Parallel Processing For Schema Evolution in Database Systems

Labib, Gamal Ali Mohamed

Queen Mary and Westfield College
University of London
Department of Computer Science

Parallel Processing For Schema Evolution In Database Systems

Gamal Ali Mohamed Labib

A thesis submitted to the University of London in partial fulfilment of the requirements of the degree of Doctor of Philosophy (Ph.D.).

October 1995

Abstract

Schema evolution in object-oriented database management systems (OODBMS) has received the attention of researchers for many years. This resulted in the introduction of different concepts, implementations, and even standards outlining the ways in which schema can evolve and how they are being supported. However, the approaches currently taken in those systems impose either limitations on the supported schema changes, restrictions on the configuration of schema and class versions, or limitation on compatibility between those versions. On the other hand, studying the impact of supporting schema evolution on system performance received little attention as a research direction for OODBMS. This thesis improves the functionality of OODBMS concerning schema evolution, and investigates its related performance aspects.

I propose a class versioning approach that extends the supported schema changes with modification to attribute semantics, supports a hierarchy of class versions, and provides forward and backward compatibility of class versions. The approach ensures full accessibility to objects using any version of their class and supports migration of objects between class versions without compromising the persistence of their data. I specify process types and configurations that may take part in accessing and manipulating persistent objects of class versions. The performance of those configurations in parallel processing environments is analysed. For this purpose, simulation is used and a prototype is built. I demonstrate the effectiveness of combining multitasking, process replication and associative techniques with pipeline and dataflow processing in improving OODBMS performance and prove the applicability and practicality of adopting the class versioning approach in OODBMS.

*To My Family
and
My Country*

Acknowledgements

I would like to thank my supervisor Dr. Dave Saunders for his invaluable support, guidance, and advice throughout the course of this research. I would also like to express my gratitude to him and to Prof. Dr. Heather Liddell for recommending me to the Overseas Research Support body (ORS). I would like to express my deep appreciation to both the Egyptian Government who provided a grant of full support for the duration of my research, and to the ORS body for offering me a two-year grant.

I would like to thank my Ph.D. committee members: Prof. Dr. Yakup Paker and Dr. Silvano Barros for their advice and continual review of my work as it progressed.

I appreciate the help and support of both Prof. Dr. Hossain Abdel Aziz and Prof. Dr. Samir El Sayaad–the Egyptian Education Councillors, and the Egyptian Education Bureau in London, UK.

I cannot forget the technical support of my colleagues in the ACE lab. specially Eng. Derek Coppin for his technical advice regarding the parallel processing aspects of my research. I must also mention the effort of Dr. Ali Sahanier and Dr. Tim. Kindberg in setting up the Equus system for me to produce my prototype, and in resolving the technical obstacles I faced.

I give my special thanks to my colleague Dr. Soheir Ghallab for her advice, support, and encouragement at all stages of my study in the UK.

I also give my thanks to Dr. Simon Monk for answering questions about the CLOSQL system and for his remarks regarding my work.

Table of Contents

PART I : GENERAL

Chapter 1 : Introduction

Chapter 2 : Background and Related Work

Summary of PART I

PART II : THE CLASS VERSIONING APPROACH

Chapter 3 : Concepts and Features

PART III : PARALLEL PROCESSING ASPECTS

Chapter 9 : The SIMD Query Engine

Summary of PART III

PART IV : Conclusions, Appendices and Bibliography

Appendix A : Query-Engine Design and Adapted Algorithms

List of Figures

List of Tables

List of Acronyms

a_i	attribute i
AM	Associative Memory
ASC	Attribute's Semantic Change
CAD	Computer-Aided Design
CAM	Computer-Aided Manufacturing
CASE	Computer-Aided System Engineering
CCU	Central Control Unit
CL	Version-Collection
CLSR	Class Server Process
CLTR	Version-Collection Transformer process
CPU	Central Processing Unit
CV	Class-Version
CVM	Class-Version Mapper process
CVQR	Class-Version Query Resolver process
DAG	Direct Acyclic Graph
D_{ai}	Domain of attribute a_i
DBA	Database Administrator
DBMS	Database Management System
DBS	Database System
DMA	Direct Access Memory
EOO	End Of Object associative word
GUI	Graphic User Interface
I/O	Input/Output
ICV	Intermediate Class-Version
JN-RS-DT	Join-Then-Restrict, Deferred-Transformation process configuration
JN-RS-ET	Join-Then-Restrict, Early-Transformation process configuration
LNF	Link-Next-Field command
LNW	Link-Next-Word command
LPF	Link-Previous-Field command
LPW	Link-Previous-Word command
LUW	Logical Unit of Work
MIMD	Multiple-Instruction Multiple-Data architecture
OAM	Object-Associative Memory
ODMG	The Object Data Management Group

OID	Object Identifier (also `ObjectID`)
OM	Object Merger Process
OODBMS	Object-Oriented Database Management System
PSM	Physical Storage Model
QE	Query Engine
RAM	Random Access Memory
RDBMS	Relational Database Management System
RS-JN-DT	Restrict-Then-Join, Deferred-Transformation process configuration
RS-JN-ET	Restrict-Then-Join, Early-Transformation process configuration
SOO	Start Of Object associative word
SIMD	Single-Instruction Multiple-Data architecture
SISD	Single-Instruction Single-Data architecture
SQL	Structured-Query-Language
SV	Schema-Version
UQS	Universal Query Server process

Part I
General

This is a two-chapter introductory part. The first chapter discusses the motivations, objectives and achievement of our research which are centred around functionality and performance aspects of OODBMS related to supporting schema evolution. The chapter concludes with an outline of the thesis. The second chapter presents a background and work related to the thesis. We end this part with a summary of its main topics.

This page is intentionally left blank

Chapter 1
Introduction

1.1. Thesis Motivation

Application environments that are expected to use object-oriented database management systems (OODBMS) require mechanisms to support schema changes. These applications are very often revolutionary and characterized by the fact that schema modifications are a rule, rather than an exception. For example, it is common that during the design of an application, the ways of classifying objects and their interrelationships evolve [Bertino 92]. Such evolution is reported by Sjøberg [Sjøberg 92] for an in-use health management system during its development and early stages of use. The number of relations modelled by the system increased from 23 to 55 while the number of attributes increased from 178 to 666.

In order to account for additional or modified database schema specification imposed by an evolving existing application or by a new application, the user is faced with two alternatives: either to update his current programs and migrate the existing data to match the new schema, or to adopt a conversion mechanism that achieves the compatibility of data between different versions of the schema without changing existing programs. If schema evolution is frequent, the first alternative will be impractical and expensive, thus making the second preferable. Database views, object versioning, modifying and versioning of classes and schemas are approaches considered to support schema evolution in some of the currently available database systems. However, the proposals for implementing those approaches suffer from drawbacks such as imposing a limitation on the supported schema changes (e.g., the lack of support for changing attribute semantics), restricting schema and class versioning to a specific configuration (that is linear versioning), providing only partial compatibility of data between class or schema versions (e.g., being restricted to forward compatibility), or compromising the persistence of object data while transforming objects between class versions. Yet, none of the existing systems attempts to deal with all those aspects at the same time, the thing which we perceive being a limitation of OODBMS functionality and applicability. We envisage that devising a new approach for schema evolution that handles the aforementioned drawbacks would be beneficial (if not crucial) in extending the applicability and operability of OODBMS.

Another related aspect to schema evolution is its impact on system performance which received little attention in existing systems and research efforts. We perceive that supporting forward and backward compatibility of class or schema versions, which is a challenging goal, through dynamic transformation of objects would impose additional overhead on system resources. If such transformation is to be carried out during run-time, then we should expect a degradation of system performance of varying extent. Very few systems that support schema evolution consider the consequences of such added OODBMS functionality. At the same time, none of those systems has quantitative performance analysis conducted for its support of schema evolution. Taking into consideration that OODBMSs are required to perform navigational and set operations as opposed to relational systems which mainly perform set operations, we perceive that investigating performance aspects related to the added support of schema evolution in OODBMSs deserves to be a research direction to achieve acceptable performance of OODBMSs.

1.2. Thesis Objectives

The research objectives are centred around two main goals: improving the functionality and performance of OODBMS with respect to schema evolution. For the first goal, we seek a class and schema versioning approach that would support the following features:

- hierarchical configuration of class-versions, that is: modifying a class definition is not restricted to the most recently-created version of the class but can be applied to any of its versions.
- changing attribute semantics, including attribute name; domain range; units of measurements; default value.
- modifying the inheritance hierarchy by adding or removing classes to (from) its structure or by adding or removing attributes and methods to (from) its classes.
- accessibility of class instances using any version of the class, that is: forward and backward compatibility of class versions.
- migration of objects between versions of their class to achieve object locality in a favoured class version, to provide extra storage of attribute values, or to permit changing attribute semantics of objects.
- preserving object data if it migrates to different class version. This feature accounts for acquired object data for existing and new attributes as the object bounces between class versions.

Proposing a class and schema versioning approach requires considering a set of related issues: how the versions hierarchy is managed; how class-version queries are generated and resolved; how qualifying objects are retrieved and modified; the physical representation of objects; programming language support to access instances of class versions; the applicability of the approach to extended object-oriented and other data models.

Since the targeted schema evolution approach would provide broader functionality compared to existing approaches; investigating its performance can produce conclusions valuable to existing and future approaches. This led to the second goal for which we pursued–the adoption of parallel processing to improve the performance of OODBMS supporting our approach. In doing so, we highlight the parallelism inherent in our approach, enumerate the different type of processes involved, and study possible process configurations that exploit parallel systems capabilities. We used tools inasmuch as resources and time permitted to achieve this target: simulation, available parallel systems, custom-designed hardware modules.

1.3. Thesis Achievements

The thesis presents a class versioning approach that extends the range of supported schema changes, supports a hierarchy of class versions, and provides forward and backward compatibility of class versions. The approach ensures the accessibility to objects using any version of their class and supports migration of objects between class versions without compromising the persistence of their data. Those features are realized by grouping the versions of a class into different version collections, for the same class, depending on the nature of the schema changes they introduce and not on the order of their creation. Objects can be mapped from any class version to another across the version collections using transformation functions attached to each class version and version collection.

The thesis discusses the concepts and features of the proposed approach with respect to the basic object-oriented data model, and studies the adoption of the approach to semantic as well as extended object-oriented data models, through case studies. The thesis also describes the implementation of the approach in a C++ environment and the extensions to the programming language to support the manipulation of objects using different class-versions. Such extensions provide seamless access to temporary (transient) and persistent objects, and eliminate the impedance mismatch between the query and the programming language.

The thesis investigates different performance issues related to the class-versioning approach. It demonstrates how to exploit the parallelism inherent in the approach and discusses the potential of MIMD (Multiple-Instruction-Multiple-Data) and SIMD (Single-Instruction-Multiple-Data) architectures in querying and manipulating objects. We implemented a prototype for querying and manipulating simple objects, within the framework of the class-versioning approach, using a backend MIMD machine running a distributed operating system. We used the prototype to demonstrate the benefits of process replication in conjunction with multitasking in achieving better query service time, higher process utilization, and load balancing among processing nodes of the parallel machine. Based on the outcome, we built simulation models for the multiprocessor environment by which we were able to study different schemes of querying complex and inter-related objects. We used simulation to investigate the performance gains of combining associative techniques with multitasking, process replication, pipeline and dataflow processing. This investigation demonstrated the feasibility of performing object transformation dynamically during run-time while still achieving acceptable system performance.

When studying the adoption of associative techniques, we made assumptions of speed-up to be achieved for objects selection and join. To validate those assumptions, we took our effort one step further and presented a design of an object-oriented query engine based on SIMD architecture that would deliver such speed-up when integrated in SISD (Single-Instruction-Single-Data) or in the nodes of MIMD machines. We conducted quantitative analysis for the engine's performance to facilitate estimating query cost.

1.4. Thesis Outline

The thesis comprises nine chapters, grouped into four parts, following the title page, the acknowledgement, table of contents, list of figures, and list of tables.

Part I is an introduction of two chapters discussing the research motivation, objectives, and background. Chapter 1 discusses the motivation for choosing object-oriented database as the main theme of conducting this research, and main problem areas that lay there. It also outlines what we attempt to accomplish in this work. Chapter 2 presents a background of the main functional and operational issues related to object-oriented database systems, and surveys the research efforts done so far in those respects.

Part II is dedicated to the proposed class-versioning approach and comprises three chapters. Chapter 3 is the backbone of this part that discusses the concepts and main features of the approach. Chapter 4 discusses the implementation aspects of the approach:

the object physical storage models; object manipulation; querying class-versions and query optimization; aspects of programming languages. Chapter 5 examines the application of our approach to different data models through case studies.

Part III deals with the parallel processing issues related to the class-versioning approach and comprises four chapters. Chapter 6 investigates the parallelism inherent in the approach, and outlines the potential of parallel processing in improving the performance of our approach. Chapter 7 describes a prototype implementation for object manipulation in a parallel processing environment, taking into consideration the conclusions reached at that point. Chapter 8 presents our experimentation using simulation to demonstrate the benefits of processor multitasking and associative techniques in improving system performance. Chapter 9 is dedicated to a proposed associative query engine and describes its design, operation and performance.

Part IV presents research conclusions, directions for future work, an appendix for further details about the query engine, and the bibliography.

This page is intentionally left blank

Chapter 2
Background and Related Work

This thesis relates to two directions of research: schema evolution in object-oriented databases and parallel processing for supporting that functionality. Accordingly, we divide our review into two main themes, each of which is dedicated to one of those research directions. In sections one and two, we present the concepts of object-oriented paradigm and the main features of OODBMS. In section three, we review the physical storage models of objects. In section four, we elaborate on schema evolution aspects in OODBMS and review its existing realization approaches. In section five, we review the parallel processing prospects in OODBMS, then in section six we review existing methods for manipulating objects in both uniprocessor and multiprocessor environments. In section seven, we review the role of computer architecture in supporting object-oriented data models. We end this chapter with a summary of its main topics.

2.1. Overview of Object-Oriented Concepts

Different perceptions of object-orientation concepts have been suggested in the literature. Normally, the presented concepts are agreed upon by the majority, and in some cases additional features are added. In this section, we attempt to gather some of the prominent basis of object-orientation.

Khoshafian [Khoshafian 93] emphasizes three most fundamental aspects of the object-oriented paradigm. Those are ***abstract data types*** , ***inheritance*** , and ***object identity*** :

- Data type is used to describe a set of objects with the same representation. Several operations are associated with each data type. Abstract data types extend the notion of a data type by "hiding" the implementation of the user-defined operations ("methods") associated with the data type. Therefore, abstract data types specify both an object's structure (appearance) and behaviour (which messages are applicable to the object). In object-oriented paradigm, as in some conventional languages, operations may have the same name but different semantics and implementations. This feature is called ***overloading*** which allows such operations to be invoked for objects of different types (or classes). A concept closely associated with overloading is ***dynamic binding*** (also known as "late binding") which allows the system to bind message selectors to the methods that implement them at run time, instead of at compile time. A ***class*** is the language construct most commonly used to define abstract data types in object-oriented programming languages. A class incorporates the definition of the structure as well as the operations of the abstract data type. A class definition (minimally) includes the following: (1) the name of the class, (2) the representation of class instances (attributes), (3) the external operations for manipulating instances of the class and their attributes. A class represents the set of all possible objects with the prescribed structure and behaviour. By contrast, the ***extension*** or extent of a class corresponds to the actual instances of the class that have been created but not destroyed.

- Inheritance, on the other hand, enables new software modules (e.g., classes) to be built on top of an existing hierarchy of modules. Class inheritance has two main aspects:

(1) ***Structural.*** Instances of a class would have values for instance variables inherited from its superclasses. Subclasses may *redefine* (or override) the type declaration of inherited instance variables. This overriding may be arbitrary or constrained.

(2) ***Behavioural.*** Inheriting behaviour enables *code sharing* (and hence reusability) among data objects. Instances of a class can be operated upon using a method defined in one of their superclasses by sending those objects the corresponding method (or message) selector. An inherited class method can be overridden in a subclass by redefinition while keeping its original name.

The combination of both abstract types of inheritance provides a powerful modelling and software development strategy since it allows *new* classes to be defined on top of existing hierarchies rather than from scratch and offers a natural model of organizing information. There are also two mechanisms of inheritance:

(1) ***Single Inheritance.*** A class can have only one immediate superclass where the database inheritance hierarchy forms a tree with the most general class at the root of the tree.

(2) ***Multiple Inheritance.*** A class can have more than one immediate superclass. In this case, the inheritance hierarchy forms a directed acyclic graph.

- Finally, object identity, which is the property that distinguishes each object from all the others. In OODBMS, each object is assigned a unique identifier which may be system generated. The most common type of object identity in programming languages, databases and operating systems is *user-defined names* of objects. With object identity, objects can contain or refer to other objects.

Bertino and Martino [Bertino 91, 93a] state that classes in the object-oriented data model are organized in an *inheritance hierarchy* , orthogonal to the aggregation hierarchy, and that a class has two notions:

- intensional; where the class forms the domain of a property (i.e., attribute) of another class, the thing which establishes a relationship called *aggregation relationship* , between the two classes. Such relationship organizes classes in an *aggregation graph* .
- extensional; where the class groups the set of objects sharing a common definition.

The Object Data Management Group (ODMG) [ODMD-93] presents in its proposed standard additional features to the object-oriented data model. It distinguishes between two kinds of entities: objects which are identified by unique OID and are mutable (i.e., retain their OIDs even if their properties change), and literals which are immutable and use their bit pattern for identity. Both entities may be atomic or structured. An object may have a single object identifier, but it may have more than one name each of which must refer uniquely to the object. The model includes a built-in set of collection types–sets, bags, lists and arrays. Those types group objects (or references to objects) and are distinguished from each other by the way they order objects and whether they allow duplicate objects in the same collection.

2.2. Why Object-Oriented Databases ?

Schema evolution, which is the main theme of this research, is intimately related to OODBMS. Manipulating and preserving persistent objects, in particular, are two of the main concerns that arise when the schema evolves. On the other hand, managing those objects lies in the domain of activities of DBMS. So, it is necessary at this point to review some of

OODBMS aspects, namely: its classification, in what way it differs from existing database systems (DBS), the functionality it offers. In what follows, we quote some of the literature points of view regarding those aspects. Similar viewpoints to what reviewed here were also expressed by Atkinson *et al.* in their manifesto [Atkinson 89], by Butterworth [Butterworth 91a, 91b], and by Budd [Budd 91].

Brown [Brown 91] regards object-oriented databases as a merging of conventional database technology and the object model. The database technology, on one hand, furnishes a generic solution to the problems of concurrent database access, data integrity and independence, security, and backups. The data model, on the other hand, aims at producing a representation whose state at any instant in time closely resembles the state of the real world.

Booch [Booch 91] assumes the existence of a programmatic interface between the object-oriented programming language and a relational database system, and that functionally the latter can be made available to the former. The conceptual distance between this programmatic interface and the high-level classes could represent a large ***semantic gap*** . This becomes clear if the class represents, from its inside view, a flattening of data from several tables in the database. Updating an object requires updating several tables, since the RDBMS disallows performing an update upon a join. Booch suggests that his semantic gap is one of the primary factors that has motivated the OODBMS. He also perceives that uniting these two levels of abstraction is undeniably a hard job.

Graefe *et al.* [Graefe 88] perceive object-oriented database systems as being new and conceptually powerful alternative to existing systems . They support structures and features that are not provided in conventional data models. Among these features are complex objects, object identity, encapsulation of methods and behaviour, hierarchies and inheritance of data types, versions of objects, and very large objects. These concepts make object-oriented programming and database systems well suited for engineering applications, including CAD, CAM, and CASE.

Bancilhon [Bancilhon 92] classifies OODBMS, based on the user's view of the system and not on the implementation or architecture basis, as four types: language oriented DBS, persistent programming languages, engineering DBS, and full OODBMS. He identifies the types of users to OODBMS, which are not necessarily mutually exclusive, and classifies them by their motivation :

- C++ users (dominating the market)- who want an OODBMS because they believe in object-oriented technology, and want a solution to impedance mismatch between C++ and a traditional DBS.
- Engineering application developers - who want a DBS fulfilling modelling and specific functionality requirements.
- New application developers - who want support for non-standard data types and extendibility.
- Database users seeking better database technology - who believe object orientation will improve the development process and the resulting applications.

The Object Database Standard [ODMD-93] defines an OODBMS to be a DBMS that integrates database capabilities with object-oriented programming language capabilities. An OODBMS makes database objects appear as programming language objects, in one or more existing programming languages. The OODBMS extends the language with transparently persistent data, concurrency control, data recovery, associative queries, and other database capabilities.

2.3. Objects Physical Storage Models

There are many ways that an object-oriented database may physically store its instances. On the logical level, an Object-oriented database is arranged as a *directed acyclic graph* (DAG). On the physical level, however, there are different alternatives to the way class instances are stored. Alternate strategies were proposed for simple and complex object storage; and in order to have a global view of the object-oriented paradigm, we review those strategies in the following subsections. In chapter 4, we will revisit those strategies and discuss how they apply to the class-versioning approach.

2.3.1. Simple Objects

Willshire and Kim [Willshire 90] identify six physical storage models, for the IS-A hierarchy, that could be used for the location of the values associated with the instance variables of a given object. Those models are:

- **Physical Storage Model 1** (PSM-1): Each instance occurs exactly once in the database. An instance is pushed to the lowest level possible in the class hierarchy. No data is duplicated, and the entire instance is contained in exactly one class.

- **Physical Storage Model 2** (PSM-2): If an instance is a member of more than one class, then the data for the instance is stored in each of those classes. The model directly implements the logical view of the database.
- **Physical Storage Model 3** (PSM-3): In each applicable class, the unique `ObjectID` is stored along with the values of the attributes particular to that class for the instance. Only the *id* field is duplicated down the IS-A hierarchy.
- **Physical Storage Model 4** (PSM-4): It is the object-oriented version of the universal relation assumption. If no two classes have exactly the same set of instance variables, class membership can be deduced by examining the pattern of nulls, but not otherwise, so the notion of a class is lost in this model. Here, all data is available in a central location, but null values waste space and complicates query processing. Data manipulation is the same as in relational databases.
- **Physical Storage Model 5** (PSM-5): Instances can belong to several classes that are not directly related by the IS-A hierarchy, that is, no class is in the set of superclasses of the other classes.
- **Physical Storage Model 6** (PSM-6): It stores all data in one table but avoids wasting space due to nulls and recaptures the class membership information. The bulk of data is stored in triples having the form: `(ObjectID, InstanceVariableID, Value)` where the `Value` will be the `ObjectID` of another object if it is not an integer or real value (for example, a String). Strings are stored as: `(StringID, StringValue)`. The model employs a binary relation that maps an object to its most specialized class: `MinClass(ObjectID, ClassID)`.

The ORION DBMS [Banerjee 87, Kim 90b, Kim 90c] is one example of a system adopting the PSM-1 model. Thakore and Su [Thakore 94] consider a combination of PSM-3 and PSM-6 models in their analysis of parallel object-oriented query processing algorithms for the OSAM* data model [Su 89, Lam 89]. The PSM-6 model is supported by the architectural design of the IFS machine [Lavington 87, 88, 91, 92].

Some of the referenced systems will be discussed later on in this chapter, while the others will be reviewed in subsequent parts of the thesis.

2.3.2. Complex Objects

The previous subsection presented models for object storage that dealt with simple objects and their representation in classes of the inheritance hierarchy. Here, we focus on other models presented by Valduriez *et al.* [Valduriez 86] for complex object storage. Each model was evaluated based on its retrieval and update performance where I/O overhead was the main criteria.

