Oriented Database Programming Language", *Proc. of the 2nd OOPSLA*, 1987.

[Bato+90] D. Batory, et al., "The GENESIS Extensible Database System", In *Readings in Object-Oriented Database Systems*, Morgan Kaufmann, 1990.

[Bert94] E. Bertino, "Index Configuration in object oriented databases", *The VLDB Journal*, 1994.

[Booc94] G. Booch, *Object Oriented Analysis and Design with Applications*, Benjamin Cummings, 1994.

[CDN93] M. Carey, D. DeWitt, J. Naughton, "The OO7 benchmark", *CS Tech. Report, Univ. of Wisconsin-Madison*, 1993.

[CM84] G. Copeland, D. Maier, "Making Smalltalk a database system", *Proc. of the ACM SIGMOD Conf. on management of data*, 1984.

[Care+90] M. Carey et al., "The EXODUS Extensible DBMS Project: An Overview" In *Readings in object oriented database systems*, S. Zdonik, D. Maier, eds., Morgan Kaufmann, 1990.

[Catt94] R. Cattell, *The Object Database Standard: ODMG-93*, Morgan Kaufmann, 1994.

[Come79] D. Comer, "The ubiquitous B-tree", *ACM Computing Surveys*, 1979.

[Deux90] O. Deux et al., "The Story of O_2", *IEEE Transaction on Knowledge and Data Engineering*, 2(1), 1990.

[FMV94] J. Freytag, D. Maier, G. Vossen, *Query Processing for Advanced Database Systems*, Morgan Kaufmann, 1994.

[Fagi+79] R. Fagin et al., "Extendible hashing - A fast access method for dynamic files", *ACM Trans. on Database Systems*, 1979.

[Grae93] G. Graefe, "Query Evaluation Techniques for Large Databases", *ACM Computing Surveys*, 25(2), 1993.

[HFLP89] L. Haas, J. Freytag, G. Lohman, H. Pirahesh, "Extensible query processing in Starburst", *Proc. of the ACM SIGMOD Intl. Conf. of management of data*, 1989.

[Haas+90] L. Haas et al., "Starburst Mid-Flight: As the Dust Clears", *IEEE Trans. on Knowledge and Data Engineering*, 2(1), 1990.

[KGBW90] W. Kim, J. Garza, N. Ballou, D. Woelk, "Architecture of the ORION Next-Generation Database system", *IEEE Trans. on Knowledge and Data Engineering*, 2(1), 1990.

[KKD89a] K. Kim, W. Kim, A. Dale, "Cyclic Query Processing in object oriented databases", *Proc. of the 5th Intl. Conf. on Data Engineering*, 1989.

[KKD89b] W. Kim, K. Kim, A. Dale, "Indexing Techniques for object oriented databases", In *Object-Oriented Concepts, Databases and Applications*, W. Kim, F. Lochovsky, eds., Addison-Wesley Publishing company(ACM Press), 1989.

[Khan91] S. Khanna, "Sorting out Signal Integrity", *Electron. Engin. Times*, June 1991.

[Kim89] W. Kim, "A model of queries for object oriented databases", *Proc. of the 15th Intl. Conf. on VLDB*, 1989

[LLOW91] C. Lamb, G. Landis, J. Orenstein, D. Weinreb, "The Object Store Database System" *Comm. of the ACM*, 34(10), 1991.

[LOL92] C. Low, B. Ooi, H. Lu, "H trees: A Dynamic Associative Search Index for OODB", *Proc. of ACM SIGMOD Intl. Conf. on management of data*, 1992.

[Litw80] W. Litwin, "Linear Hashing : A New Tool for File and Table Addressing", *Proc. of the 6th Intl. Conf. on VLDB*, 1989

[MS86] D. Maier, J. Stein, "Indexing in an object oriented database", *Proc. of IEEE Workshop on Object-Oriented DBMSs*, 1986.

[Moha93] C. Mohan, "A Survey of DBMS Research Issues in Supporting Very Large Tables", *Proc. of the Foundations of Data Organization and Algorithms*, 1993.

[NHS84] J. Nievergelt, H. Hinterberger, K. Sevcik, "The Grid File : An Adaptable, Symmetric Multikey File Structure", *ACM Trans. on Database Systems*, 9(1), 1984.

[RP94] C. Ratzlaff, L. Pillage, "RICE: Rapid Interconnect Evaluation using AWE" *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, June 1994.

[RS87] L. Rowe, M. Stonebraker, "The POSTGRES Data Model", *Proc. of the 13th Intl. Conf. of VLDB*, 1987.

[Rumb+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice-Hall, 1991.

[SS94] B. Sreenath, S. Seshadri, "The hcC-tree: An Efficient Index Structure for object oriented databases", *Proc of 20th Intl. Conf. on VLDB*, 1994.

[Stro94] *The C++ Programming Language*, Addison-Wesley, 1994.

[Sund90] "PVM : A framework for parallel distributed computing", *Concurrency: Practice and Experience*, 2(4), 1990.

[Vidy95] R. Vidya, "OSHADHI: A Biodiversity Information System", *ME Thesis, CSA Dept, Indian Institute of Science*, 1995.

[WLH90] K. Wilkinson, P. Lyngbaek, W. Hasan, "The Iris Architecture and Implementation" *IEEE Trans. on Knowledge and Data Engineering*, 2(1), 1990.

[WN94] J. Wiener, J. Naughton, "Bulk Loading into an OODB: A Performance Study", *Proc. of the 20th Intl. Conf. on VLDB*, 1994.

[ZM91] S. Zdonik, G. Mitchell, "ENCORE: An object oriented approach to database modeling and querying", *Data Engineering*, 14(2), 1991.

[Zwil+94] M. Zwilling et al., "On Shoring Up Persistent Applications", *Proc. of ACM SIGMOD*, 1994.

Appendix A

Grammar of query language

```
command_line        :    stmts

stmts               :
                    |    stmts select_stmt

select_stmt         :    select_clause
                         SEMI_COLON
                    |    SEMI_COLON
                    |    quit SEMI_COLON
                    |    clear SEMI_COLON

select_clause       :    select
                         identifier_list
                         from design_name
                         opt_where_clause

design_name         :    identifier

identifier_list     :    identifier
```

```
                        |       identifier_list COMMA identifier

opt_where_clause        :
                        |       where
                                OPEN_PAREN
                                attributes
                                select_operator
                                attributes opt_clause
                                CLOSE_PAREN

attributes              :       attribute_value
                        |       attribute_name
                        |       OPEN_PAREN
                                select_clause
                                CLOSE_PAREN

opt_clause              :
                        |       set_operator
                                attributes
                                select_operator
                                attributes
                                opt_clause

select_operator         :       set_operator
                        |       relop


set_operator            :       and
                        |       or
```

```
attribute_name          :       identifier
                        ;


relop                   :       NE
                        |       LT
                        |       LE
                        |       GT
                        |       GE
                        |       EQ

attribute_value         :       double_or_integer
                        |       quoted_string

double_or_integer       :       double
                        |       integer
```

Appendix B

DIAS Object Model