- **Direct Storage Model,** clusters the parent object with its *subobjects* (or constituent objects) where alternative strategies may be used (such as depth-first or breadth-first). This model avoids decomposing complex objects and consequent re-joins for retrieval. So, it is efficient in the retrieval of entire complex objects. However, if subobjects are to be referenced by objects other than those physically containing them, then joins would be required.

- **Normalized Storage Model,** is similar to mapping complex objects to a third normal form in a relational database. To capture the one-to-one, one-to-many, and many-to-many relationships, foreign keys are introduced in cross-referencing tables. The model provides better performance in accessing subobjects contained in other parent objects whose access becomes unnecessary to prove their relationship with the subobjects. It also provides better support for many-to-many relationships since objects are neither replicated nor stored with only one parent while being referenced by others. The model has three variants: *binary (Decomposed) model,* which is similar to the PSM-6 model discussed in pervious subsection and is based on pairing each attribute value of an object with the `ObjectID` and supports cross-references of related objects; *N-ary model*, in which each type of objects is stored in a single file; *partial decomposed model*, which is a hybrid between the previous two variations where vertical object partitioning is based on attribute affinities.

  The model is efficient in supporting arbitrary access patterns, but still incurs joins for reconstructing complex objects upon retrieval.

Valduriez *et al.* consider two types of indices to improve the retrieval of complex objects and their constituent objects: *binary join indices*, which relate objects of two types; *hierarchical join indices*, which have a more general structure and capture the structure of a complex object.

2.4. Schema Evolution in Object-Oriented Databases

Different approaches for supporting schema evolution exist with varying motivations and capabilities. It has been recognized that dynamically modifying a database schema but yet retaining its old versions as being very important for advanced applications like CAD/CAM, multimedia documents, software engineering, where defining the structure and the behaviour of objects is a crucial activity. Motivated by such application characteristics, object, class and schema versioning and database views approaches were introduced where existing objects, or definitions of classes and schemas are maintained while versions of those entities are created to reflect the required modifications. Systems supporting any of those approaches normally provide means of compatibility between versions and eliminate the need to modify programs to cope with either version. Class and schema modification, on the other hand, are approaches that do not support such compatibility and require existing objects to be transformed and programs to be modified to be in concert with the evolving schema. Object transformation can be either immediate following schema evolution, or lazy meaning that objects are updated to the new schema only whenever they are accessed by the user. Guidelines by which schema may evolve have been identified and adopted by some of the existing object-oriented database systems. In the following subsections, we shall review those guidelines. Then, we shall discuss each of the approaches mentioned earlier, and review the main features of some of their notable representatives.

2.4.1. Class Invariants

Every OODBMS contains a number of integrity constraints that must be maintained across schema changes. These constraints, generally called class invariants in the literature, impose a certain structure on class definitions and on the inheritance graph. Depending on the object model, those constraints may allow more or less restricted types of class hierarchies. Casais identifies class invariants [Casais 91] as:

- *Representation invariant*, states that the properties of an object (attributes, storage format, etc.) must reflect those defined by its class.
- *Inheritance graph invariant*. The structure deriving from inheritance dependencies is restricted to form a connected, directed graph without cycles (so that classes may not recursively inherit from themselves), and rooted at a special predefined class called OBJECT.

- *Distinct name invariant.* All classes, methods and variables must be distinguished by a unique name.
- *Full inheritance invariant.* A class inherits all attributes from its ancestor, except those that it explicitly redefines. Naming conflicts occurring because of multiple inheritance are resolved by applying some precedence scheme and selecting one attribute as the one being inherited.
- *Distinct origin invariant.* No repeated inheritance is admissible: an attribute inherited several times via different paths appears only once in a class representation.
- *Type compatibility invariant.* The type of a variable redefined in a subclass must be consistent with its domain as specified in the superclass. This constraint means that the new type must be a subclass of the original one. Some systems require identical method signatures of a class and its descendants.
- *Type variable invariant.* The type of each instance variable must correspond to a class in the hierarchy.
- *Reference consistency invariant*, guarantees that there are no dangling references to objects in the database. When a user holds a reference to an object, then both the object and the reference to it are retained. Instances can only be deleted when they are no longer referred to. Some systems require that two references to the same object before modification also point to the same entity after modification.

2.4.2. Primitives for Schema Evolution

From the reviews conducted by Nguyen and Rieu [Nguyen 89] and Casais [Casais 91], we may classify schema changes into the following categories:

- *Changes to class definitions*, includes adding or deleting new instance variables and methods; modifying existing instance variables and methods (e.g., changing their name, domain, constraints, default values).

- *Adding and deleting classes*, is a common feature affecting class lattice in object-oriented data models. This includes creating new specialized classes from existing classes, or removing existing classes for generalization.

- *Changing class relationships*, which allows incremental definition of objects or modelling new semantics. One way to achieve this is by adding a specialization relationship between relevant classes in the lattice and in which case would require the support of multiple inheritance.

- *Reordering inheritance dependencies*, by relocating classes in the lattice.

Not all the systems reviewed here, however, support all categories of schema changes. On the other hand, supporting such changes to a database schema may require the propagation of changes on class definitions throughout the inheritance hierarchy, applying changes to simple-class instances, and applying changes to composite objects and their component objects (which involves the way component objects are created and shared). Two issues arise when considering schema evolution. The first is its completeness in an OODBMS and is concerned with the provision of operations that would support the aforementioned changes. The second is its correctness and is concerned with the generation of class structures that satisfy all integrity constraints. If the invariant properties of the inheritance hierarchy cannot be preserved, then changes to class structure should be rejected.

2.4.3. Schema and Class Modification

The purpose of this approach is to bring existing objects in line with a modified class. Screening is one method that defers modifying the persistent store; object data is either filtered or corrected before it is used. This method may compromise execution speed by screening. Another method, conversion, changes all instances of the class to the new class definition, ensuring that auxiliary definitions such as class methods agree with the new definition. In this method, much time may be spent to reflect class modification on existing database objects. In what follows, we shall review some systems adopting those methods.

GemStone [Penney 87, Bretl 89] is an OODBMS employing conversion method. GemStone is composed of: *Stone* providing persistent object management, concurrency control, authorization, transaction, and recovery services; *Gem* augmenting the Stone services by providing compilation mechanisms for OPAL (based on Smalltalk) programs, authorization control, monitoring of user sessions, predefined set of OPAL classes and methods. GemStone automatically propagates class modifications to subclasses if such changes do not affect any of predefined schema invariants. It applies additional invariant for an evolving schema, called *preserving information invariant* which insures that there is no loss of information. This emerges from the fact that in GemStone there is no explicit delete operation.

ORION [Kim 88a, Kim 90b, Kim 90c] is a multitasking database system intended for applications in AI, CAD. It has single-user and multi-user versions and runs in both workstation and distributed environments. Banerjee et al. [Banerjee 87] present a mechanism for screening in ORION. If the modification to class definition is addition of a new attribute, then for each object read request a default value or a 'nil' is returned for the attribute till an application assigns a value for it. Deletion of an attribute, on the other hand, is dealt with by screening it when objects are accessed. Supporting object and schema versioning was also proposed and implemented in ORION as we shall see in Subsections 2.4.5, 2.4.7.

Lerner and Habermann [Lerner 90] propose automatic updating of database objects to the latest version using a system called OTGen (Object Transformer Generator). OTGen is a program generator that is applied to the delta of the data definitions and lets it produce the necessary programs and tables that can transform the existing data into the new format. OTGen shares with ORION and GemStone the kind of data transformations that can be automated. The OTGen allows the database administrator to reorganize the data by editing the transformations tables with the assistance of interactive OTGen environment. This requires that the application software be modified to deal with the latest versions, and objects to be updated to the latest version. OTGen supports screening and forward versioning of objects only. If the objects are more than one version out-of-date, a series of transformations will be called, each updating the objects one version.

Nguyen [Nguyen 89] proposes a scheme that enables a prototype object-oriented knowledge base system called *Sherpa* to fully support the propagation of changes and the dynamic classification of the instances whose class definitions are modified. The scheme provides for both immediate and deferred updates.

2.4.4. Database Views

In relational database systems, a view is introduced as a virtual relation derived by a query on one or more existing relations with authorization augmented to control access to data. This provides support of external schemas at the convenience of the user alongside data protection. Views are useful in allowing different interpretations of the same class, without altering the inheritance hierarchy. They also allow partitioning class objects into subsets (or subclasses) based on constraints over those objects. A view can also enforce access control to instances of a class, for example, by defining predicates on the values of specific attributes of the class. In object-oriented databases, proposals for view mechanisms have been made to support schema evolution at two levels of granularity: class views [Scholl 90, Bertino 92], in which a view is a single class; schema view [Kim 89], in which a view is a schema of multiple classes interrelated through attributes, methods, and IS-A relationship.

Scholl and Laasch [Scholl 90] describe an approach in which a view is inserted in the inheritance hierarchy and is handled as a subclass or a superclass of some existing class(es). This approach has the drawbacks of complicating the inheritance hierarchy, and the possible inclusion of classes that are not semantically meaningful to the user.

Bertino [Bertino 92] proposes a view mechanism that adds to the schema a new dimension called *view derivation* . A view derived from a class may have fewer (or more) properties and methods than the class. This allows the view to augment class definition and support schema changes. A view, like a class, can be defined as subview of other views. The view mechanism suggests that some of the database objects are to be manipulated exclusively by the views that created them, and not by the classes in the inheritance hierarchy. This is due to the possible mismatch of properties and/or methods of a class and its related views. Views normally do not support updating data entities, whether those entities are tuples or objects. It is unclear how the view mechanism would handle cases where an object is required to have values for a property not in its class definition, or beyond the property domain. The mechanism also implies the violation of object encapsulation by introducing the view as an external abstract data type, that is capable of manipulating objects of more than one class, in different inheritance paths of the class hierarchy.

2.4.5. Object Versioning

A version of an object can be thought of as a semantically significant snapshot of the object, taken at a given point in time [Bertino 93]. Normally, a new version is created to depict different implementations of the same object and revisions on the object. In other words, the generation of new versions of an object could be associated with any of a number of different semantics, the simplest of which is perhaps any update to the object. Derivation and history are basic conceptual mechanisms of version management. Katz [Katz 90] describes the general requirements for version management systems and reviews version management mechanisms. In what follows, we review some systems that adopt this approach in support of schema evolution.

Chou and Kim [Chou 88] present the version model developed for ORION. It supports three types of versions: *transient version*, can be updated, deleted, or versioned with a new transient version; *working version*, stable and cannot be updated, may be deleted, may be versioned with a new transient version; *released version*, cannot be updated or deleted, may be versioned with a new transient version, can be derived from a working version but, in contrast to the other two types, resides in a public database rather than private workspace. A versionable class would have a version-derivation hierarchy grouping all versions of an instance of the class. A newly created version is *transient* while its parent version becomes a *working* type. The version model employs a change notification method whereby an updated object has two distinct timestamps: *change-notification timestamp*, indicates the time the object was created or the last time it was updated; *change-approval timestamp*, indicates the last time at which the owner of the object approved all changes to the objects it references. The user can specify object attributes to be *notification-sensitive* so that the instance is notified of any changes to those attributes.

AVANCE [Bjornerstedt 88, 89] is a research prototype that supports sharing of persistent objects, nested transactions, decentralization of both data and control, and a strongly typed programming language. AVANCE is designed to provide three levels of abstraction to implementations: the object manager, the AVANCE virtual machine, and the high-level language PAL. In AVANCE, an object is represented by a *Packet* with an identifier used for referencing object's components (called *Packet Slices*). A *Packet Slice* consists of a set of persistent instance variables (i.e., attributes) belonging to one specific packet type that may be inherited by the packet. AVANCE supports object versioning using three mechanisms:

- *Packet State History*, in which a new version is created with only differences (deltas) stored for modified packet slices while old versions may be retained to reflect the packet state over a continuous time.
- *Simple Packet Versions*, is based on the previous mechanism and enables the creation of identifiable packet versions which may be referenced by an integer value.
- *Complex Packet Version*, enables the creation and handling of a network of packet versions rather than being restricted to linear versioning.

This object versioning is extended to the versioning of class definitions. AVANCE supports the declaration of exception handlers that may convert an object between its different representations and its version expected by the query. In the absence of proper exception handlers, access to objects would be aborted.

The International Standard ISO/IEC [ISO 93] groups versions of objects in *Working Sets*. New working sets can be created from existing sets (by modifying any of its object versions), forming a hierarchy of working sets. Within the hierarchy, a path connects the new set to the existing set on which it is based. Object versions in one working set may reference objects of other classes, located in different working set path by establishing a directional *reference path* between the two sets.

El-Sharkawi *et al.* [El-Sharkawi 90] consider versioning caused by object migration to different classes, in temporal OODB, as a result of adding, dropping or modifying object attributes. They present an object migration mechanism to support such updates and categorizes instance variables, classes, and schemas according to object migration and actions required from the database management system (DBMS) when an object update occurs.

EXODUS project [Carey 86] is a "database generator" rather than a single DBMS for use by all applications. EXODUS provides powerful fixed components in addition to a collection of tools for the database implementor to use in building the desired system based around these components. EXODUS provides primitive level of version support where updating an object leads to the creation of a new version with a unique `ObjectID`. EXODUS allows versioning large objects and supports sharing unchanged components of an object between its versions.

IRIS [Lyngbaek 86, Fishman 89] consists of an object manager which is built on top of a relational database system (Allbase) acting as the storage manager for the system. IRIS provides simple version control facilities. A user-defined object can be made versionable and can be accessed through checkout and checkin commands which create the next version of the object. Locking is used to control sharing of versions.

2.4.6. Class Versioning

Supporting schema evolution with this approach is achieved by the creation of a new version of the class definition being modified. Unlike class modification approach, existing class definition is preserved while the new version of the class presents the modified class definition. This allows the coexistence of multiple versions of the same class in the database. Programs written to access a particular version of a class need no modification to access instances of that version, as well as instances of new or old class versions. This achieves forward and backward compatibility between class versions. Our proposed schema evolution approach falls under this category of mechanisms. We dedicate part II of the thesis to the presentation of our approach. In this subsection, we review in some detail a group of prominent mechanisms that support class versioning.

The CLOSQL system [Monk 92, 93] defines update and backdate functions on attributes of the old and new versions of a class definition, instances of any version of the class can be converted to instances of any other version. CLOSQL supports only linear versioning by which a new version of a class can only be generated from the latest version. This may not suit CAD/CAM developers where a designer may require the modification of a particular class-version depicting certain details of the real world that are not depicted in the latest class-version. The proposed approach accounts for six types of schema changes: change of attribute semantics (such as attribute domain, range, units of measurements), addition or removal of (derivable/non-derivable) attributes, addition/removal of class (with/without inheritance complications). An example of versioning a class is illustrated in Figure 2.1, where version CV_1 of class `Person` defines attribute `Salary` to be counted in Dollars, while the newer version CV_2 counts the salary in Pounds. Also, The attribute `Name` of CV_1 is dropped in CV_2 and replaced by two attributes instead, `FirstName` and `SurName`. We notice that the later attributes are derivable from `Name` by projecting name parts, and vice versa by concatenating those parts. Attribute `Age` is not declared in CV_2, so it is initialized with a default value (25 in the example of Figure 2.1) as objects are mapped from CV_2 to CV_1. The effectiveness of update/backdate functions is utilized here to convert `Salary` values of objects from either version to the other, and to materialize the relationship between naming attributes of both class versions.

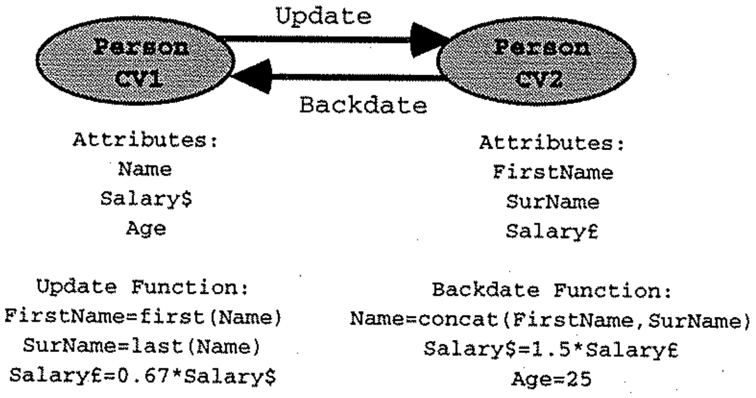


Figure 2.1: Changing Units of Attributes in CLOSQL

CLOSQL adopts dynamic conversion of objects from their current class version to different versions of the class. To query a class version other than the current version of the user, objects of the non-current version have to be transformed first to the current version definition. Then, qualifying objects migrate to the current class version and reside there. However, this strategy may be inefficient, from a performance point of view, if large object-bases are employed. In the context of object mapping, CLOSQL addresses the case of losing attribute values if the object is converted to other class version not defining that attribute. A problem occurs if the object is to be converted back to its original class version. CLOSQL handles this problem by storing attribute values that may be lost. However, this solution is implemented apart from the update/backdate function approach, so objects may still lose their attribute values if an attribute is removed from an intermediate version along the forward or backward route of conversion. This is illustrated in Figure 2.2 where an object is being transformed from class version CV_4 to CV_6. We notice that attribute `Sex` of CV_4 is dropped in CV_5 but is redefined in CV_6 as attribute `Gender`. In this case, the storage manager will not recognize the equivalence of both attributes as no update or backdate functions in CV_5 correlate those attributes. This leads to the inevitable loss of attribute data. So, we come to the conclusion that part of the database may be rendered inaccessible for some user queries.

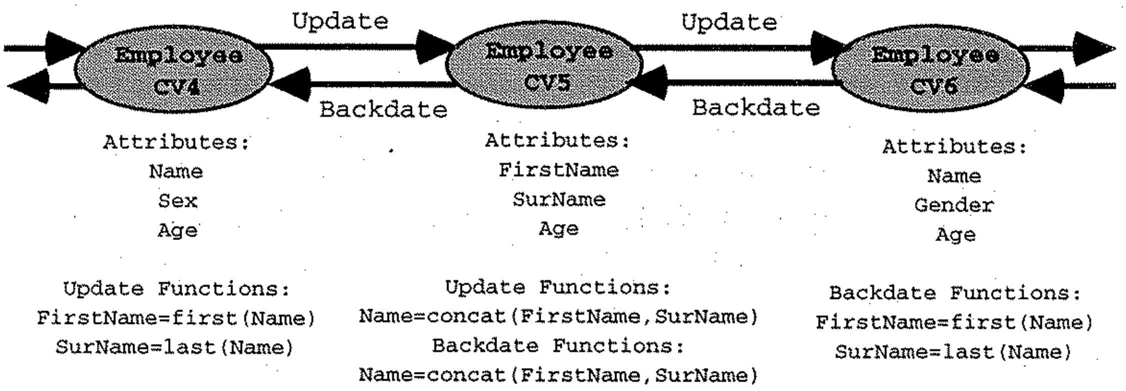


Figure 2.2: Data Preservation in CLOSQL

The ENCORE system [Skarra 86] defines a *version set interface* for each class to contain the union of attributes, methods and domain values of all versions of that class. In this mechanism of class versioning, changes to a class definition result in automatic creation of new versions of the class and its subclasses. Each class will have its new version included in its version set interface. An object remains bound to a particular class version unless it is explicitly coerced to another version. In doing so, the object is bound to lose some of its data if the target class version does not define all of the existing object attributes. A program written to communicate with the version set interface will always find a reference to the attribute it requires, even if it does not exist for objects of the same class version. ENCORE updates the version set interface of a class with newly introduced class methods and relaxed constraints on the domain of an attribute. Strengthening (e.g., narrowing) the domain of an attribute, however, does not affect the version set interface. ENCORE defines a handler in the class version for every attribute that appears in the version set interface, but not in that version of the class. The handler can return a default value for its attribute if the object does not provide value. The handler also detects and warns of write/read errors that emerge when a program attempts to update an object's attribute beyond its class version domain, or when the read object has attribute values unexpected by the program. Unlike CLOSQL, the ENCORE system supports a hierarchy of class versions, but lacks the support of changes in attribute semantics such as domain type and units of measurements. Like CLOSQL, the ENCORE suffers a weakness in preserving object data if it is coerced from one class version to another. Figure 2.3 illustrates an example of versioning class `Employee` in ENCORE.

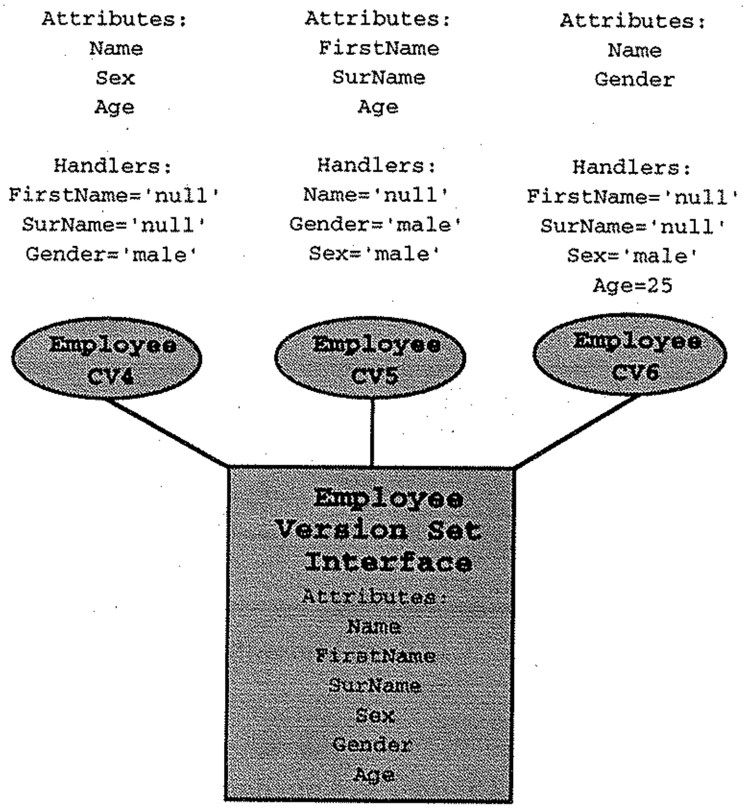


Figure 2.3: Class Versioning in ENCORE

Clamen [Clamen 92, 94] introduces a general and functional mechanism to support schema evolution and *instance adaptation* for centralized and distributed object-oriented database systems. Instance adaptation allows multiple representations of objects to persist simultaneously. It supports multiple interfaces to objects where each interface is related to a particular class-version. An object is represented by the disjoint union of facets each of which encapsulates the state of the object for a different class-version. A facet can be accessed accordingly with its related interface. Clamen divides attributes into four groups: **Shared**, common to multiple class-versions; **Independent**, cannot be affected by any modifications to attribute values in the other facets (e.g., `IdNumber` in Figure 2.4); **Derived**, can be derived directly from the values of attributes in other facets (e.g., `Name`, `FirstName`, `SurName` in Figure 2.4); and **Dependent**, affected by changes in attribute values in other facets but cannot be computed solely from those values (e.g., `Salary$`, `Salary£` in Figure 2.4 as deriving either attribute's value requires the exchange rate in addition to the other attribute's value). The consistency between the facets can be maintained by providing *dependency functions* acting on facets to propagate any update to their dependent and derived attributes whenever a related attribute is modified. Read and write methods (similar to dependency functions) are proposed to simulate derived and dependent attributes in order to save their slot space in object facets.

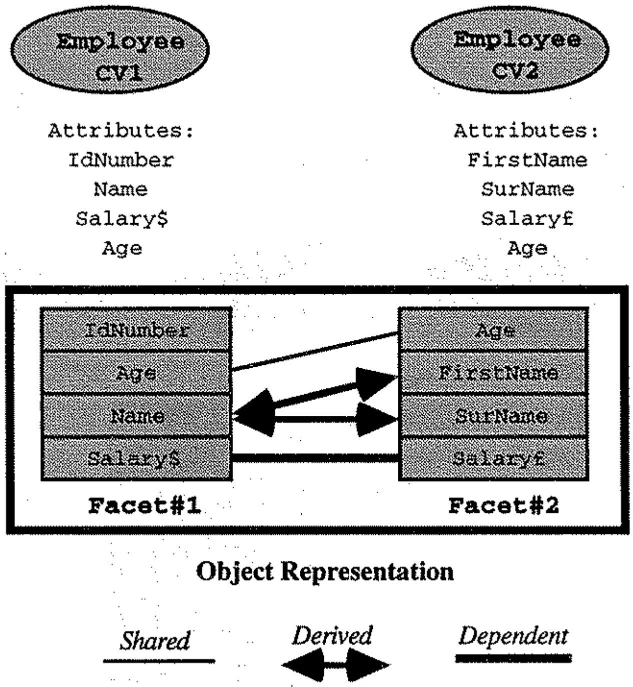


Figure 2.4: Clamen's Approach of Schema Evolution

Shared attributes, on the other hand, may occupy a single slot common to all facets in a multifaceted representation. This is intended for further optimization of space and update cost. In such case, copying the new value into that slot, is all that is required. One drawback of this approach is the overhead of propagating a modified attribute value to multiples of facets to update their dependent attributes values, even with lazy propagation being employed. The approach requires the clustering of object facets in order to minimize read and write overhead of object data. However, with the distribution of object facets between several databases at different sites, such clustering would not be feasible and the cost of propagating updates of object data to its facets would impose additional overhead.

Removing a facet can require considerable changes to existing dependency functions and read/write methods as those functions and methods relate pairs of facets. It may also cause the loss of dependent object data in other facets if simulation is employed. Representing shared and derived attributes directly without simulation, on the other hand, would sacrifice space for performance. Clamen does not consider changes to the range of attributes domain which would affect shared and derived attributes and would raise a problem of incompatibility between attributes.

2.4.7. Schema Versioning

The class versioning approach of realizing schema evolution maintains a single schema for the database within which modifications to the inheritance hierarchy and class semantics would lead to the creation of versions of affected classes. Schema versioning, on the other hand, creates a new version of the database schema whenever such modifications take place. In this case, the user refers to specific versions of the schema rather than to versions of individual classes. An objects created according to a particular version of the schema would be an instantiation of its class declared in that schema version. Different implementations of this approach exist that provide accessibility to objects in different schema versions. Here, we review examples of such implementations.

Kesim [Kesim 93] incorporates object and schema versioning in her proposed approach for modelling temporal aspects of objects. This work formulates change in the context of a historical database which stores all past states of objects in the database. The schema evolution is kept simple and excludes changes to attribute semantics. Versions of objects or schema are referred to by giving time points: one for the object time, and one for the schema time. In this approach, object versioning can be linear or complex (parallel).

Kim and Chou [Kim 88b] describe schema versioning in ORION where a schema version may be derived from the basic database schema or from other schema versions in such a way that the underlying database and its schema versions would be organised in a hierarchy. Object versioning is also employed where each schema version would have its own versions of database objects. Accordingly, an instance deleted in one schema version may still exist in other schema versions.

Byeon and McLeod [Byeon 93] unify both schema views and schema versions in their novel concept of *virtual database* . The schema of a virtual database may have three kinds of classes: *imported* which is copied with/without modification from some bases of the virtual database; *local* which is newly created in the virtual database; *derived* which is extracted from other classes whether they are imported, derived, or local. An instance of the virtual database is either: *imported* which is an instance copied from some bases of the virtual database or *local* which is newly created in the virtual database. Since a virtual database may be created on top of one or more virtual databases, the underlying database as well as its virtual databases are organized into a directed acyclic graph (DAG) rooted at the database. Like database views in [Bertino 92], the virtual database approach suggests exclusive manipulation of some objects apart from existing database schema. It also requires the propagation of object updates to the virtual databases that have derivatives of the object or to imported objects. In some cases, it requires creating the object of an imported class in its original database, then copy the object to the virtual database that imported the class. Such handling of objects would lead to redundancy of data within the system and would impose additional overheads for system operation.

2.5. Parallelism in Object-Oriented Databases

Parallel processing can play a significant role in serving object-oriented systems, be they programming or database environments. Issues of parallelism in relational databases have been the basis of many of the approaches currently employed in OODBMS to improve their performance. Different views of the potential of parallel processing in such environments have been expressed in the literature. In this section, we review those views, part of which has been considered in building our object-manipulation prototype (to be discussed in Chapter 7) and in our performance investigation conducted in Chapter 8.

Pears and Gray [Pears 92] consider the following approaches for OODBMS:

- Pipelining of operators in query dataflow graphs using separate processing nodes allocated to each operator.
- Independent task execution in which operators at the same level in the dataflow graph are executed simultaneously. The employment of vertical fragmentation can also increase concurrent operations and reduce communication costs.
- Horizontal partitioning of data to enable processing only relevant data and associating a query operator with each partition, as in the OFC system [Lam 89].

De Groat [De Groat 92] outlines the opportunities within OODB for exploiting parallelism at the following levels:

- **The OODBMS**- storage techniques for value-based retrieval (based on indexes or on associative techniques for both memory and disks as in the IFS/2 [Lavington 87, 88, 91, 92]); special processors for activities such as locking; better backup and recovery techniques.

- **The stored objects**- parallel execution of object methods and their sub-methods as in the PKBZ system [Fathy 91].

- **The query engine**- applying parallelism at the low-level set operations such as union, intersection etc., or for the evaluation of the query graph as in [Thakore 94].

- **The application code**- parallelizing the application code is of interest to the extent that it interacts with the parallelization of the OODB itself.

Thakore and Su [Thakore 94] consider vertical partitioning of data to improve retrieval and update parallelism and to avoid access to (and consequently locking of) data not needed by the query. Chorafas and Steinmann [Chorafas 93] perceive a potential of associative techniques for database management with implementation advantages such as the elimination of indexing schemes; a resulting decrease in storage requirements; the capability of querying any field; greater flexibility in adapting to changing requirements. Kim [Kim 90a] recognizes:

- **Path Parallelism**, employing concurrent processing of different navigational paths in the query graph.

- **Node Parallelism**, in which processing various nodes of simple predicates in the query graph is performed in parallel, as in the OFC system [Lam 89].

- **Class-Hierarchy Parallelism**, in which subclasses comprising the class hierarchy are processed in parallel.

2.6. Manipulating Complex and Inter-Related Objects

Manipulating complex queries in object-oriented databases requires database management systems to support the navigational feature of queries, while retaining the set-operation capabilities of relational databases. Resolving queries in object-oriented database, thus, have been categorized into three methods of node traversal in a query graph. Those methods are:

- **Forward traversal-** classes in the query graph are traversed (or visited) in a depth-first order starting from the root of the query graph, and following through the successive domains of each complex instance variable.

- **Backward traversal-** classes in the query graph are traversed starting from the leaf classes of the query graph, followed by their parents, working toward the root class.

- **Mixed traversal-** both forward and backward traversal modes are utilized to resolve queries that primarily contain Boolean operators.

Qualifying objects of the root class in a query graph will be those with successful traversal along all visited paths . In addition to the order of visiting classes in the query graph, the way of retrieving complex-class instances, during query processing, must also be considered. Bertino and Martino [Bertino 91] recognize two strategies to achieve this requirement:

- The **nested-loops** evaluates each instance of a class in the query graph according to the query predicates. If the instance qualifies, it is passed to the child node (in forward traversal), or to the parent node (in backward traversal) in the query graph. This process continues until all nodes of the query graph are traversed, and a single complex object is produced for the root node. The process is repeated to evaluate all instances of the root complex class.

- The **sort-merge** processes all instances of a node in the query graph at the same time with respect to the predicates of the query, and passes qualifying objects to the proper node in the query graph. At that node, objects are sorted and merged with the node's class instances according to a join constraint, and the resulting instances are evaluated according to query predicates related to the node. This process is repeated for all nodes in the query graph and ends with the production of all qualifying root class instances at the same time.

The advantage of the latter strategy over the former is that the storage pages containing instances of a node's class are accessed only once for a query, thus improving query response time. Bertino and Foscoli [Bertino 92] provide mathematical functions for cost evaluation of the different combinations of query graph traversal modes and object retrieval strategies. Blasgen and Eswaran [Blasgen 77], on the other hand, analyzed both join algorithms for relational databases in uniprocessor systems. The results show that in the absence of indices, a sort-merge algorithm performs best. In multiprocessor systems, the nested-loops algorithms works by joining each unit of one (the outer) relation with all of the units in the other (the inner) relation each of which may be allocated to a separate processor. The multiprocessor sort-merge algorithm, on the other hand, employs a parallel sort of both relations on the joining attribute. This is followed by a uniprocessor merge on the joining attribute to perform the join. Intuitively, the nested-loops algorithm should outperform the sort-merge since the amount of parallelism that can be attained is high. When sorting in parallel, one may be able to start with a large number of processors, but after each stage the number of processors decreases and the amount of data examined by each processor increases until in the final stage one processor must examine the relation in its entirety. Both nested-loops and sort-merge methods can benefit from join indices proposed for either object-oriented or relational databases. Valduriez proposed join indices that are adaptive to parallel execution [Valduriez 87]. Bertino and Guglielmina also propose and evaluate the performance of a path-index approach that associates the values of a nested attribute with the instances of the class root of a given aggregation hierarchy [Bertino 93b]. Valduriez and Gardarin [Valduriez 84] propose hashing as well as nested-loops and sort-merge methods for the join of relations in multiprocessor systems. The methods may well be adapted for object-oriented databases. In their proposal, Valduriez and Gardarin analyzed algorithms for the three methods in the SABRE multiprocessor system [Gardarin 81]. The nested-loops turned-out to be the simplest, with execution time inversely proportional to the number of processors. It worked best when the number of processors was high. The sort-merge was more complex but performed better as the operand relations became larger. The hashing algorithm is better when the number of matching tuples in the larger relation is small. In this case, using bit arrays resulted in avoiding much useless access to the hashed file. Boral and DeWitt [Boral 81] verified the general superiority of the parallel nested-loops algorithm in their analysis of the performance of DIRECT parallel machine. They demonstrated how the machine efficiently employs nested-loops algorithm without maintaining indices.

2.7. Parallel Architecture Role In Object-Oriented Systems

Single-instruction-single-data (SISD) known as von-Neumann organization, single-instruction-multiple-data (SIMD), and multiple-instruction-multiple-data (MIMD) are architectures adopted for realizing parallel computer systems. Some of the existing systems exploit such architectures in supporting object-oriented programming and OODBMS. In this respect, we find a diversity of system directions: Chancer library support for object-oriented Occam programming [Chalmers 89], PRESTO run-time library for multiprocessor distributed object-oriented applications [Bershad 88]; associative object memory supporting address translation and constant management of live objects [Hyatt 93], parallel associative access to objects with the IFS/2 [Lavington 87, 88, 91, 92], parallel object manipulation using a multiprocessor architecture in PKBZ OODBMS [Fathy 91], distributed parallel object manipulation with von-Neumann architecture in DOOM [Odijk 87], pipeline and data-flow processing of associations between objects in the OFC [Lam 89, Lee 89]; specialized hardware demonstrated by the RISC processor (SOAR) for Smalltalk-80 system [Ungar 87], Rekursiv architecture [Pountain 88, Harland 86, Harland 88]. None of those systems, however, explicitly addresses issues related to schema evolution.

2.8. Summary

In this chapter, we conducted reviews and presented the basics of two directions of research: schema evolution in object-oriented databases and parallel processing potential in supporting that functionality. For schema evolution, we provided a background of the object-oriented paradigm and database systems. We then discussed the different approaches of supporting schema evolution and provided some prominent examples of each approach. Those approaches are: *schema and class modification*-GemStone [Penney 87, Bretl 89], ORION [Kim 90b, Kim 90c], OTGen [Lerner 90], Sherpa [Nguyen 89]; *database views* [Bertino 92, Kim 88, Scholl 90]; *object versioning*-ORION [Chou 88], AVANCE [Bjornerstedt 88, 89], ISO/IEC [ISO 93], [Katz 90, El-Sharkawi 90]; *class versioning*-CLOSQL [Monk 92, 93], ENCORE [Skarra 86], Clamen [Clamen 92, 94]; *schema versioning*-ORION [Kim 88b], [Byeon 93, Kesim 93]. In the dissertation, we intend to tackle performance issues related to schema evolution and investigate the potential of parallel processing in improving OODBMS performance. So, we presented in this chapter what the literature perceive as prospect areas for parallelism in object-oriented environments. We also reviewed the methods of manipulating objects in sequential and parallel systems upon which our investigation will be based. Finally, we referred to existing systems that exploit parallel processing in object-oriented paradigm: Chancer [Chalmers 89], PRESTO [Bershad 88], [Hyatt 93], IFS/2 [Lavington 87, 88, 91, 92], PKBZ [Fathy 91], DOOM [Odijk 87], OFC [Lam 89, Lee 89], SOAR [Ungar 87], Rekursiv [Pountain 88, Harland 86, Harland 88].

From the conducted surveys, it was clear that both directions of the research have not been addressed explicitly by any of the existing systems. Therefore, in the following two parts of the dissertation, we shall tackle the limitations of existing systems supporting schema evolution, and then we shall deal with the performance issues related to our proposed solution to such limitations.

This page is intentionally left blank

SUMMARY OF PART I

In this part, we presented an introduction to the thesis. We discussed our research motivations, objectives, and achievements. As we target schema evolution from both functionality and performance points of view, we presented a background and reviewed research efforts related to both domains. The background topics included the concepts of the object-oriented paradigm, the definition and the role of object-oriented database systems, and the methods by which database schema may evolve. We quoted key references from the literature to those topics. With regard to the performance domain, we reviewed research efforts exploiting parallel computer architecture to the advantage of object-oriented programming and database systems. We elaborated on some of the referenced work to highlight the contribution of our research as we progress in discussions throughout the thesis.

The following part of the thesis will deal with the functional domain of schema evolution from our research perspective. We follow that part with a dedicated part for the performance domain that represents our analysis of parallel processing potential to support schema evolution.

This page is intentionally left blank

Part II

The Class Versioning Approach

In this part, we present our proposed approach for supporting schema evolution and discuss its related issues. In chapter 3, we present the concepts of the approach and its main features. In chapter 4, we discuss different aspects related to the approach including the applicability of object storage models; the manipulation of class-version instances; query optimization; programming languages aspects. Then we study in chapter 5 the applicability of our approach to other data models through case studies.

This page is intentionally left blank

Chapter 3
Concepts and Features

In this chapter, we shall discuss the concepts and main features of our proposed class-versioning approach. The first section presents the objectives to be achieved with this approach, followed by revisiting the IS-A data model in section two to describe an extension to its semantics. In section three, we present a taxonomy of the supported schema changes and discuss how they affect complex classes. Section four deals with the issue of compatibility between different versions of a class. Section five defines class-versions and version-collections and discusses their role in schema evolution. Section six explains how version-collections are being formed. Section seven discusses transformation-path sensitivity which is concerned with the preservation of object data when transforming it between class-versions. Section eight presents the novel concept of intermediate class-versions which contributes to the preservation of object-data. In section nine, we describe the different schemes for configuring class-versions in a graph of version-collections. In section ten, we give some examples of schema evolution based on the class-versioning approach, then in section eleven, we discuss object migration between class-versions, its motivations and safeguards. Section twelve elaborates on how schema-versioning is achieved through the framework of class-versioning. Finally, we derive some conclusions emerging from the foregoing discussions.

3.1. Objectives of the Class-Versioning Approach

The proposed approach aims to support and achieve the following objectives:

- ***Dynamic access to an object using any version of its class:*** This means that user applications designed using an existing or a new schema can access objects created under other schemas without the need to either reorganize the database to a particular schema, or alter the applications to match the schema that created the objects. As the current object structure is maintained, it is dynamically mapped to that under the used schema while the application is running.

- ***Object migration between versions of the same class:*** Considering the case where an object is frequently accessed by a schema different from that under which it was created, we require the facility to migrate the object to the corresponding class in the active schema and eliminate the overhead of repeated dynamic object mapping.

- ***Information conservation:*** We need to avoid losing object data or rendering objects inaccessible in the course of querying the database or mapping objects between different versions of their class.

- ***Schema and class changes:*** We seek the support of semantic changes to attribute definition, alongside the traditional changes supported by existing OODBMSs. That is, we require the provision of a wider range of supported changes to the database schema.

- ***Hierarchical pattern of versioning classes:*** We aim at supporting hierarchical versioning of classes, where it is possible to modify any existing class-version and create a new version accordingly. That is, class-versioning should not be restricted to the latest class-version as in the linear versioning approach. Supporting hierarchical versioning of classes can be a necessity for application domains such as CAM/CAD.

On the other hand, the approach does not attempt to support: active OODBMS (we target passive OODBMS only); versioning of objects.

3.2. The Data Model Extension

In the basic object-oriented data model, a class **C** comprises a set **A** of attributes **a** defining its structure and a set **M** of methods **m** defining its behaviour. Each attribute has a name and a domain associated with it. Ideally, full support of abstract data typing requires the operations associated with an abstract data type to be ***complete*** and ***correct*** . To help the programmer better express the behaviour of abstract data types, object-oriented programming languages need to provide constructs to indicate the ***constraints*** that test the correctness or completeness of the abstract data type. Khoshafian mentions two approaches to providing such language constructs [Khoshafian 93] :

- Placing constraints on objects and instance variables such that access and update constraint routines are executed when manipulating instances of the abstract data type. Those constraint routines are incorporated into the definition of the class and may be associated with either the object as a whole or a particular instance variable of the object. The latter strategy is supported in some languages such as C++ [Gorlen 91] where the value assigned to an attribute is checked for a match with the attribute type and the value is converted if the match is not achieved. For example assigning a float value to an integer attribute causes the truncation of the fraction part of the value.

- Associating preconditions and postconditions with the operations (methods) of the abstract data type rather than with the objects. This approach is taken in Eiffel [Meyer 88].

In our approach, we extend attribute semantics from merely characterizing the type of values that may be assigned to the attribute (i.e., integer, float, character etc.) to specifying the range, type, and units of measurement (if any) of those values. However, the support for constraint checking is not guaranteed to exist for all programming languages. C++, for example, supports only standard domains for user-defined types based on the language generic types. For instance, the user cannot specify a range of values between 1000 and 8000 for `Employee`'s `Salary` in Pounds (£) based on type ***integer*** . In this case, the implied range of values is automatically set between -32,767 and +32,767 and values beyond that range are truncated. To cope with this deficiency, the user would have to adopt the second approach where constraint routines become part of his application. With the introduction of object-oriented database systems, there is a way around this problem by embedding constraint checking in the DBMS itself while extending attribute semantics as indicated above. This in effect would move part of the application specifications and the programming language declarations to the DBMS. In the following section, we shall demonstrate how this additional feature of OODBMS is integrated in the class-versioning approach.

3.3. Taxonomy of Supported Schema Changes

In our proposed approach, the supported schema changes, depicted in Figure 3.1, are categorized according to their influence on the configuration of class-version hierarchy. Schema changes are divided into two distinct groups:

Group-I presents changes that do not affect attribute semantics within a single class. Such changes are:

- changing class name.
- changing class methods (either signature and/or implementation).
- addition/removal of an attribute (derivable/non-derivable).
- addition/removal of a member-class within a leaf complex-class.
- addition/removal of a sub-class.

Group-II presents the changes affecting attribute semantics. These are: the attribute's name, units of measurement, domain type, range of values, or default value. Attribute semantics may be expressed in BNF as:

```
<attrib_dcl>    := attribute <attrib_name> domain_type <type>
                   [<domain_range>] [domain_units <units>]
                   [default_value <value>]

<domain_range> := [upper_bound <value>] [lower_bound <value>]|
                   discrete_extension <values>

<values> := <value> | <value> <values>

<type>    := <simple_type> | <struct_type>

<value>   := <number> | <string>

<units>   := <string>
```

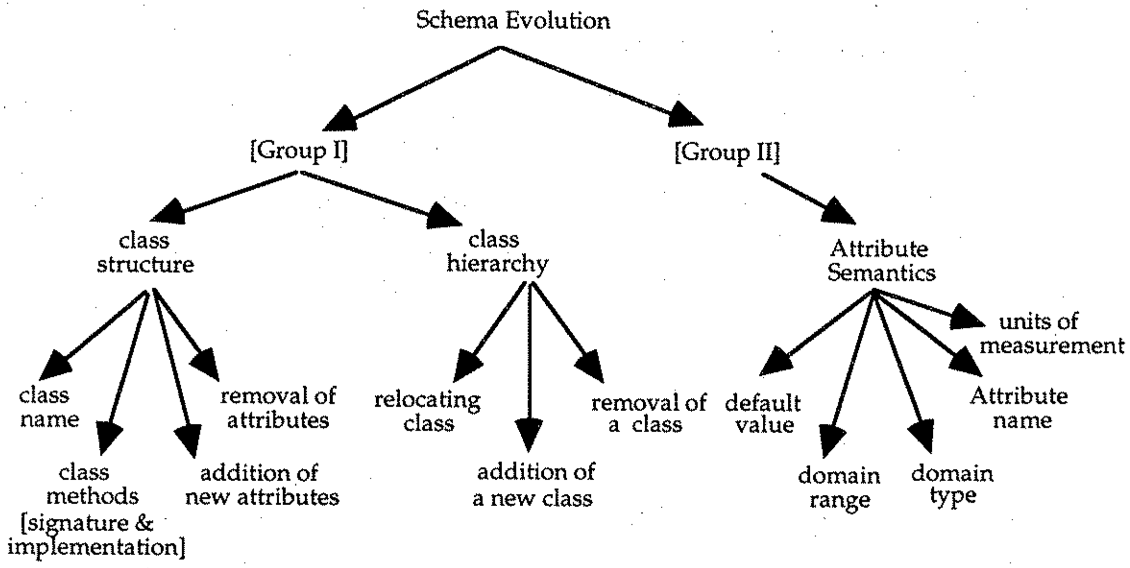


Figure 3.1: Supported Changes in Database Schema

Changes of either group may need to be propagated down the class hierarchy if they are performed on a non-leaf class. This in effect will cause versioning all affected classes in the schema. If the changes to the schema target a simple leaf-class, then only that class needs to be versioned. Clamen [Clamen 92, 94] avoids the propagation of changes to subclasses (in some cases of schema evolution) by allowing versioning the class hierarchy where subclasses become subtypes of class-versions and not classes. We do not follow this approach since the inheritance characteristic of the data model dictates that subclasses should evolve in concert with their supertypes. Versioning a complex class, on the other hand, does not result in versioning its constituent classes. However, modifying a constituent class, with either a Group-I or Group-II change, has the effect of changing the semantics of the complex attribute it represents. So, in that case, the complex class should be versioned to reflect the Group-II change, while the constituent class will be versioned according to its modification group. Examples of schema evolution in the context of the class versioning approach are presented in Section 3.10.

As we mentioned in Section 3.1, our approach is mainly concerned with the mapping of object structure between class-versions. Following object transformation, class methods are activated in the destination version according to the messages received by the transformed object. This process takes place within the user application where objects are normally manipulated.

3.4. Attributes Compatibility

Definition 3.1

A simple attribute can be assigned only atomic values while a complex attribute, whose domain is a set of objects, can specify domain values as `ObjectIDs`. An attribute is identified by its name and the class that created it, and is referred to by a 'low case' symbol that implies such information about the attribute. The domain of an attribute $\mathbf{a_i}$ is a set $\mathbf{D_{a_i}}$ representing the domain values of $\mathbf{a_i}$, and implies the attribute's type and units of measurement (if any).

Definition 3.2

We call two attributes $\mathbf{a_i}$, $\mathbf{a_j}$ ***identical*** , if and only if they have the same names, domains, and default values. In other words, attributes are isomorphic if they agree on their semantics. This is denoted by the equality operator '=' such that $\mathbf{a_i} = \mathbf{a_j}$.

The Object Database Standard [ODMD-93] considers two objects to have the same type (i.e., class) if and only if they have been declared to be instances of the same named type. Objects that have been declared to be instances of two different types are not of the same type, even if the types in question define the same set of properties and operations. Also, type compatibility for objects follows the subtyping relationships defined by the type hierarchy. That is: an object of a subtype is compatible with its supertypes but the reverse is not possible. When we consider those features of the data model, we distinguish between types (or classes) of the same schema version and versions of a class in different schema versions. The first case is directly subjected to the standard constraints, while the second should not since class versions implicitly refer to the same real-life entity even though they might have different properties and behaviour. According to the standard, no implicit conversion of objects between compatible types is provided by the model.

Therefore, such conversion should be enforced by the OODBMS. In what follows, we shall present our point of view regarding attribute compatibility that would govern object conversion between class versions.

Definition 3.3

We call two attributes $\mathbf{a_i}$, $\mathbf{a_j}$ ***Fully-Compatible*** , denoted by $\mathbf{a_i} \equiv \mathbf{a_j}$, if we can define two functions: α for $\mathbf{a_i}$, and β for $\mathbf{a_j}$ such that:

$$\forall\ x \in \mathbf{D_{ai}}\ ,\ y \in \mathbf{D_{aj}} \text{ we have } \alpha(\beta(y)) = y \text{ and } \beta(\alpha(x)) = x$$

where D_{ai} and D_{aj} are the domains of a_i and a_j respectively.

In other words, the full-compatibility constraint requires the source domain to be isomorphic to the destination domain upon transformation such that each value in the domain of either attribute has a corresponding value in the other attribute's domain. The definition assumes the provision of adequate mapping functions to achieve the compatibility of class-versions, and it ensures that attribute data is not lost when mapping objects between class-versions or version-collections (this will be discussed in detail later on).

Adopting this definition does not allow changing attribute's range of values for the same domain type and units of measurements. For example, changing the currency of an employee's `Salary` from pounds **£** to dollars **$** requires scaling the domain's decimal range by the agreed exchange rate. However, it is not possible to change the salary range for the same currency. The latter change may be required if the versioned `Employee` class ought to represent either `Junior-Staff` or `Senior-Staff` class only.

In the remainder of the chapter, we shall use the shorthand presentation $\alpha\mathbf{D_x}$ to depict a transformation function α operating on domain $\mathbf{D_x}$ of attribute $\mathbf{x}$.

Definition 3.4

We call two attributes $\mathbf{a_i}$, $\mathbf{a_j}$ ***incompatible*** , denoted by $\mathbf{a_i} \neg\equiv \mathbf{a_j}$, if it is not possible to declare transformation functions between the attributes to fulfil the compatibility constraints of Definition 3.3, or if the declared transformation functions α mapping $\mathbf{a_i}$ to $\mathbf{a_j}$ and β mapping $\mathbf{a_j}$ to $\mathbf{a_i}$ produce disjoint domains, That is:

$$\alpha\mathbf{D_{ai}} \cap \mathbf{D_{aj}} = \beta\mathbf{D_{aj}} \cap \mathbf{D_{ai}} = \varnothing$$

An example of incompatible attributes is having `Salary£` with range of values £1000-4000, and `Salary$` with range of values $15,000-30,000. Having an exchange rate £->$ of 1.5 would map `Salary£` to the domain $1500-6000 and `Salary$` to the domain £10,000-20,000. In either case, it is clear that the transformed domain have no common values with the destination domain, thus making both disjoint.

Definition 3.5

A general form of attribute compatibility, denoted by $\mathbf{a_i} \nabla \mathbf{a_j}$, implies that:

$$\alpha \mathbf{D_{ai}} \cap \mathbf{D_{aj}} \neq \varnothing \text{ and } \beta \mathbf{D_{aj}} \cap \mathbf{D_{ai}} \neq \varnothing$$

where α maps $\mathbf{a_i}$ to $\mathbf{a_j}$ and β maps $\mathbf{a_j}$ to $\mathbf{a_i}$.

This is the complement of Definition 3.4 as it requires attributes domains to have at least some values to be mapped between themselves. The mapping functions α and β would provide the *'undefined'* value for any mapped value that is out-of-domain. This definition implies the possibility of having two *partially-compatible* attributes for which only part of their domains can be mapped to each other.

Partial compatibility can be illustrated with attribute `Colour` whose domain given by `{White,Red,Blue,Black,Grey}` can be directly mapped to the domain `{C1,C2,C3,C4,C5}` where `White` corresponds to `C1` and so on. Having two versions of this attribute with domains `{White,Red,Blue}` and `{C3,C4,C5}`respectively means that both domains have only a single value in common, that is `Blue` (or `C3`). This makes both versions of the attribute *partially-compatible* with mutually exclusive values.

Definition 3.6

We define attribute $\mathbf{a_1}$ as ***derivable*** from attributes $\mathbf{a_2\, a_3 \ldots a_n}$, denoted by $\mathbf{a_1} \lrcorner [\mathbf{a_2}, \mathbf{a_3}, \ldots, \mathbf{a_n}]$, if the domain-values of $\mathbf{a_1}$ can be computed or produced from the domains of the attributes contained in the list according to some criteria. That is:

$\mathbf{a_1} \lrcorner [\mathbf{a_2}, \mathbf{a_3}, \ldots, \mathbf{a_n}]$ if there is a function ψ such that:

$$\psi : \mathbf{D_{a2}} \times \mathbf{D_{a3}} \times \ldots \mathbf{D_{an}} \rightarrow \mathbf{D_{a1}} \, [\forall\, x \in \mathbf{D_{a1}}, \exists\, y_2 \in \mathbf{D_{a2}}, y_3 \in \mathbf{D_{a3}}, \ldots y_n \in \mathbf{D_{an}}] \text{ such that } \psi(y_2, y_3, \ldots y_n) = x$$

A derivable attribute, from our approach perspective, represents a property of the object and has storage allocated for it in the object. It may be assigned a value using a member-method or directly (if it is declared as `Public`) by copying a value from an application variable. An example of derivable attributes is attribute `FullName` which can be derived from two attributes `FirstName` and `SurName`, and vice versa. This can be achieved by concatenating the last two attributes (with a blank separator in-between). Extracting either `FirstName` or `SurName` from `FullName` is possible by projecting the first or the last word of the full name.

Modifying attribute's domains in the ORION and GemStone as well as in other systems has been subjected to a schema consistency rule that requires the domain of an inherited subclass attribute to be the same as that specified in its origin superclass, or be a subset of it. Also modifying an inherited attribute's domain can only be by means of generalization so as to make existing objects accessible to the modified class definition.

When considering versioning of classes, the compatibility between super-classes and sub-classes should be reserved as well as between class-versions. We adopt a complimentary view that permits changing the attribute's domain, either by generalization or specialization. Before we elaborate on that view, we need to present some definitions relating attributes domains.

3.5. Class-Versions and Version-Collections[1]

Definition 3.7

A class-version **CV** is a modified definition of an existing class **C**, introducing some changes to either the class structure or its behaviour, or both. The original class definition is considered to be the first version of the class and is denoted by CV_0. We define version **i** of class **C** as $\mathbf{CV_i = \{A_i , M_i\}}$ where $\mathbf{A_i = \{a \mid a \text{ is an attribute of } CV_i\}}$ is the set of class-attributes defined in version **i**, and $\mathbf{M_i = \{m \mid m \text{ is a method of } CV_i\}}$ the set of class-methods defined in version **i**.

A class can have more than one class-version, each of which may belong to one or more schemas.

The version-collection **CL**, on the other hand, is an overall view of the class structure to the system[2] and comprises the union of all attributes defined in the version-collection related class-versions. Collections of class-versions are different from object collections. The first is related to the database schema while the second is a grouping of class instances. An object created according to a particular class-version can be mapped directly to its version-collection definition. The database can have more than one version-collection for a class, each of which groups a different collection of class-versions.

[1] In the definitions that follow, we shall use the variables i, k, m, n to represent non-negative integers.

[2] in contrast to the version-set in ENCORE system that acts as an interface between the application and class instances.

Mapping objects from one version-collection to another is performed by applying inter-version-collection functions, which we call *Forward / Backward* transformers[3]. The Forward transformers map a version-collection to its immediate successor version-collections, while the Backward transformers aim at the immediate ancestor version-collection instead. In this way, the system can make the instances of each class-version available to other class-versions of the same class.

Definition 3.8

Version-collection **k** of class **C** is defined as $\mathbf{CL_k} = \{\mathbf{A_k}, \mathbf{T_k}\}$ where $\mathbf{A_k} = \cup\ \mathbf{A_i}$ [$\forall$ $\mathbf{CV_i}$ that belongs to $\mathbf{CL_k}$ where $\mathbf{A_i} \in \mathbf{CV_i}$] is the set of version-collection attributes. For each successor version-collection $\mathbf{CL_i}$ of $\mathbf{CL_k}$, let $\mathbf{T_{ki}} = \{\mathbf{t} \mid \mathbf{t}$ is a function mapping one or more attributes of $\mathbf{CL_k}$ to an attribute of $\mathbf{CL_i}\}$. We define $\mathbf{TF_k} = \cup\ \mathbf{T_{ki}}$ to be the set of forward transformers mapping attributes of $\mathbf{CL_k}$ to the attributes of its descendent version-collections. Let $\mathbf{TB_k} = \{\mathbf{t} \mid \mathbf{t}$ is a function mapping one or more attributes of $\mathbf{CL_k}$ to an attribute of the predecessor version-collection of $\mathbf{CL_k}\}$ be the set of backward transformers. Then we define $\mathbf{T_k} = \mathbf{TF_k} \cup \mathbf{TB_k}$. The first version-collection created for a class is denoted by $\mathbf{CL_0}$ and houses the first version of the class ($\mathbf{CV_0}$).

3.6. Forming Version-Collections

In this subsection, we explain how and when class-versions can join a version-collection. We also describe the presentation of version-collections and how to correlate them.

Definition 3.9

Each version-collection has associated with it a membership-set **M** specifying which class-versions belong to the collection. That is:

$$\forall\ \mathbf{CL_k}\ \exists\ \mathbf{M_k} = \{\mathbf{i} \mid \mathbf{CV_i}\ \textbf{belongs to}\ \mathbf{CL_k}\}$$

Definition 3.10

A version-collection exists if and only if it has at least one member class-version. That is:

$$\exists\ \mathbf{CL_k} \Leftrightarrow \mathbf{M_k} \neq \varnothing$$

[3] similar to the Update/Backdate functions proposed for the CLOSQL system except that they are applied between version-collections and not between class-versions.

This implies that dropping all class-versions of a version-collection results in dropping the latter as well.

Class-versions are grouped into (or become members of) version-collections according to the category of changes they present. Class-versions grouped into the same version-collection should agree on attribute semantics (Group-I changes), while class-versions introducing different attribute semantics (Group-II changes) should be separated into different version-collections. This leads to the following definition.

Definition 3.11

A class-version $\mathbf{CV_i}$ can be a member of a version-collection $\mathbf{CL_k}$ if the following condition is met:

$\mathbf{a} \in \mathbf{A_i} \Rightarrow \exists\, \mathbf{b} \in \mathbf{A_k}$, **a** and **b** represent the same class property, and $\mathbf{a = b}$.

The decision of affiliating a class-version to a version-collection would require knowledge of the relationship between class-version attributes and those declared in the version-collection. Such knowledge, available to a system analyst or a database administrator (DBA) or an application programmer, should indicate whether the conditions of the previous definition are met or not.

For every attribute that appears in the version-collection but not in the class-version definition, a special intra-version-collection function, which we call a *Handler*[4], is provided to assign a default value to that attribute or to derive its value from the values of other object attributes. Those handlers are not part of the class definition and thus are distinguished from its methods.

Definition 3.12

For a class-version we define a set **H** that groups the class-version handlers such that:

$$\mathbf{H_i = \{h \mid h \text{ is a handler of } CV_i\}}$$

Figure 3.2 illustrates two version-collections, each of which has three class-versions. In that example, forward transformers are defined in version-collection CL_1 to map objects of its class-versions to CL_2. Backward transformers are defined in version-collection CL_2,

[4] as in the ENCORE system due to the similarity of their role.

which was created after CL_1, to map objects of its class-versions to CL_1. Handlers are defined in each class-version as necessary.

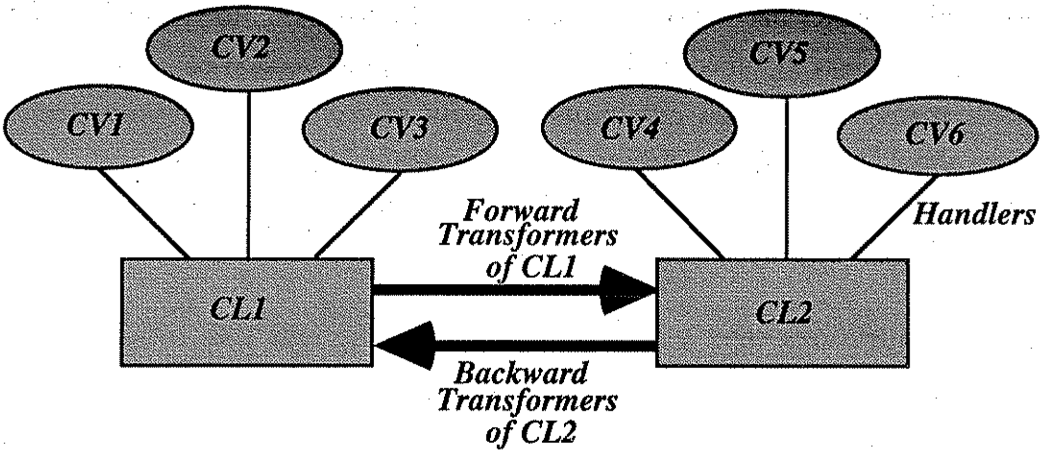


Figure 3.2: Class-Versioning using Version-Collections

We shall use thin edges between a class-version and a version-collection or between two version-collections to illustrate the affiliation of such entities to each other, and thick arrows between version-collections to depict transformers from a source to a destination collection.

We would like to note that omitting such arrows in any of the figures that follow does not imply the absence of transformers, but should only indicate their irrelevance to the purpose of illustration.

Arrows of medium thickness will be used to indicate specialization direction from a superclass to a subclass in the inheritance hierarchy. Constituent classes, on the other hand, are connected to their parent complex-class via edges of medium thickness.

At this point, we would like to distinguish between two approaches of handling the case when attribute semantics, including attribute name, are changed:

- the attribute semantics are considered changed but still model the same real-world property (e.g., attributes `voltage->Kvoltage` where a value of the first is $\frac{1}{1000}$ of that of the second).
- the attribute is considered dropped, and replaced by a new attribute modelling (misleadingly) a different real-world property in such a way that both attributes are mutually derivable using a transformation function (such as the rate $\frac{1}{1000}$).

The second approach qualifies the new class-version to reside in the same version-collection of the original class-version as Definition 3.11 is not violated. However, this would require declaring a handler for every class-version in the same version-collection to cope with the new attribute. This points to the possibility to finding a way around Definition 3.11 to use the same version-collection but at the expense of complicating the management of class-versions. The modified attribute of the first approach, on the other hand, would force the new class-version to reside in a different version-collection leaving other class-versions of the original version-collection untouched. So, when managing schema evolution with our approach, proper knowledge and awareness should enable distinguishing between creating a new attribute, modelling a new real-world property, and modifying an existing attribute to keep the modifications to existing class-versions to a minimum..

Version-collections of a class form a collection-graph **CG** in which any two version-collections declaring transformation functions to each other are connected with an undirected link. Linked version-collections are called neighbours.

Definition 3.13

We define the collection-graph **CG** as:

$$\mathbf{CG} = \{(\mathbf{m}, \mathbf{n}) \mid \mathbf{CL_m}, \mathbf{CL_n} \textbf{ are neighbours, } \mathbf{m} \neq \mathbf{n}\}.$$

Observation 3.1

The neighbourhood relation is ***symmetrical*** in a sense that pairs (m, n) and (n, m) are equivalent and it is sufficient to include either of them in Collection-Graph to imply the other.

Version-collections are transparent to the user, and since they act as an interface between class-versions, a class-version requires only one set of Handlers and this simplifies the definition of class-versions. In the case of managing a hierarchy of class-versions (for example by extending the CLOSQL versioning configuration), a class-version will require groups of transformation functions to map its objects to each of its immediate succeeding class-versions as well as to its preceding class-version. We shall elaborate on the formation of the collection-graph in Section 3.9 based on the definitions presented in the following section.

3.7. Transformation Path Sensitivity

Transforming objects between version-collections should preserve object data created or maintained at the source version-collection, and required at the destination version-collection. In this section, we present definitions related to the transformation of objects between version-collections and express how such routes become sensitive from data preservation point of view.

Definition 3.14

The route connecting any two version-collections $\mathbf{CL_s}$ and $\mathbf{CL_d}$ in the Collection-Graph is called transformation-path, denoted by $\mathbf{P_{s,d}}$. A transformation-path specifies version-collections involved in transforming objects from the source $\mathbf{CL_s}$ to the destination $\mathbf{CL_d}$.

Definition 3.15

Given $\mathbf{P_{s,d}}$ as the transformation path from version-collection $\mathbf{CL_s}$ to $\mathbf{CL_d}$, we call $\mathbf{P_{s,d}}$ a ***safe path*** if :
$\forall\ x \in \mathbf{A_s}$, $y \in \mathbf{A_d}$, $x \nabla y$, and values in $\mathbf{D_x}$ corresponding to values in $\mathbf{D_y}$ are not lost through the path, i.e., not replaced by the *'undefined'* value.

Observation 3.2

Path safety is a ***symmetrical*** property for neighbouring version-collections. That is: if $\mathbf{CL_s}$ and $\mathbf{CL_d}$ are neighbours in the collection-graph, and $\mathbf{P_{s,d}}$ is safe, then $\mathbf{P_{d,s}}$ is safe too.

This symmetry property is *not necessarily transitive* for version-collections that are not immediate neighbours. This is illustrated in Figure 3.3 where three class-versions declare different domain ranges for attribute `Salary`. We can see that paths $P_{1,2}$ and $P_{2,3}$ are safe as no `Salary` value within the domain of the destination class-versions (CV_2 and CV_3 in this case) is taken out of objects mapped from CV_1 and CV_2 respectively. However, objects mapped from CV_1 to CV_3 are bound to lose their `Salary` data if $S <$ £2000 since such values are beyond the domain range of version-collection CL_2. This makes path $P_{1,3}$ unsafe.

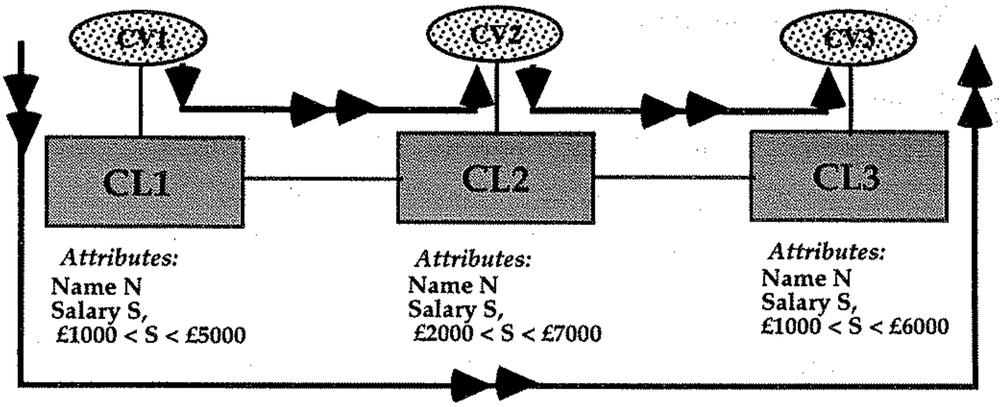


Figure 3.3: An Example of Symmetry Property of Version-Collections
Transformation paths are shown in thick, double arrowhead links

Adopting the generalized-compatibility defined in Section 3.4 would make checking path-safety rather difficult and would consequently complicate the way in which the collection-graph is configured. We choose to restrict the attribute-compatibility of Definition 3.5 as follows.

Definition 3.16

Attribute compatibility presented by the operator ≈ such that $\mathbf{a_i} \approx \mathbf{a_j}$

$$\text{implies that } \alpha\mathbf{D_{ai}} \cap \mathbf{D_{aj}} = \mathbf{D_{aj}} \text{ or } \alpha\mathbf{D_{ai}}$$

$$\text{That is: } \alpha\mathbf{D_{ai}} \subseteq \mathbf{D_{aj}} \text{ or } \alpha\mathbf{D_{ai}} \supseteq \mathbf{D_{aj}}$$

$$\text{and also implies that } \beta\mathbf{D_{aj}} \cap \mathbf{D_{ai}} = \beta\mathbf{D_{aj}} \text{ or } \mathbf{D_{ai}}$$

$$\text{That is: } \beta\mathbf{D_{aj}} \subseteq \mathbf{D_{ai}} \text{ or } \beta\mathbf{D_{aj}} \supseteq \mathbf{D_{ai}}$$

where α maps $\mathbf{a_i}$ to $\mathbf{a_j}$ and β maps $\mathbf{a_j}$ to $\mathbf{a_i}$.

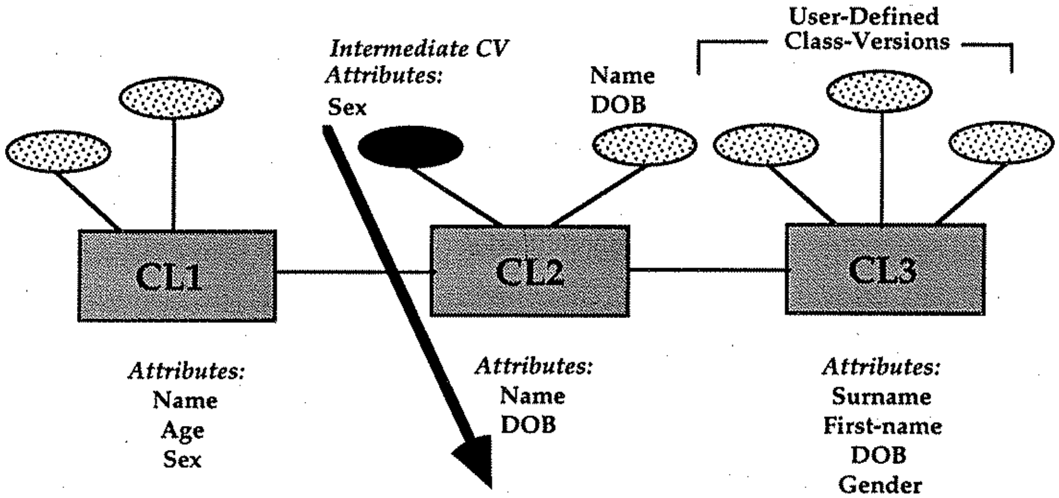
This definition requires that different domains of the same attribute or related attributes to be subsets of each other such that the intersection of any two domains results in either one of them. Adopting that definition permits both fully and partially-compatible attributes, but avoids having domains with mutually exclusive values. Accordingly, the two versions of attribute `Colour` described for Definition 3.5 would not qualify for Definition 3.16, but versions with domains `{White,Red,Blue}` and `{C1,C2,C3,C4}` do qualify. Thus configuring the collection-graph becomes simpler as we shall describe in Section 3.9. Accordingly, we can rewrite Definition 3.15 as follows:

Definition 3.15b

Given $P_{s,d}$ as the transformation path from version-collection CL_s to CL_d, we call $P_{s,d}$ a ***safe path*** if :
$\forall\, x \in A_s, y \in A_d, x \approx y$, and values in D_x corresponding to values in D_y are not lost through the path, i.e., not replaced by the *'undefined'* value.

3.8. Intermediate Class-Versions

The concept of version-collections would work fine as long as no class attributes are dropped in a version-collection and then redefined in another version-collection. If such a situation occurs through the route of object migration or mapping from one class-version to another, then the dropped attribute data will be lost and replaced by default values along the route to the destination version-collection. This renders the path as being unsafe. This is demonstrated in Figure 3.4(a) where attribute `Sex` is missing in CL_2 since it is dropped from all its class-versions. Mapping an object from a class-version in CL_1 to another in CL_3 will require transforming the object to CL_2 format as an intermediate step. Thus, the `Sex` value will be dropped from the object at CL_2 and replaced by the default value of attribute `Gender`. To avoid this situation, an intermediate class-version (**ICV**) can be defined in CL_2 to include the dropped attribute `Sex` of CL_1 (indicated by the dark block in Figure 3.4(a)). Thus, the definition of CL_2 will be entitled to include that attribute and, consequently, it will be possible to define forward and backward transformers relating it to the `Gender` attribute of CL_3.



User-Defined Class-Versions
Intermediate CV Attributes: Sex
Name
DOB
CL1
CL2
CL3
Attributes: Name Age Sex
Attributes: Name DOB
Attributes: Surname First-name DOB Gender

Figure 3.4(a): An Example of Losing Data During Object Mapping

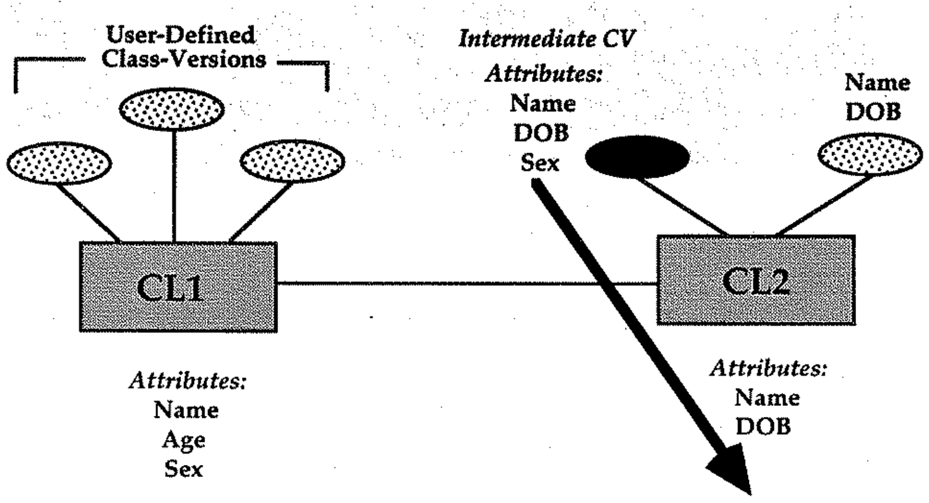


Figure 3.4(b): An Example of Losing Data Due to Object Migration

A more complicated situation may arise when objects are motivated to migrate from their current class-version to a different class-version, for the reasons discussed in Subsection 3.11.1. If the destination class-version has a dropped attribute that is defined in the source class-version, then migrating objects will be bound to lose that attribute's data for the lack of storage to accommodate the attribute at the destination. If the loss of data is to be avoided, then an intermediate class-version may be defined with the appropriate attributes that preserve existing object data, and introduce the additional attributes specified by the application in question. An example of such situation is shown in Figure 3.4(b) where the user-defined class-version in version-collection CL_2 lacks attribute `Sex`. Thus, objects migrating from class-versions of CL_1 to CL_2 would lose their data for that attribute. Defining a new class-version in CL_2 (indicated by the dark block in Figure 3.4(b)) that introduces attribute `Sex` in addition to existing CL_2 attributes would provide a suitable residence for migrating objects. Should migrating objects return to a class-version in CL_1, then old `Sex` values will be maintained in objects that acquired them previously. Making a dynamically created class-version (either application-related or intermediate) or a version-collection operational during the system run-time would require recompiling the overall class handling programs. Depending on the implementation environment, it may be required to link new handlers and transformers to the existing function library, and it may be necessary to update system control tables as well to reflect the collection-graph modification. If this turns out to be a time consuming process, then system operation might be interrupted. It should be noted that the addition of an intermediate class-version will require declaring additional Handlers for class-versions already residing in the same version-collection to account for the new attributes. If those handlers assign a **'nil'** or a default value for those attributes, then the extra effort imposed on the database administrator (DBA) to realize this modification would be trivial, and can be automated in the schema editor.

The intermediate class-versions proposed here are treated in a similar way to that of ordinary class-versions. So, preserving object data requires no separate implementation apart from that of the version-collection approach.

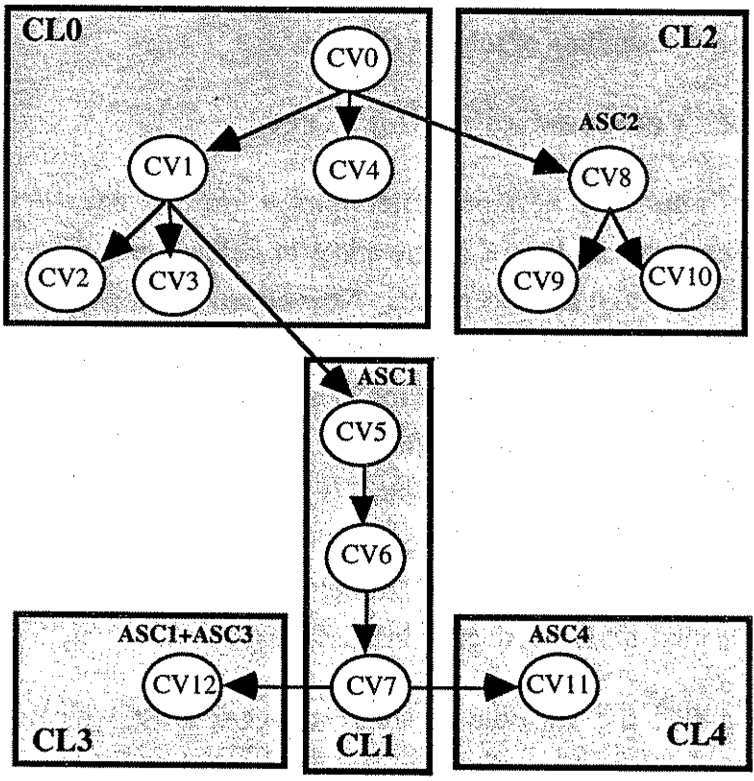
3.9. Configuring the Collection-Graph

In the previous section, we highlighted some situations related to the data preservation requirement that would influence the configuration of the collection-graph. In this section, we put more emphasis on the influence of both attribute compatibility and data preservation aspects on the configuration of the collection-graph. In that respect, we may have two version-collection configurations: the *Collection Tree* and the *Collection Lattice*. In what follows, we shall discuss the basic characteristics and features of each configuration and their impact on the handling of schema changes.

3.9.1. The Version-Collection Tree

In this configuration, version-collections form an acyclic graph where each version-collection would have a single ancestor collection and may have more than one collection affiliated to it as descendants. New version-collections are added to the tree in such a way to provide a safe transformation path between its class-versions and those of the other collections. This configuration can be applied if only compatible attributes are permitted when versioning classes, otherwise safe paths cannot be guaranteed between all versions of a class. The latter situation is illustrated in Figure 3.7(a) where the new class-version CV_3 cannot be accommodated in the collection-graph without compromising the safety of its objects data for either attribute **x** or **y**. That situation is caused by allowing partially-compatible attributes to exist in class-versions.

Assuming that the attribute semantic changes (ASC) introduced by the current class-version are preserved in the new class-version, while the latter introduces additional semantic changes, then a new version-collection will be created for the new class-version and affiliated (connected in the collection-graph) to the version-collection of the modified class-version. Otherwise, the new version-collection can be affiliated to any of the existing version-collections in such a way that simplifies the transformation functions required. We may exemplify both cases with the illustration in Figure 3.5 of a class-version tree depicting the hierarchy of creating those versions. The figure shows how class-versions are grouped in version-collections, depending on the semantic changes they present. Let us consider CV_{12} that was created from CV_7 and maintained the semantic changes ASC_1 of the latter alongside its own ASC_3. Because of the changes ASC_3, CV_{12} had to be separated in a new version-collection CL_3. Also, modifying ASC_1 to ASC_4 separates CV_{11} in a new version-collection CL_4. However, configuring the collection-graph of this class treats CL_3 and CL_4 in two different ways. This is shown in Figure 3.6 which depicts the collection-graph. According to the previous argument, we observe that CL_3 is affiliated to CL_1 since the latter presents the closest changes to those of the former, while the other version-collections introduce different semantic changes that would complicate transformation functions to and from CL_3. On the other hand, CL_4 is connected to CL_0 in the collection-graph based on the hypothesis that ASC_4 is simpler to present in terms of transformation functions to and from CL_0 than other version-collections.



ASC = Attribute Semantic Change

Figure 3.5: Class-Version Hierarchy

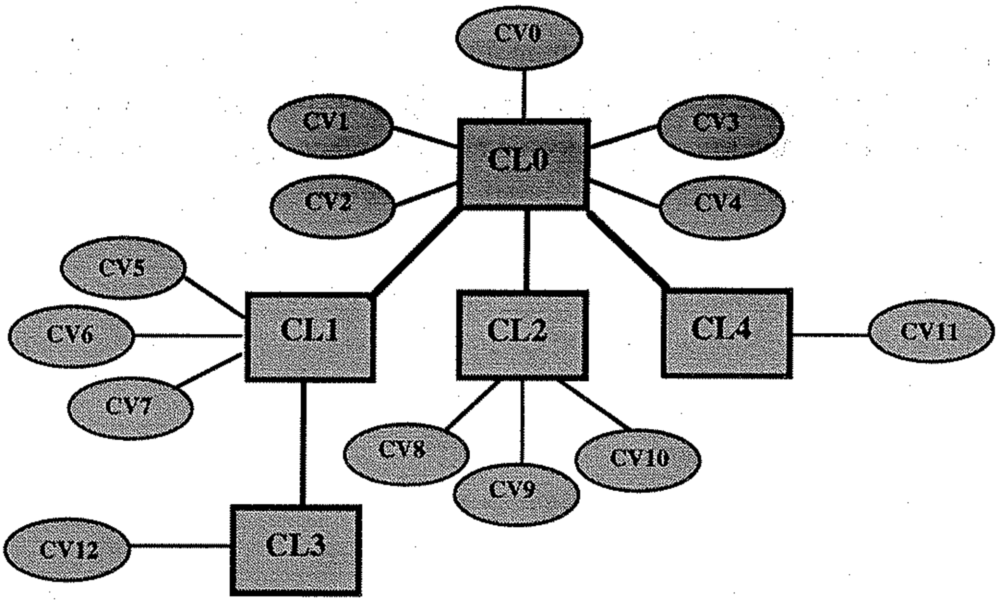


Figure 3.6: Tree Version-Collection Graph

Constructing the collection-graph in this way can be beneficial for the following reasons:

- It implies that a version-collection other than the root will have only one group of backward transformers to its sole ancestor, and one group of forward transformers set to each of its immediate descendant version-collections. On the other hand, the root version-collection (that is CL_0) requires only Forward transformers while leaf version-collections require only Backward transformers.

- There would be a single transformation path connecting any two class-versions in the collection-graph, and that path is guaranteed to be safe.

- Since changing the attribute semantics tends to complicate the Forward/Backward transformers, the flexibility of arranging version-collections as described, instead of restricting their affiliation to be direct to CL_0 or to their immediate ancestor version-collection, can help reduce those changes to simple straightforward mappings of attributes.

- It shortens the transformation path between class-versions, compared to the traversal of class-version hierarchy. For example, accessing objects of CV_{12} from CV_{10} (see Figure 3.6) requires traversing the path CL_3-CL_1-CL_0-CL_2 and counts for 5 transformation stages. If we traverse the path connecting both versions in the class-version hierarchy (shown in Figure 3.5), 7 transformation stages will be required. In some cases, a class-version may become more distant from its origin version, but at the same time become closer to other versions in the collection-graph. For example CV_{11} was 2 transformation stages away from CV_{12} in the class-version hierarchy, but became 5 stages away in the collection-graph. At the same time, CV_{11} became 3 transformation stages away from CV_0 instead of 5 stages in the class-version hierarchy. This situation is due to the flexibility of relocating its version-collection.

3.9.2. The Version-Collection Lattice

This configuration is introduced to support partially-compatible attributes in class-versions. As we indicated in the previous subsection, such attributes may prevent fitting new version-collections in the collection-graph for failing to provide safe paths. With the lattice configuration, resolving such situation is possible. We illustrate this by the examples of Figure 3.7. In those examples, we have a group of class-versions and version-collections that present a variety of domain combinations for two attributes **x** and **y**. The relationships between those domains accompany the illustration.

In Figure 3.7(a), we introduce new entities: CV_3 and CL_3. Connecting CL_3 to only one of the existing version-collections (e.g. CL_0 ,CL_1 or CL_2) would not provide safe paths for attributes **x** and **y** between any two version-collections. For example, the path [CL_3 ,CL_0, CL_1] is not safe for $\mathbf{x_2}$ as $\mathbf{x_1}$ of CL_0 has a narrower domain than that of $\mathbf{x_2}$. This causes the loss of $\mathbf{x_2}$ values outside that domain, and deprives CL_1 from getting those values.

In Figure 3.7(b), we reconfigure the collection-graph by connecting CL_3 to two version-collections: CL_0, CL_1, and removing the existing edge between CL_0 and CL_1 that becomes redundant. Now, the path connecting any two version-collections would deliver attribute values of the source collection that are covered by the domain of the destination collection.

Figure 3.7(c) presents another example of reconfiguration where the new entities: CV_4 and CL_4 combine $\mathbf{x_3}$ and $\mathbf{y_3}$ and lead to the configuration in Figure 3.7(d). We notice that CL_4 is connected to CL_2, CL_3, and the existing edge between CL_1 and CL_2 became redundant.

Introducing a new version-collection may have more than one connection alternatives in the collection-graph. This is illustrated in Figure 3.7(d) where the new version-collection

CL_5 may be affiliated to either CL_0 ,CL_1 ,CL_3 , or CL_4 and still achieve safe paths for attributes **x** and **y**. This situation emerges from the fact that CL_5 provides domains for both attributes that are narrower than those defined by the aforementioned collection-versions. In this case, only one collection is selected to connect the new collection in the graph and avoid cycles.

Having safe-path cycles in the collection-graph can prevent breaking existing paths and would simplify configuring the graph. However, having alternative safe-paths would impose more processing overhead to choose the optimum path connecting any two class-versions.

Although this type of adaptable configuration ensures a safe transformation path between partially-compatible attributes in different class-versions, it requires modifying existing transformation functions to support the changes in transformation paths. A broken path between two neighbouring version-collections would require removing their relevant forward and backward transformers, while a newly-established path would require defining new forward and backward transformers. In practical terms, this process may not prove to be feasible in real-time applications if frequent class-versioning affects attribute semantics. From the foregoing discussion, we can formalize a procedure for configuring the collection-graph as follows:

- Connect the new $\mathbf{CL_j}$ to an existing $\mathbf{CL_i}$ in the collection-graph in such a way that: $\forall\ \mathbf{x} \in \mathbf{A_i}\ , \mathbf{y} \in \mathbf{A_j}\ , \mathbf{A_i} \in \mathbf{CL_i}\ , \mathbf{A_j} \in \mathbf{CL_j}\ , \mathbf{x} \approx \mathbf{y}$, we should have some function θ that maps **x** to **y** such that $\theta \mathbf{D_x} \subseteq \mathbf{D_y}$.

- Otherwise, reconfigure the collection-graph and avoid cycles such that any transformation path between any two collections $\mathbf{CL_i}$, $\mathbf{CL_k}$ passing through the new $\mathbf{CL_j}$ would be safe. That is: $\forall\ \mathbf{x} \in \mathbf{A_i}\ , \mathbf{y} \in \mathbf{A_j}\ , \mathbf{z} \in \mathbf{A_k}\ , \mathbf{A_i} \in \mathbf{CL_i}\ , \mathbf{A_j} \in \mathbf{CL_j}\ , \mathbf{A_k} \in \mathbf{CL_k}\ , \mathbf{x} \approx \mathbf{y} \approx \mathbf{z}$, we should have: $\theta \mathbf{D_x} \subseteq \mathbf{D_y}\ , \psi \mathbf{D_y} \subseteq \mathbf{D_z}$ **or** $\theta \mathbf{D_x} \supseteq \mathbf{D_y}\ , \psi \mathbf{D_y} \supseteq \mathbf{D_z}$ where θ maps **x** to **y**, and ψ maps **y** to **z** .

In some situations, affiliating new version-collections to more than one of the existing collections in the collection-graph may be required to provide safe transformation paths. This would lead to lattice configurations while re-configuring the collection-graph may be avoided. In such configuration, having more than one path linking a collection to other version-collections would imply that there is at least one safe path passing though it. An example of such situation is shown in Figure 3.8(a). In this example, we find that CL_4 cannot be fitted in the graph without compromising its $\mathbf{z_3}$ attribute values. One way to solve this problem is to affiliate CL$_4$ to both CL$_1$ and CL$_2$. It can be seen that path [CL$_0$, CL$_1$, CL$_4$] is safe but [CL$_0$, CL$_1$, CL$_2$, CL$_4$] is not since attribute y of CL$_0$ and CL$_1$ may lose its value when transformed through CL$_2$.

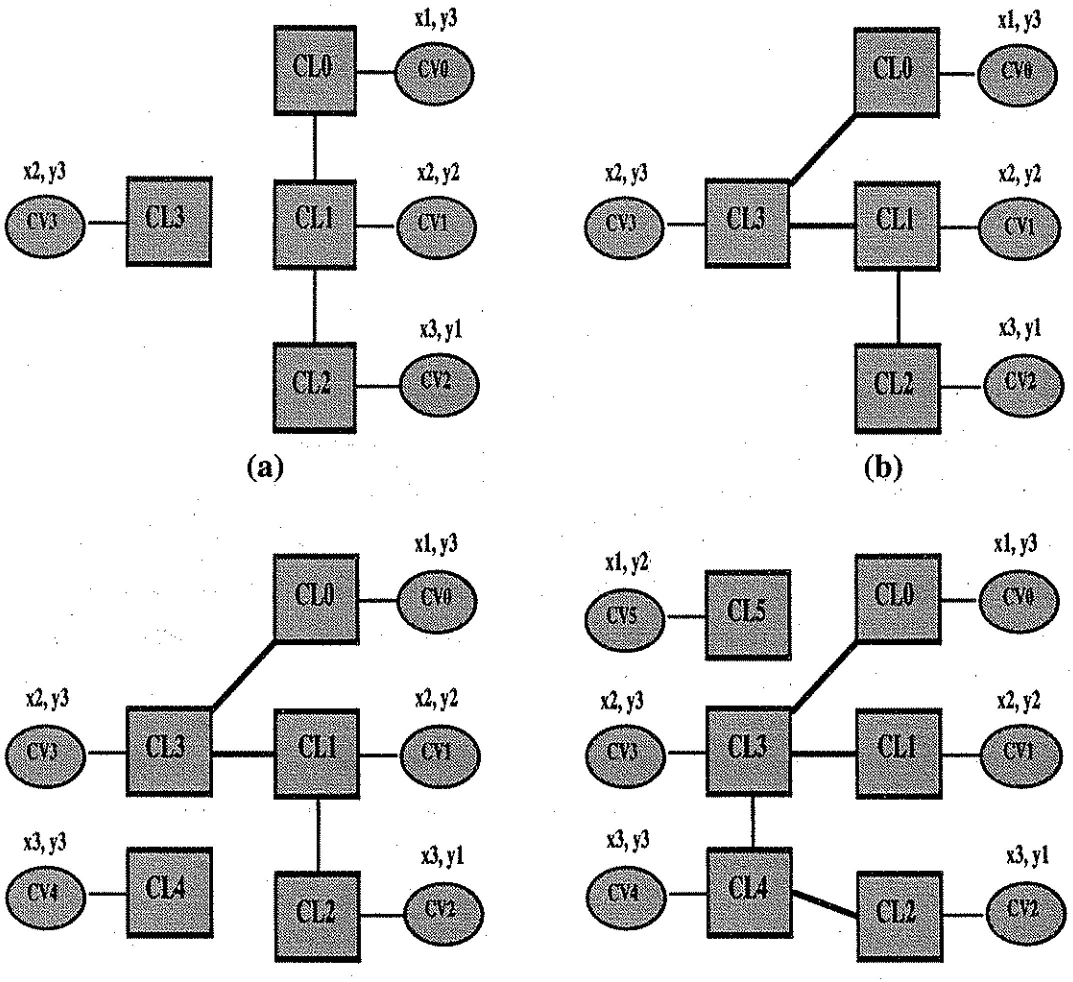


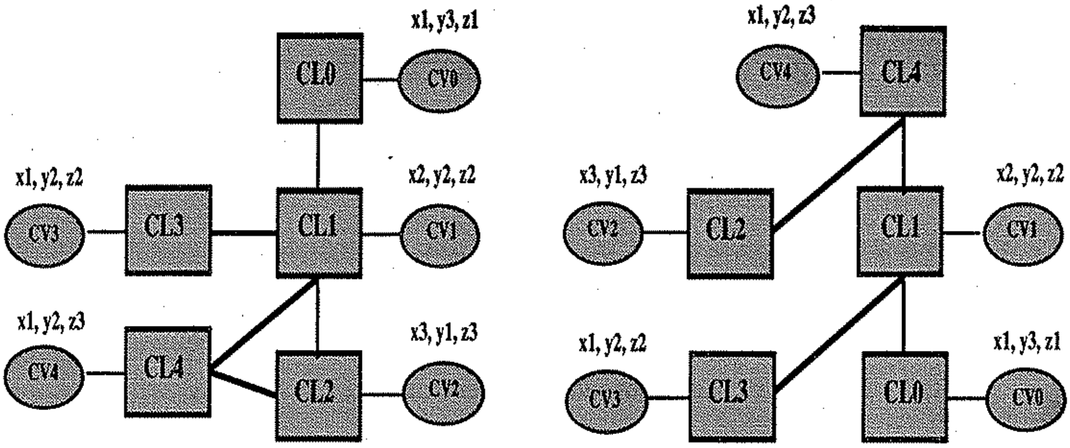
Figure 3.7: Lattice Version-Collection Graph (1)

$\theta Dx_1 \subseteq Dx_2$, $\psi Dx_2 \subseteq Dx_3$, $\omega Dy_1 \subseteq Dy_2$, $\zeta Dy_2 \subseteq Dy_3$
where θ maps x1 to x2, ψ :maps x2 to x3, ω :maps y1 to y2, ζ maps y2 to y3

Viewing the lattice configuration with the destination version-collection at its root, and considering safe paths only to and from that collection, we get a tree-configuration out of the collection-graph. This is illustrated in Figure 3.8(b) where CL$_4$ is set as the root collection.

A variation of the configuration of Figure 3.7(d) is illustrated in Figure 3.9. It can be seen that path [CL$_3$, CL$_1$, CL$_2$] is safe but [CL$_3$, CL$_0$, CL$_1$, CL$_2$] is not since attribute x$_2$ of CL$_3$ may lose its value when transformed through CL$_0$ to CL$_1$. Also, the path

[CL_0 , CL_3 , CL_4 , GL_2] is safe but [CL_0 , CL_1 , CL_2 , CL_4] is not since attribute y_2 of CL_1 may lose its value when transformed through CL_2 to CL_4. CL_5 can still be connected to either CL_0 , CL_1 , CL_3 , or CL_4 as in Figure 3.7(d).



(a) (b)

Figure 3.8: Lattice Version-Collection Graph (2)
$\theta D_{x1} \subseteq D_{x2}$, $\psi D_{x2} \subseteq D_{x3}$, $\omega D_{y1} \subseteq D_{y2}$, $\zeta D_{y2} \subseteq D_{y3}$, $\alpha D_{z1} \subseteq D_{z2}$, $\beta D_{z2} \subseteq D_{z3}$ where θ maps x1 to x2, ψ :maps x2 to x3, ω :maps y1 to y2, ζ maps y2 to y3, α maps z1 to z2, β :maps z2 to z3.

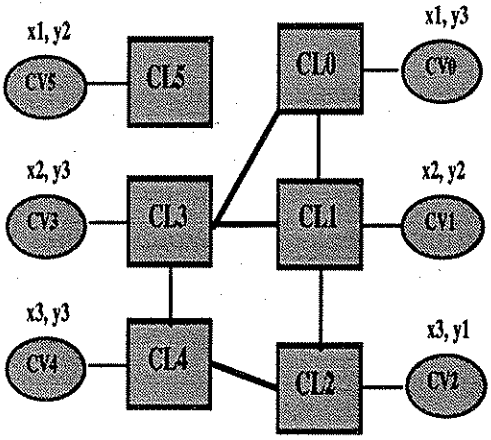


Figure 3.9: A Variation of the Collection-Graph in Figure 3.7(d)

3.9.3. The Tree Configuration VS. The Lattice

- The tree configuration supports compatible attributes only, while the lattice supports both compatible and partially-compatible attributes. This makes the latter appropriate for a wider range of applications than the former. Being restricted to the tree configuration means that the database management system can only support compatible attributes, and this limits the changes or modifications allowed for attribute semantics.

- Adopting the lattice configuration requires keeping track of safe paths between any two version-collections in the lattice as version-collections are being added, or removed from the lattice so as to help the user avoid using unsafe alternate paths. This would make structuring the collection-graph more difficult. The tree configuration, on the other hand, provides a unique, and safe path between any two version-collections at all times. So keeping an updated complex control information about the collection-graph is not required and structuring the collection-graph is more or less straight forward.

- Since the tree structure is not re-configurable, the existing transformation functions defined in each version-collections need not be changed as the tree grows or shrinks. Removing non-leaf version-collection, if permitted, would require reconnecting its affiliated collections to an alternative collection and consequently requires modifying their existing transformation functions.

3.10. Examples of Schema Evolution

In this section, we introduce four examples illustrating some changes to the schema and how the version-collection concept deals with them.

Example 1:

In this example, CV_2 adds a derivable attribute `Power` (P) to the version-collection, while CV_3 replaces attribute `Current` (I) with `RatedCurrent` (RC) and changes the class name to `NewMachine`. Also, CV_4 drops V and RC attributes of CV_3. It is clear that all those changes to the class definition belong to *Group-I* , thus permit the creation of the new class-versions (CV_2, CV_3, CV_4) in the same version-collection (CL_1). As depicted in Figure 3.10, handlers are attached to each class-version to map the missing attributes in its definition to those of the version-collection.

Example 2:

This example (see Figure 3.11) presents three versions of a schema that access the same database. Each schema-version introduces some changes to the inheritance hierarchy of subclasses of class `Machine`. In the example, we focus on versioning class `Drill` due to the modification of its inherited attributes. We can see that class `FastMachine` has been removed from the class lattice of schema-version SV_1 causing the creation of schema-version SV_2 and a new class-version CV_2 of class `Drill`. The latter class-version does not inherit attribute `Speed` any more. So, CV_2 is provided with a handler returning a 'nil' for that attribute. Also, a new class `HandTool` has been added to the class lattice. So, schema-version SV_3 and a new version of class `Drill`, CV_3 inheriting the attribute

`Weight`, have been created. CV_3 has a handler for attribute `Speed`, while the previous two versions of the same class are provided with new handlers for the newly-introduced attribute `Weight` in the version-collection. We would like to note that existing class-versions of class `Drill` can be used in other schema-versions without the need to change them if the modification affects subclasses of `Drill`, and not its superclasses or the class itself. So, we may come to a general conclusion that version-collections do not depict a relationship between schemas in terms of the class-versions those schemas use. However, version-collections only relate versions of the same class to each other and this relationship is in terms of the attributes they present.

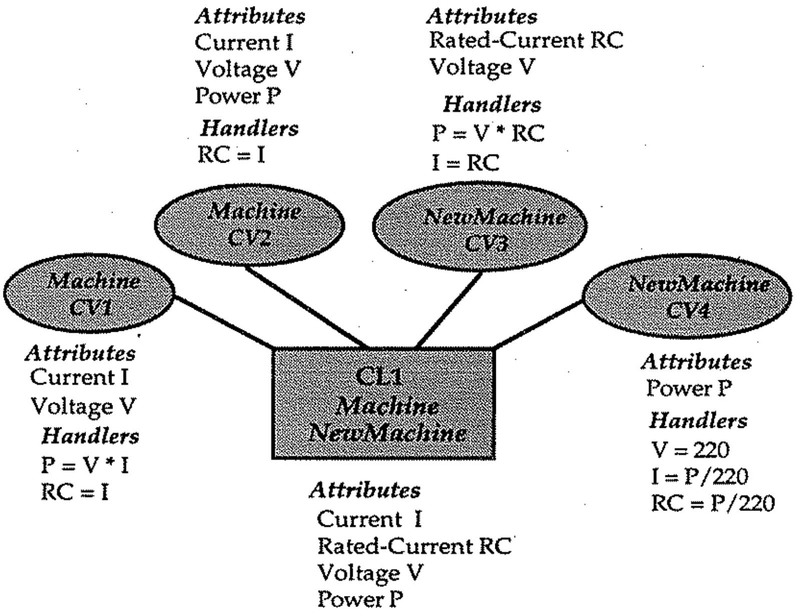


Figure 3.10: An Example of Group-I Changes (1)

Example 3:

In this example (see Figure 3.12), three changes have been made to the definition of class `Employee`, namely: the default value of attribute `Sex`, the units of attribute `Salary`, and different semantics of attribute `Age`. As the three changes belong to *Group-II* , the new class-version had to be separated in a new version-collection CL_2 which can be mapped to the previous one (CL_1) using the Backward transformers of CL_2. Version-collection CL_1 can be mapped to CL_2 using the Forward transformers of CL_1.

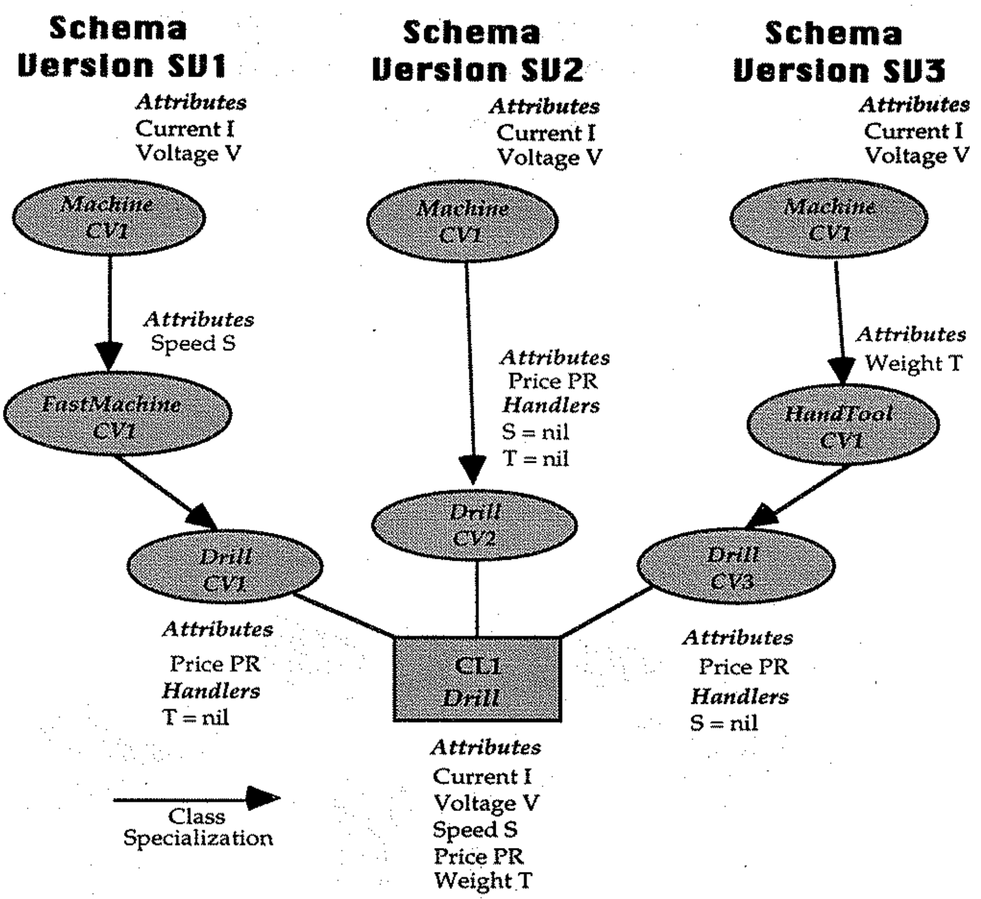


Figure 3.11: An Example of Group-I Changes (2)

Example 4:

In this example, we illustrate the handling of complex objects where versions of their complex class specify different versions for the constituent classes. Figure 3.13 depicts a sub-schema composed of class `Employee` that has `Person` as its superclass, and defines two constituent classes: `Spouse` and `Children`. The attributes defined by each class are shown between brackets. The class-versions shown in the collection-graph introduce some changes to the declarations of those attributes, and are configured accordingly. We notice that CV_1 of `Employee` is related to CV_4 of `Spouse` and CV_5 of `Children` while CV_2 of `Employee` is related to CV_6 of `Spouse` and CV_7 of `Children`. So, mapping an employee's object from CV_1 to CV_2 requires subsequent mapping of his spouse's object from CV_4 to CV_6 and his children's objects from CV_5 to CV_7. Versions CV_1 and CV_2 of class `Employee` are shown connected to their constituent class-versions using arc links. We notice that mapping the parent and its constituent objects is done separately for each object in its related collection-graph. The transformation path for each object may also incur different number of nodes. Mapped objects are then joined at the destination class-versions and delivered to the user.

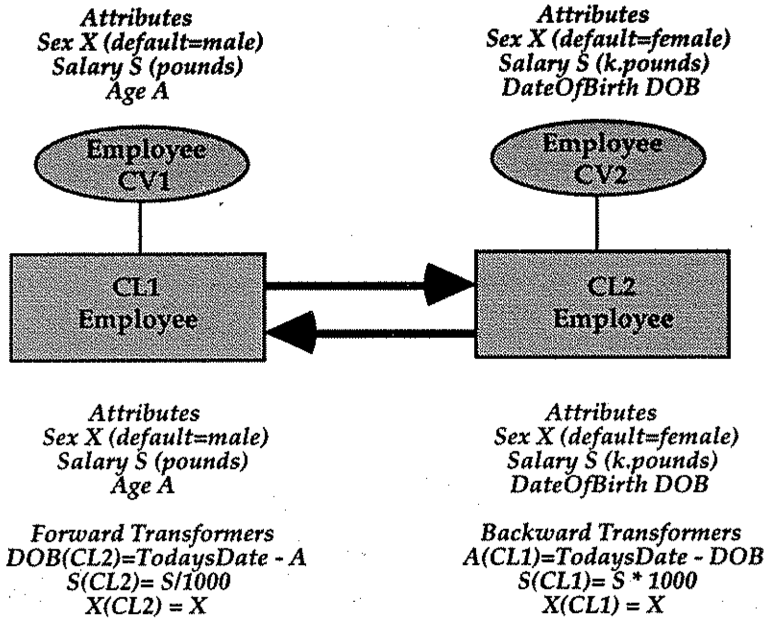


Figure 3.12: An Example of Group-II Changes

3.11. Object Migration Between Class-Versions

Object migration has been considered in several object-oriented and temporal DBMS for a variety of motivations and was supported by different schemes and frameworks. In the real world which the databases are set to model, objects may *migrate* from classes to classes dynamically in many practical situations. For example, a person who was single but later on got married (hence migrates from class `Single` to class `Married`) and who was recently promoted from a programmer to a project manger (hence he migrates from class `Programmer` to class `Manager`). Also, objects can be *transmitted* to more than one class at the same time. For example, a student becomes an engineer and at the same time a lecturer after he graduates. Objects can also be *propagated* to several other classes while still maintaining their membership in current classes. For example, a person who becomes an American citizen while he still maintains his Australian citizenship (hence he is propagated to be a member of class `American` in addition to the current `Australian` membership). Li proposed a framework to support such changes in object membership for the Ontos DBMS [Li 94] . Another cause of object migration can be updating object's attribute values or adding or dropping attributes in class definition that in effect changes the object's position in the class lattice. The Object Data Management Group (ODMG) recognizes a similar situation in its standard of the object model. The ODMG [ODMD-93] permits objects to dynamically acquire a type (i.e., a class), to dynamically lose a type, and to dynamically change representation. An example of such dynamics is when an object becomes an instance of a more immediate subtype of a type of which it is already an instance.

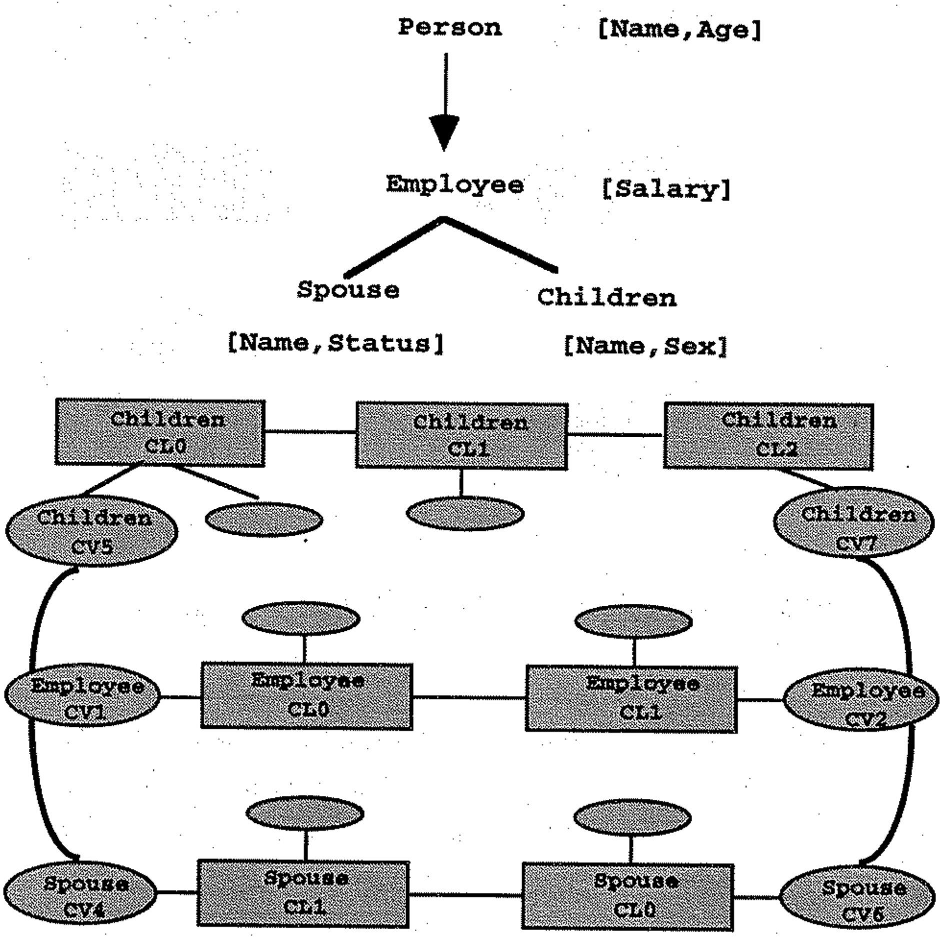


Figure 3.13: Complex Object Manipulation

Adding the time dimension to objects may also require keeping versions of the object's history distributed among several classes. El-Sharkawi [El-Sharkawi 90] presented an approach for handling such situations.

Within the context of the class-versioning approach, we consider a different view of object migration from those mentioned above. Object migration in our approach has close resemblance to that proposed by Clamen [Clamen 92, 94]. Clamen considers migrating parts (or fragments) of the object, which are called facets, from one database to another to improve locality between the object and the objects it is related to (typically via reference pointers). Clamen also considers *facet migration* whereby facets might relocate to sites where they are referenced. Though we do not split objects into sub-components, we perceive that some of the motivations considered by Clamen would require an object to migrate from its creator class-version to reside in a different class-version in the Collection-Graph. In this section, we shall describe those situations and state the constraints imposed in our approach to control their occurrence.

3.11.1. Motivations for Objects to Migrate

We may expect some situations where an object needs to have the mobility to move as a whole between class-versions, instead of being stationary in its creator class-version during its lifetime. There are three situations in which an object migrates to other class-versions. These are:

(1) ***Object Updates***: A user dealing with a particular class-version may attempt to update an instance of another class-version. We recognize two types of updates:

Type-A: targets an attribute that is not declared in the object's class-version and is managed instead by a handler. In this case, extra storage will be needed to store the new assigned attribute value and the object will have to migrate to a different class-version that provides the extended storage and preserves other object attributes.

Type-B: attempts to change the value of a declared object attribute but beyond the attribute's domain defined in the object's class-version. Committing such updates will disqualify the object to reside in its class-version though storage space is already provided for the updated attribute. So, the object will have to migrate to a class-version that defines the appropriate domain for that attribute.

(2) ***Dropping Class-version***: Considering the case where a class-version may no longer be required by any user (that is, it is virtually dropped), the database administrator (DBA) should decide whether its instances are to migrate to another class-version and specify that destination, or whether they are to be deleted altogether.

(3) ***Frequent Object Access***: Here, we consider the case where objects are frequently accessed by a class-version other than their creator class-version. In this case, it would be advantageous, from system performance point of view, to

migrate such objects to the more active class-version so as to reduce transformation workload accompanying their manipulation.

3.11.2. Migration Safeguards and Limitations

In this subsection, we discuss migration-related situations that might affect preserving object data or transaction consistency. We mentioned earlier in Section 3.8 the effect of dropping class attributes through the transformation path and provided the intermediate class-versions to avoid losing attribute's data during migration.

Considering Type-B updates, on the other hand, such operation has the potential of rendering the updated object inaccessible by its own class-version due to the modification of attribute constraints. We assume here that the creator class-version is still active (that is, it remains accessible by some users). So, such operation should be restricted to users with proper update privileges.

Since objects are allowed to migrate in our approach, according to the previously mentioned rules, there might be a situation when a user processes an object at some class-version and later on encounters the same object at a different class-version within a single LUW[5]. This can occur in a multi-user environment when a user updates an object already processed by a second user, and causes the object to migrate to another class-version whose objects are to be manipulated by the second. To avoid this type of inconsistent access to objects, we append to each migrating object a collection of log records specifying the migration routes of the object and the associated time stamp of each migration. So, when the user attempts to manipulate such objects, which are only expected to be encountered in intermediate class-versions as we explained earlier, he can check whether he has previously encountered the object or not. This procedure is illustrated in Figure 3.14. At time t_3, user-A accesses Page$_j$ of CV$_y$ where he encounters object O which he had already read at time t_1 but in Page$_i$ of CV$_x$. Checking the object's migration log (we assume that CV$_y$ is an intermediate class-version) reveals that the object migrated for user-B from Page$_i$ of CV$_x$ which he has processed at time t_1. Since t_1 is smaller than t_2, so he concludes that he has already processed that object and bypasses it. User-C migrates the same object again but after user-A processes it at CV$_y$, so another record is inserted in the object's log to reflect this action. In this example, we assume that the user locks the whole page during processing its objects, so both users B and C lock their source and destination pages simultaneously during their LUW.

[5] Logical Unit of Work.

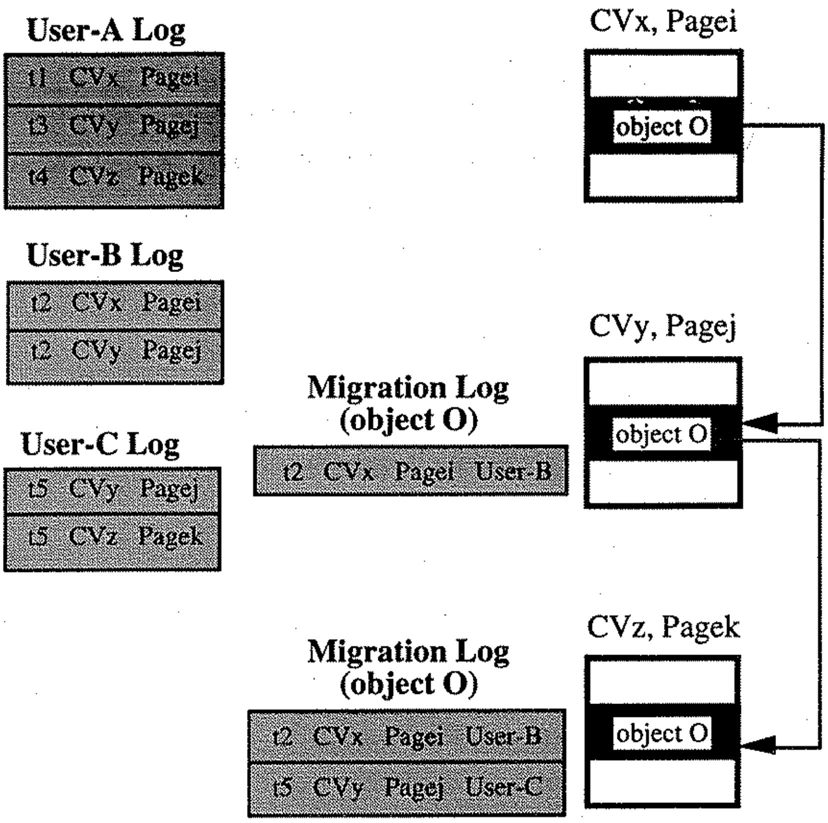


Figure 3.14: Handling Object Migration

A situation with similar consequences is recognized in IBM's SQL/DS relational DBMS, but not due to object migration. In this case, locking is responsible for data inconsistency when the isolation level is *cursor stability* [IBM 87]. In such situation, queries within a single LUW are subject to the following data inconsistencies :

(1) If the user LUW reads data twice it can get different results (another user can meanwhile modify the data and COMMIT the modification).

(2) Since other concurrent users can read and modify rows which have been read by the user LUW (but are not CURRENT OF CURSOR), a modification based upon a prior read can be incorrect.

(3) If the user LUW is traversing a table via an index, it can find the same row (or rows) twice due to a concurrent update. This can occur because, after reading the row the first time, another user can update the index column value (and COMMIT) so that the user LUW now encounters the row again (with its updated index column value).

(4) If the user LUW is traversing a table via an index, it can fail to find a row (or rows) at all (even though it meets the user criteria) due to a concurrent update. This can occur because while the user LUW is reading, another concurrent user modifies the row (and COMMITs) so that its index column value causes the row to be behind (earlier than) the row the user LUW is currently reading.

3.12. Supporting Schema Versioning

Schema versioning is supported within the framework of the class-versioning approach where a database schema specifies which versions of its class are being used. Introducing any of the changes mentioned in Section 3.3 to the schema results in the creation of a new version of the database schema being required. In that case, the user needs to select one of the existing schema-versions as the basis for creating his own. It is not necessary for a new version of the schema to introduce a new class-version for each class in its hierarchy. That is, different schema versions may contain the same versions of some of their classes, while introducing different versions of the other classes.

Choosing a schema-version implies which classes are available for the user to modify and which class-version definitions he may edit. Should the user require the addition of a class which has versions included in other schema-versions, then he may refer to those schema-versions to choose the appropriate class-version as the basis to include in his own schema. The user may then modify the definition of the class-version to his requirements. Upon building the class-hierarchy of the new schema-version, the user may wish to add to a class-version an attribute that exists in other schema-versions. This may be achieved by locating a version-collection of that class where an appropriate definition of the attribute exists, and then copying that definition to his class-version.

Modifying a schema-version may involve some propagational changes to classes across the inheritance hierarchy. The schema editor should manage such operation and ensure that the new schema-version is consistent.

When accessing the database, it is sufficient for the user to specify his version of the schema, and this would imply which class-version he is querying. The DBMS is expected to allow the user to explicitly specify class-versions, other than those declared in his version of the schema, to be involved in his queries provided that those classes are taking part in his schema-version.

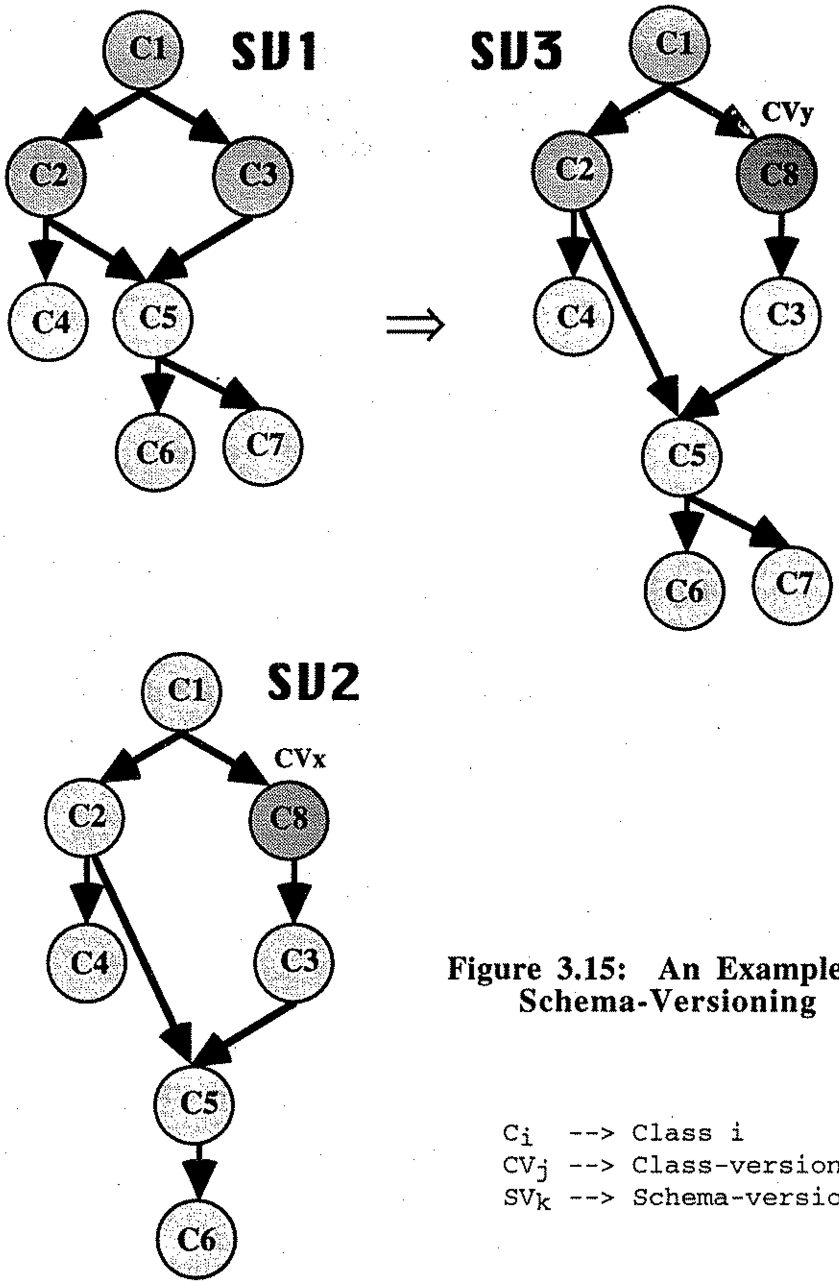


Figure 3.15: An Example of Schema-Versioning

```
Ci  --> Class i
CVj --> Class-version j
SVk --> Schema-version k
```

An example of schema versioning is illustrated in Figure 3.15 where schema-version SV_3 is to be created from SV_1. However, the new schema-version requires the addition of class C_8 which does not exist in SV_1. So, the user selects one version of that class from its collection-graph, and he decides on CV_x of SV_2 that matches closely his requirements of that class. The user then changes CV_x definition which causes the creation of class-version CV_y. The changes made to SV_1 should be propagated, as they occur, to its class-definitions and when the final version of the schema is to be committed, each class-definition is compared to that of its originating class-version and the new class-version, if any, is placed in the appropriate version-collection in the corresponding collection-graph.

3.13. Summary

In this chapter, we introduced the concept of version-collections to persistent object management which enables OODBMS to support both schema and class-versioning. The proposed approach is based on grouping the versions of a class in different version-collections, for the same class, depending on the nature of the schema changes they introduce and not on the order of their creation. Objects are mapped from any class-version to another across the version-collections using special transformation functions attached to each class-version (which we called Handlers) and version-collection (which we called Forward and Backward Transformers). The novelty of the proposed approach lies in its combined features that are partially supported by existing systems. Herein, we summarize those features:

- It supports hierarchical versioning of classes. So, class-versioning is not restricted to the latest class-version as in the CLOSQL linear versioning approach.

- In addition to supporting the standard schema changes available in many OODBMS, it also supports semantic changes to attribute definition, restricted in Kesim, the AVANCE and ENCORE systems. This widens the spectrum of supported changes to the database schema.

- It provides both forward and backward compatibility between class-versions, as opposed to the OTGen, the ORION, and the GemStone systems which support only forward compatibility.

- It shortens the object-transformation path between any two class-versions in the database, in contrast with the case of extending the linear versioning approach or traversing the nodes of the class-versions tree.

- It eliminates the chance of losing object data or rendering objects inaccessible in the course of querying the database or mapping objects to different class-versions. This is achieved by introducing the novel concept of intermediate class-versions. This makes our approach a unified mechanism that preserves object data as schema evolves, in contrast with existing systems such as the CLOSQL whose preservation is separate from the conceptual approach.

- It reduces the number of transformation functions required for any class-version, since only one group of handlers is defined for the class-version to map its objects to its version-collection. Traversing the class-version hierarchy, on the other hand, would require collections of transformation functions for each class-version to map its objects to *each* of its immediate succeeding class-versions as well as its preceding class-version.

We classified supported schema changes into two groups, based on their semantics. We emphasized the possible configurations of version-collections of a class as being Tree and Lattice, and highlighted the applicability of each configuration. We also discussed how schema-versioning is supported in the context of the class-versioning approach. Examples of schema evolution were given to demonstrate how the approach is applied to OODBMS.

Future work would be needed to determine the practicality of implementing this approach in commercial OODBMSs. For application domains featuring frequent Group-II changes to the database schema, configuring the collection-graphs and its consequences may cause interruption to on-line systems and may require careful handling in updating system tables. Comparing the effort of defining the handlers or the Forward/Backward transformers, with that of modifying existing programs and data to match the new schema, we expect that the result will be in favour the former option in large database environments.

This page is intentionally left blank

Chapter 4
Implementation Aspects

In this chapter, we present different aspects related to object representation and manipulation within the context of the class-versioning approach. We start with a section revisiting the existing physical storage models adopted for persistent objects. This is followed by sections two and three that describe how objects can be retrieved and updated from different class-versions. In section four, we present different methods for querying the database, then we go through query optimization issues in section five. In section six, we discuss query manipulation in the C++ programming language, and present a seamless approach for extending language semantics to handle persistent objects. We end the chapter with conclusions reached from the foregoing discussion.

4.1. Physical Storage Model Issues

4.1.1. Object Storage Models Revisited

We reviewed in Chapter 2 studies of the different physical storage models for simple and complex-objects. Here, we discuss how schema evolution is supported by those models assuming schema modification, not versioning, is applied. In PSM-1, the creation of a new class can cause some major changes in the storage. If it is a subclass of an existing class, then some existing instances may migrate to this new class. The deletion of a class not only removes the instance variables particular to that class, but also the inherited variables for its subclasses. Addition of an instance variable will require more storage space for this class and all its subclasses. The reverse is true for deleting an instance variable. In PSM-2, if a new class is added, and instances for that class inserted, the new data will propagate up the hierarchy. When a class is to be deleted, all those instances must be deleted in that class all the way up the hierarchy. If a non-leaf node is removed, then changes are also propagated down the hierarchy. If a new instance variable is added to a class, then that class and all its subclasses must account for this change. Deletion of an instance variable is similar to that in PSM-1, except that more occurrences of the same data values will have to be removed. In PSM-3, a new class will have the instance `variable-id` along with any new instance variables particular to the new class. No changes will be made to any existing classes except for linking the new class to the DAG. Deletion of a class still requires deletion of all instances of that class. Changes in instance variables remain local to the class changed. In PSM-4, a new column is added to the relation for each new variable. The additional storage space is proportional to the size of the entire database. If a class is deleted, all instance variables particular to it must be removed. This means the removal of entire columns in the relation. PSM-5 is a relaxation of the restrictions on PSM-1. So, schema evolution when it occurs will behave in much the same manner. In PSM-6, creating a new class means that a `class-id` is assigned. Deletion of either a class or instance variable means deleting those value triples, and associated string pairs, for the instances. Adding a instance variable involves selecting an identifier for the variable, then add a new value triple for every instance of the class.

We treat complex-objects representation in the class hierarchy as similar to simple objects with subobjects representing attribute values in their parent objects. So, modifying a constituent class definition is in effect a modification to its corresponding attribute in the complex class and would affect other classes in the class hierarchy according to the six models discussed earlier.

4.1.2. Storage Models Applicability

Here, we discuss the applicability of simple and complex-object storage models to the class-versioning approach. For the PSM-2 model, the object migration and the compatibility constraints mentioned in Chapter 3 makes it imperative for all occurrences of a migrating object to move together to their corresponding class-versions in the destination schema, or be deleted. This emerges from the possibility of updating any of the object occurrences in the old schema in such a way that renders it incompatible with the migrated instance. This can be illustrated with the example shown in Figure 4.1, where updating an `Engineer`'s `Salary` to £8000 takes place in schema-version SV_2 which expands beyond the `Salary` range of his migrated `Employee` in schema-version SV_1. In that case, the `Person` and `Engineer` occurrences in SV_2 related to the `Employee` become incompatible with his occurrence in SV_1. So, removing the `Engineer`'s occurrences in either schema would be necessary, or otherwise the update should be rejected. The former option means the loss of `Speciality` data if the `Engineer` occurrence is removed, while the latter option does not benefit from object migration support of the class-versioning approach to achieve such update. We consider such physical model unsuitable to support the class-versioning approach for such anomalies and constraint checks it imposes. The PSM-3 model simplifies modifying the schema as changes made to a single class need not be propagated throughout the inheritance hierarchy. However, processing an instance of a subclass in a particular schema-version would require joining all instance occurrences in superclasses according to the class-versions declared in that schema-version. Therefore, PSM-3 may be regarded as vertically partitioning object data and this poses special requirements within the context of the class-versioning approach. Considering the case when an attribute of the destination class-version is derived from more than one attribute of the source class-version, then all attributes involved in the derivation must be considered as implicitly requested by the user and should undergo the necessary transformations. The PSM-4 model removes one of the class-versioning approach advantages. That is the optimization of used storage by providing handlers for undefined attributes in class-versions. The PSM-5 model requires enforcing full compatibility of attributes between classes that are not in the same inheritance hierarchy branch. This emerges from the possibility of modifying an attribute constraint in one class of the schema in such a way that disqualifies the attribute's value in another class having an equivalent attribute. Enforcing full attribute compatibility in this model would lead to consequential, unintended and propagational versioning of classes throughout the schema, and would complicate the configurations of version-collections. The PSM-6 model requires class-version instances to be assembled from their attribute triples format associated with each class-version. It would be also subjected to the partitioning requirement enforced on PSM-3.

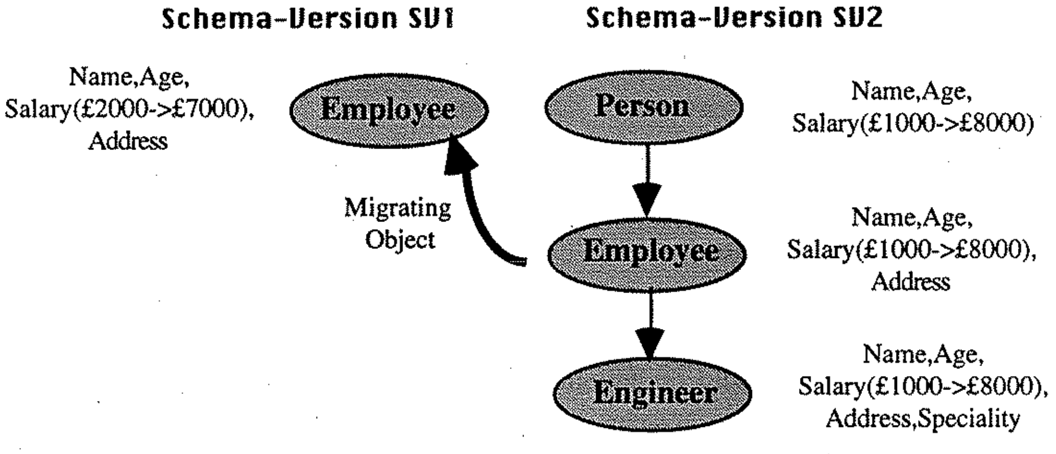


Figure 4.1: Constraint Anomalies in PSM-2 Model

The foregoing discussion leaves us with the PSM-1 model which places only one occurrence of the object in the database. This feature leads directly to a conceptual harmony between the physical model and the class-versioning approach. Unlike PSM-3, the PSM-1 model requires that some schema changes be propagated to sub-classes, which will cause the creation of new class-versions to those classes. However, other storage models tend to complicate the implementation of the class-versioning approach due to either the joins involved or possible data inconsistency. Therefore, we arrive at the conclusion that the PSM-1 model is the most suitable environment for adopting our approach, while PSM-3 and PSM-6 come in second and third places respectively.

For complex-objects, we assume the application of the *Direct Storage Model* in conjunction with PSM-1. In this case, versioning a complex class would have similar characteristics to those of PSM-1 model. The *Normalised Storage Model* , however, would be subject to the vertical partitioning requirements mentioned above.

4.2. Object Retrieval

Delivering objects to the user requires mapping them from their current class-version to the user-specified class-version. This process is performed according to the following sequence of operations:

(1) First the object is mapped to its version-collection definition using its class-version handlers.

(2) The mapped object is then transformed across the route of version-collections in the collection-graph to its destination version-collection.

(3) At the destination, the object's undefined attributes are masked-out to match the user's class-version definition.

(4) The resulting object is checked against attribute constraints declared in the class-version.

(5) Finally, the object is delivered to the user.

Since this sequence of operations is to be applied to each object in the result collection, so we perceived that it would be advantageous to run those operations simultaneously, in dataflow mode, to improve system performance. We shall elaborate on this issue throughout the discussion in Part-III of the potential of parallel processing in object manipulation. As far as simple-objects are concerned, the previous sequence of operation will be adequate and sufficient. However, with complex-objects, more work has to be done regarding constituent-objects. To retrieve a constituent-object, first the `ObjectID` and class-version should be determined. If the destination class-version of the complex-object specifies a class-version for the constituent-object, then the object is transformed using the aforementioned sequence of operations. Once the composite-object components have been transformed to the destination class-versions, the composite-object can be assembled and passed on to the user. It should be noted that a constituent-object can either be shared between several composite-objects, or dedicated to a single one. This is dependent on the employed data model.

4.3. Updating Objects

Updating persistent objects can be realized using either of the following methods:

(1) First, the object is retrieved into the application address space following the steps mentioned in Section 4.2. Upon retrieval, object updates are performed and the modified object is either:

- committed to the database in the current user-specified class-version, while the existing image of the object is deleted from the database in a separate operation.

- transformed back to its originating class-version (if different from the current user-specified class-version) replacing its existing unmodified image.

(2) A query is issued for object's class-version specifying either the `ObjectID` or attribute values as search predicates. Qualifying object(s) is(are) then located in the database and updated according to the update predicates of the query.

Updating objects of a class-version other than that specified for the user may result in modifying object data with illegal values (i.e., values beyond the domain of the object). Therefore, the DBMS should conduct constraint checks on update data at run-time and reject objects violating those constraints. The ENCORE system accounts for such situation by invoking a handler that returns an error code to the user. It is clear that the second method is cheaper than the first as no additional transformations are required. However, choosing the method of update would depend on the application requirements.

4.4. Methods of Querying Class-Versions

Querying class-versions can be performed using either of the following methods:

(1) This is considered the simplest method for manipulating objects in our approach. All objects belonging to the class under consideration are transformed from their current class-versions to the destination class-version

specified by the user. The user query is then applied to each transformed object and that which qualifies is passed on to the user.

(2) The user query related to a particular class-version is passed to a special preprocessor that accesses version-collection forward/backward transformers and generates a group of queries, each of which targets one version-collection in the collection-graph. Objects are passed to filters (software or hardware) that execute their class-version handlers and compare the mapped attributes to the predicates of the version-collection query. Qualifying objects are then mapped to the intended class-version that was originally declared in the user query. This involves the forward/backward transformers along the transformation path. An example of querying class-versions, depicted in Figure 4.2 is shown in Figure 4.3(a) where a user query is issued for class-version CV_4 in CL_2. In order to access all objects of that class in the database, the user query is passed to the preprocessor which generates a query for CL_0. The generated query then enters another phase of preprocessing which generates two queries for CL_1 and CL_4. The query for CL_1 enters a new phase of preprocessing which generates a query for CL_3. Generated queries are then resolved as described above.

(3) The user query is passed to the preprocessor that accesses both class-version handlers and version-collection forward/backward transformers and generates a group of queries, each of which is related to one class-version from the class-version hierarchy. Each class-version query is then resolved and qualifying objects are handled as described earlier in Section 4.2. Figure 4.3(b) illustrates the additional preprocessing phases required to generate class-version queries from version-collection queries of the previous example.

From the query language perspective, an SQL-like query needs to specify the targeted class-version of each class involved, e.g.,

```
Select * From Employee(CVx) where Salary > £5000;
```

where CV_x specifies version x of class `Employee`.

Query generation according to the latter two methods would then proceed. We shall elaborate on query language aspects in Section 4.6.

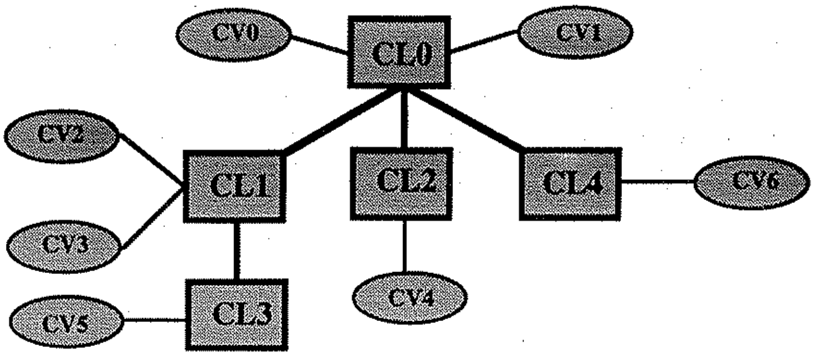


Figure 4.2: The Collection-Graph

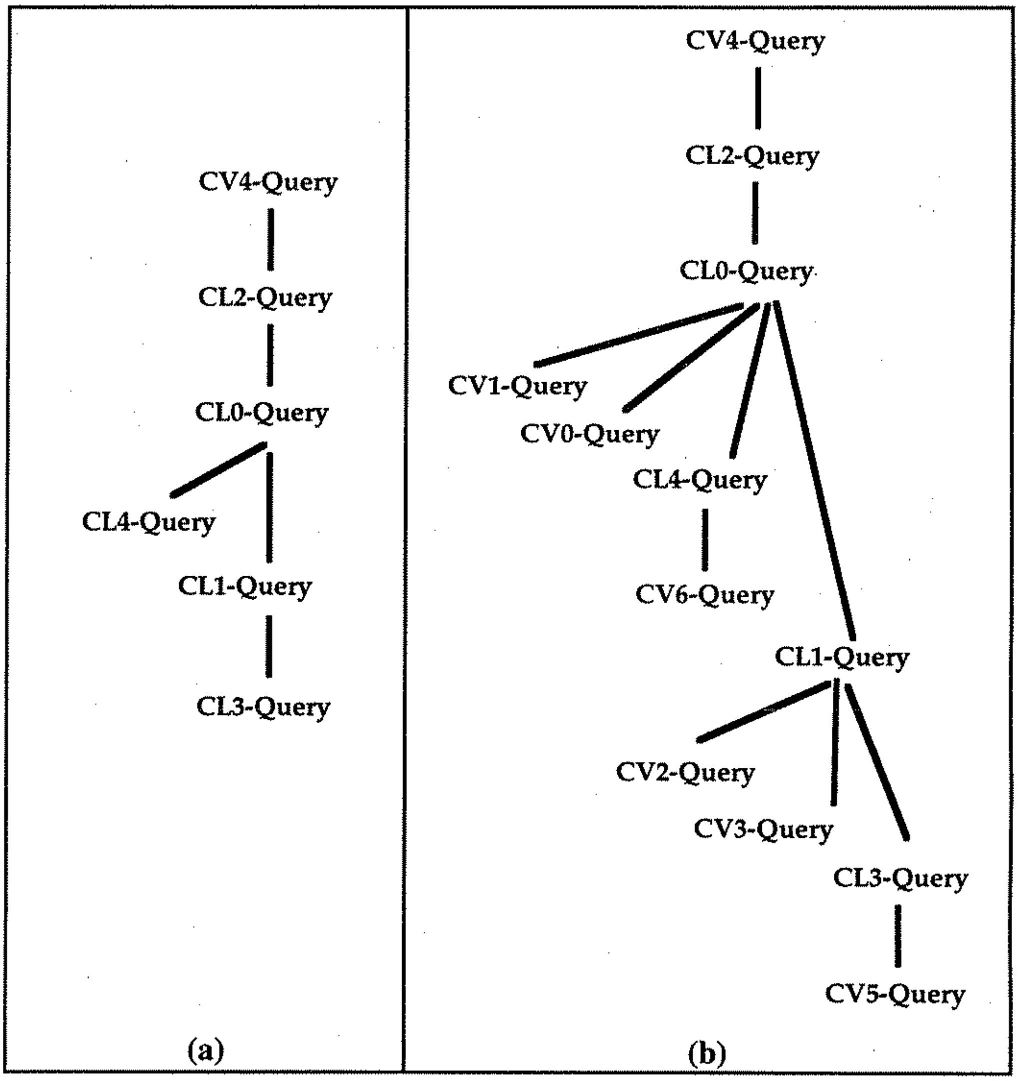


Figure 4.3: Query Generation from Class-Version CV_4

Despite the simplicity of the first method, it is considered the most expensive since it requires the transformation of all class objects, whether they qualify or not. This may lead to extended query service time and reduced object sharability due to extended lock time. The third method, in contrast to the second, imposes additional overhead in generating class-version queries but optimises the database filter operation. In either case, the transformation of objects to the destination class-version would only involve the qualifying objects, and not all objects of the specified class as was proposed by the first method. This would help minimize the required system resources, including processing and work space. It would also improve object sharing as only qualifying objects will be locked according to user specification.

The overhead of generating and processing class-version queries will depend on the number of class-versions and version-collections as well as the predicates involved in user query. For interactive environments involving large collection-graphs, the overall system performance may be affected by the query generation process. So, we consider the potential of parallel processing in query generation in Part-III.

4.5. Query Optimization

The main objective of query optimization is to choose the least costly execution plan and minimize the query service time. In relational database systems, such process would decide on the processing plan to choose for a query from a set of alternative plans. In doing this, the query optimizer takes into consideration (amongst other factors) how relations are best accessed (i.e., via indices and which index to choose, if any, or via sequential scanning of relation pages even with the presence of indices) and the best order of processing the predicates. Query optimization for object-oriented databases can be complicated by the fact a query may contain methods whose cost is difficult to predict and would make the result of the query dependant on the order of evaluation of predicates [Bertino 93]. Different techniques for query optimization are proposed in the literature such as [Ahad 88, Zdonic 88]. However, we do not intend to concentrate on the application of those techniques. Here, we exploit the special features of the class-versioning approach that would help optimize user queries. We recognize certain conditions upon which we may decide whether a particular class-version should or should not participate in resolving a query. Such a decision may be taken during query preprocessing (i.e., query generation phase). Other conditions pertain to individual object eligibility to reside in the destination class-version. Such conditions come into effect during query processing (i.e., at run-time). In what follows, we shall describe those conditions.

(1) When the interrogand is represented in the class-version by a handler that returns a default value, then that value is compared with the query-specified value. Depending on the outcome of the comparison, all, some, or none of the objects of that class-version may qualify for the query.

(2) When the interrogand specifies a value, or a range of values, that lies beyond the domain of the corresponding attribute in the targeted class-version. In this case, no objects will qualify for the query from that class-version. This case is predictable with partially-compatible attributes.

(3) When any of the ANDed interrogands qualifies for either of the previous conditions.

If the targeted class-version is expected to return no objects for the query according to any of the aforementioned cases, then it should be excluded from the query evaluation.

(4) When a user query specifies a range of values for the interrogand which is partly convertible to the domain of a class-version, then the query optimizer may add additional conditions to the generated query for that class-version to eliminate incompatible objects. The added conditions may involve the mapped attribute's *upper-bound* and/or *lower-bound* (see Section 3.3) in the destination class-version. Assume, for example, that we have two class-versions CV_X and CV_Y of class `Employee` where the former class-version specifies domain range £1000-7000 for attribute `Salary`, while the latter extends that range £1000-9000. Let a user-query for CV_X be:

```
Select * From Employee(CVX) where Salary > £5000;
```

In the absence of this rule of optimization, the generated query for CV_Y would be:

```
Select * From Employee(CVY) where Salary > £5000;
```

This could return employees objects whose salaries exceed £7000 and would then disqualify after mapping them to CV_X since they violate the domain range of `Salary` in CV_X . To avoid mapping objects destined to disqualify because of domain constraints, we apply this rule that generates the modified query for CV_y :

```
Select * From Employee(CVy) where Salary > £5000 And
                    Salary ≤ £7000 ;
```

Such query would isolate employees with salaries exceeding CV_x upper limit of salaries, at CV_y prior to any mapping taking place.

Looking at class-versions from this perspective, we find that they have the effect of partitioning database objects of the same class into subsets with each subset related to a version of the same type. This, in effect, would improve the performance of query evaluation by considering only object subsets with non-zero hit ratio and excluding those that do not qualify for the query beforehand.

(5) This condition is related to the formation of the collection-graph at schema editing stage rather than at query processing stage. As we mentioned in Subsection 3.9.2, connecting a version-collection to the collection-graph lattice might have more than one alternative all of which would provide safe paths. One factor that can be involved in selecting the connection is the cost of executing transformation functions along the routes leading to the version-collection and expected to be accessed by user-queries.

(6) In the case of permitting cycles to exist in collection-graphs, then evaluating the cost of executing transformation functions along alternative routes could help in query optimization by choosing the cheapest route.

Some commercial database systems provide tools for the user to explore the cost of processing his query before actually executing it. For example, the EXPLAIN facility of SQL [IBM 87] performs query optimisation procedure and provides the user with weighted cost of his query and its processing plan. With the availability of such facility in OODBMS and by providing it with means of analysing the above-mentioned considerations, the user would have the ability to predict the cost of his queries and to manage his evolving schema in an efficient way.

4.6. Programming Language Aspects

Khoshafian and Abnous [Khoshafian 90b] recognise six approaches for incorporating object orientation capabilities in databases:

(1) Novel database data model/data language approach.
(2) Extending an existing database language with object orientation capabilities.
(3) Extending an existing object-oriented programming language with database capabilities.
(4) Providing extendible object-oriented DBMS libraries.
(5) Embedding object-oriented database language constructs in a conventional host language.
(6) Application-specific products with an underlying object-oriented database management system.

We considered the third approach to support manipulating objects within the context of the class-versioning approach. The choice was made for its practicality and relative simplicity for implementing our prototype for accessing persistent objects (discussed in Chapter 7). Achieving this goal by embedding a database query language into a conventional programming language to access and manipulate a database is well known to cause *impedance mismatch*. Impedance mismatch often occurs between different models of data embodied in both the query language and programming language [Atkinson 78]. Alltalk is an example of an object-oriented programming and database system, based on Smalltalk-80 [Riegel 88]. Alltalk provides the programmer with a single syntax for accessing objects, be they in memory or on disk. Thus, it eliminates impedance mismatch between the programming language and the database sub-language. On the other hand, the Object Data Management Group (ODMG) presents in its standard for object model [ODMD-93] a query language, named OQL, as an extension to SQL to be used in C++ programming. We followed suit of Alltalk in seamlessly extending the capabilities of C++ [Gorlen 91, Schildt 91], the chosen programming language for our implementation. The language choice was motivated by C++ popularity. The ODMG, among many other experts in the field of object-orientation, perceives that the most important programming language for OODBMS has proven to be C++.

We apply the following database extension class-methods that may be overloaded (as application-related methods) in each declared class: `search()`, `fetch()`, `remove()`, `insert()`, and `modify()`. The `search(max-obj-count)` method causes the selection operation to terminate after a number of qualified objects equal to max_obj_count is selected. For example, with max-obj-count set to 1, the selection ends at the first hit of a qualifying object. This method returns the actual number of selected objects when the query ends. The user may apply this count in fetching qualifying objects, one at a time, in any order into main memory with method fetch(obj_sequence) where obj_sequence is less than or equal to that count. The insert() method commits its invoking object to the database, while method remove() deletes the object's copy in the database (located using object_id). Method modify() replaces the object's copy in the database with its memory-resident version (using object_id to locate the copy). Figure 4.4 shows a sample user declaration of three classes prior to the addition of the aforementioned class-methods. The class declarations, placed in a header file, are passed on to a proposed C++ extension preprocessor (shown in Figure 4.5) which embeds those class-methods into user declarations and generates their appropriate method-implementations. Figure 4.6 illustrates the preprocessed class declarations with new methods highlighted. We implemented this functionality in a preprocessor for accessing objects stored in the IFS/2 backend machine (referenced in Chapter 2) from a SUN workstation. The implementation was then updated to access objects in our prototype of object manipulation (discussed in Chapter 7) that support single inheritance.

In order to have a complete implementation of our proposal, we perceive that the declared schema in user source-code should be combined with the current database schema (resident in the DBMS) and transformed within the DBMS into class-versions and version-collections as necessary. This process is indicated in Figure 4.5 with a link between the DBMS and the preprocessor. Generated class's version_id would then be embedded in user source-code by the preprocessor as a new class attribute. As shown in Figure 4.5, extended class declarations should be passed on alongside the application source code to the C++ Compiler. The output code of the compiler would then be linked with DBMS runtime routines that enable class-methods to access the DBMS. Executing the generated object-code can then take place.

Unlike ODMG proposal, we propose unifying the manipulation of transient and persistent objects using the language constructs. Though not implemented in our prototype, we propose using nested conditional C++ statements to build-up nested queries instead of SQL-like phrases embedded in any C++ statements. Figure 4.7 illustrates this proposal where persistent objects are declared in the same way as transient objects. They are located and manipulated (i.e., selected, fetched, inserted and deleted) in the DBMS (executing in a host or a backend machine as in our prototype) using class methods embedded into each class declaration. Once in the user's local memory, persistent objects are manipulated in the same way as transient objects using programming language operations and application-related methods.

Declaring persistent objects in the extended C++ can be achieved by affixing the keyword `persist` to the object name (`emp_persist` in our example). Conditional statements involving the persistent object (the `while` statement in Figure 4.7) would imply the search predicates that are part of the user-query. Any search() method following such conditional statements would act on the basis of the implied predicates. Once a persistent object is fetched into its declared in-memory object, it can be manipulated in a similar way to transient program objects. Introducing extended attribute semantics (see Section 3.2) would require extending C++ typing facility to include domain range, default values, and units of measurements. The alternative to this approach is to embed attribute semantics as preconditions and postconditions in class methods. A graphic-user-interface (GUI) may be employed to declare attribute semantics and modified class definitions whereby the corresponding C++ declarations are automatically generated. This approach achieves a seamless handling of persistent objects without the introduction of SQL-like semantics to the programming language. Although our proposal is not as user-friendly as SQL-supported programming, C++ programmers should not face difficulty in adopting to it.

4.7. Conclusions

In this chapter, we discussed the applicability of physical storage models for simple and composite objects. We noted the requirements imposed by our approach to adopt some of those models regarding compatibility of attributes, and accessibility of derived attributes. We also distinguished between the models in terms of processing overhead involved in joining object attributes and storage space optimisation. We described the alternative methods of retrieval and update of simple and complex-objects and highlighted their processing requirements. We also presented alternative methods for querying the database and the cost involved in each method. Transforming all objects of a class to the class-version specified in the query prior to restricting them was the most expensive method. Version-collections and class-versions query generation from user-query were two alternative methods that would limit the number of transformed objects though the version-collection method would incur special implementation requirements. With respect to query optimization, we described the aspects inherent in the class-versioning approach that would contribute to improving query execution cost.

Finally, we proposed a C++ extension that would facilitate manipulating instances of class-versions in a seamless way, eliminating impedance mismatch. The language extension for accessing persistent objects in the DBMS and its supporting preprocessor was implemented for single inheritance class hierarchies to access objects stored in the IFS/2 backend machine (referenced in Chapter 2) from a SUN workstation. The implementation was then updated to access objects managed by our prototype (discussed in Chapter 7). We proposed extending the preprocessor to allow querying the database using nested conditional C++ statements rather than using SQL-like statements to eliminate impedance mismatch.

```
// class declarations

class Person
      {
      char         name[40];
      int          age;

      // declaration of class methods

      };

class Spouse       // would be a constituent-class of Employee
      {
      char         name[40];
      int          age;
      char         status;     // 'd' divorced, 'm' married

      // declaration of class methods

      };

class Employee: public Person
      {
      int          salary;
      Spouse       partner;

      // declaration of class methods

      };
```

Figure 4.4: Class Declarations Prior Submission to the Preprocessor

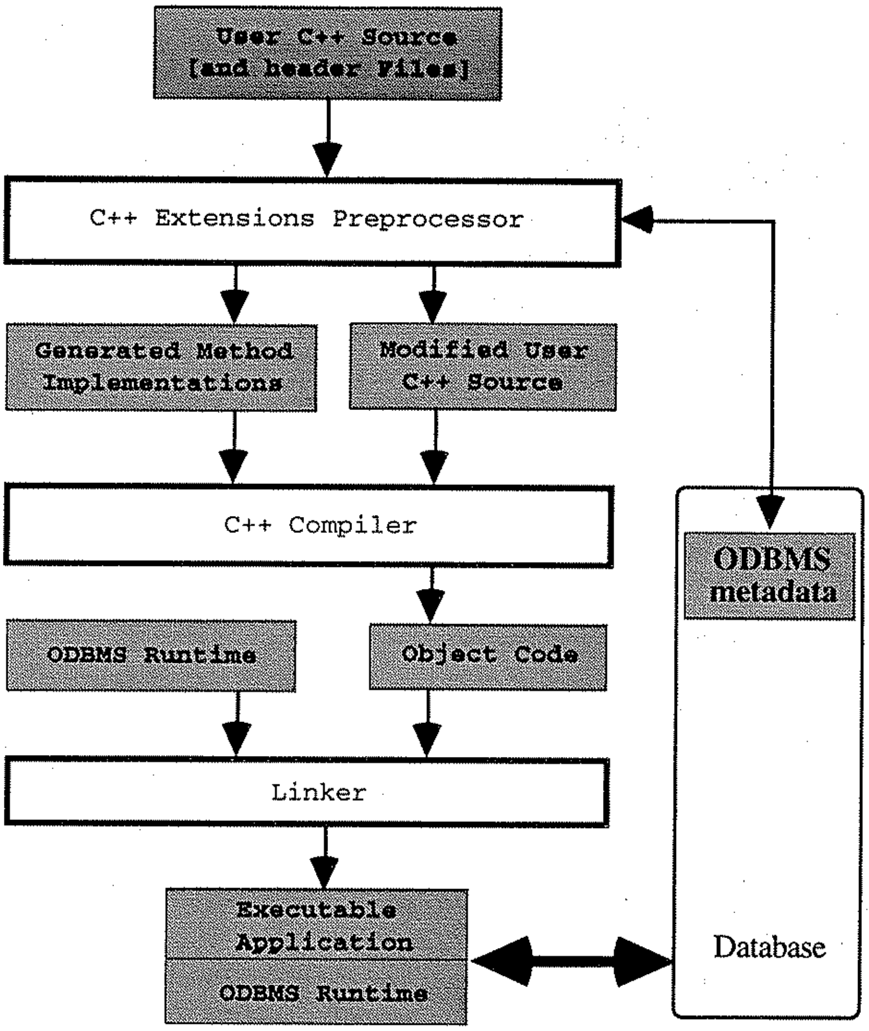


Figure 4.5: Supporting C++ Extension for Persistent Object Manipulation

```
class Person
      {
      char        name[40];
      int         age;
      long        obj_id;     // object's unique ID
      static      int   class_id;
      static      char* class_name;
      static      int   version-_id;

      public:

      int   fetch(long obj_sequence);
                              // get result from DBMS
      long  search(long max_obj_count);
                              // search database for object
      long  remove();         // remove matching object from DB
      int   insert();         // store object in DB
      int   modify(Person update_obj);
                              // update object in DB

      // declaration of class methods

      };

class Spouse      // would be a constituent-class of Employee
      {
      char        name[40];
      int         age;
      char        status;     // 'd' divorced, 'm' married
      long        obj_id;     // object's unique ID
      static      int   class_id;
      static      char* class_name;
      static      int   version-_id;

      public:

      int   fetch(long obj_sequence);
                              // get result from DBMS
      long  search(long max_obj_count);
                              // search DB for object
      long  remove();         // remove matching object from DB
      int   insert();         // store object in DB
      int   modify(Spouse update_obj);
                              // update object in DB

      // declaration of class methods

      };

class Employee: public Person
      {
      int         salary;
      Spouse      partner;

      public:

      int   fetch(long obj_sequence);
                              // get result from DBMS
      long   search(long max_obj_count);
                                 // search DB for object
      long   remove();           // remove matching object from DB
      int    insert();           // store object in DB
      int    modify(Employee update_obj);
                                 // update object in DB

      // declaration of class methods

      };
```

Figure 4.6: Preprocessed Class Declarations

```
// class version declarations
#define     ALL            0        // all versions of a class
#define     EMPFAM         2        // version #2 of class Employee

// preset values
#define     MAX_COUNT      10       // maximum number of qualifying
                                    // objects allowed in the result
main()

Employee    emp[MAX_COUNT];         // list of in-memory objects
Employee    emp_persist;            // persistent object declaration
query       out_query;              // a query structure that would contain
                                    // the predicates following preprocessing
long        count;

out_query.version = EMPFAM;         // which source versions

while (emp_persist.age == 30 && emp_persist.salary >= 1000 &&
                        emp_persist.partner.status == 'd')
                                    // predicate declaration
   {
   count = emp_persist.search(4);
                                    // count qualifying objects but
                                    // 4 at most are required

   for (i = 0; i < count; ++i) emp[i] = emp_persist.fetch(i);
                                    // load qualifying objects into
                                    // memory object list
   }

out_query.version = EMPFAM;         // which source versions

for(;;)                             // no predicates for this query
   {
   count = emp_persist.search(99999);
                                    // count existing Employee objects

   for (i = 0; i < count; ++i) emp[i] = emp_persist.fetch(i);
                                    // load qualifying objects into
                                    // memory object list
   }
```

Figure 4.7: A Sample User Program

This page is intentionally left blank

Chapter 5
Approach Applicability : Case Studies

Throughout the discussion of this part of this thesis, we considered the basic object-oriented data model for the application of our class-versioning approach. In this chapter, we extend our study to the application of our approach to extended object-oriented and semantic data models. In section one, we discuss how collections of objects may be manipulated within the context of our approach. In section two, we present a case study of adopting our approach to the OSAM* data model. Finally, we summarize conclusions drawn from this study.

5.1. Extended Object-Oriented Data Models

Some OODBMS such as GemStone [Penney 87, Bretl 89] and the standard proposed by the Object Data Management Group (ODMG) for the object model support object collections of types: Set, Bag, List, and Dictionary. Given that bags and lists may contain duplicate objects, having different versions of the same collection class may lead to the existence of several versions of the same member-object in the same or in a different class-version. This situation may be realized if one member-object, that has duplicates in the same class-version, migrates to a different class-version and is then updated. If the update makes the occurrences of the object incompatible, then we would have a state of inconsistency in the database. To avoid such inconsistencies, the implementation environment should ensure that all occurrences of the object reside in the same version of the collection class, and be updated all together, even if the operation underway targets a single occurrence. The reliance on the implementation emerges from the fact that the class-versioning approach does not support object versioning.

5.2. Extended Semantic Data Models

Gray states the aim of conceptual data modelling is to capture description of objects and their behaviour in the real world and to find structured representation for them in the database. Classical data models, namely the relational, the hierarchical and the network models, are not suitable for conceptual data modelling. The major reason for this is that all three models fail to capture much of the semantics associated with the data The deficiencies of classical data models for conceptual data modelling have triggered intense research work in the data modelling area. This work has been popularly known as ***semantic data***

modelling. Basically, these models aim to capture the meaning of the data in a more or less formal way, so that database design can become systematic and the database itself can behave intelligently [Gray 92].

In semantic data modelling, information is modelled in terms of atomic units called entities or objects. Most semantic data models capture important static constraints as part of the structure itself. Those constraints are categorised in [Urban 87]. Semantic data models and object-oriented data models share some common notions. Both employ the notion of objects with unique object identity organized around types or classes. Both support the notion of inheritance through the hierarchy of types. Although object-oriented data models possess additional properties like encapsulation and late binding that are not normally part of semantic data models, they do not typically incorporate all the constraints provided by the semantic data models [Bancilhon 88]. Since our proposed class-versioning approach is concerned with mapping the structure of objects between different class-versions, and does not deal with object behaviour, so is worthwhile to investigate the applicability of the approach to semantic-data models and the limitations of class-versioning caused by the semantics-capturing capabilities of such models. We chose the Object-Oriented Semantic Association Model (OSAM*) presented by Su and Lam, in [Su 89, Lam 89, Lee 89], to conduct the following case study. The reason for choosing that model is that it is semantically rich and combines the features of both the semantic and the object-oriented data models. In the following, we shall present the main concepts of the OSAM* data model, and discuss the different issues arising from the application of the class-versioning approach the data model.

5.2.1. A Review of the OSAM* Data Model

The OSAM* model extends the association types between classes in the conventional object-oriented data model. In addition to generalization and aggregation, it introduces interaction, composition, and crossproduct associations. The OSAM* distinguishes between two types of object classes: the ***entity object class*** (E-class) which models the structure and behaviour of objects in the application world, and the ***domain object class*** (D-class) that primarily serves as descriptive data of some other objects. The latter type corresponds to attributes in the object-oriented data model, and they also can be simple or complex. The D-classes can be shared amongst different E-classes. The model calls the class which enters into association with other class(es), the ***defined class*** (DC), while the related classes are called the ***constituent classes*** (CCs). The supported semantic association types are:

Generalization **{Type-G or G-Association}**(E->E)[1]

models the superclasses-subclasses concept of the conventional object-oriented data model.

Aggregation **{Type-A or A-Association}**(E->E/D)

represents the class attributes as in the conventional object-oriented data model.

Interaction **{Type-I or I-Association}**(E->E)

an instance in the association-class represents an interaction between or among some instances in its constituent-classes (called Type-I attributes in Figure 5.1a). An example of this type of association is illustrated in Figure 5.4.

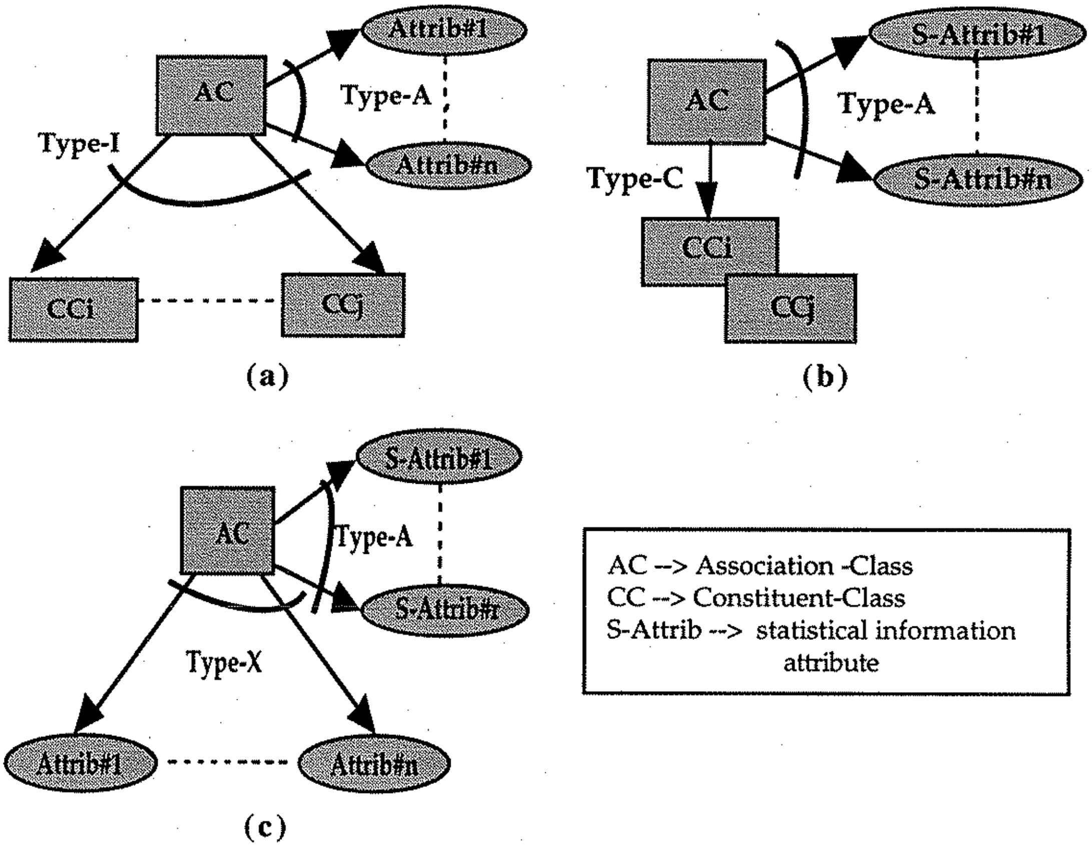


Figure 5.1: (a) The I-Association Class, (b) The C-Association Class, (c)The X-Association Class

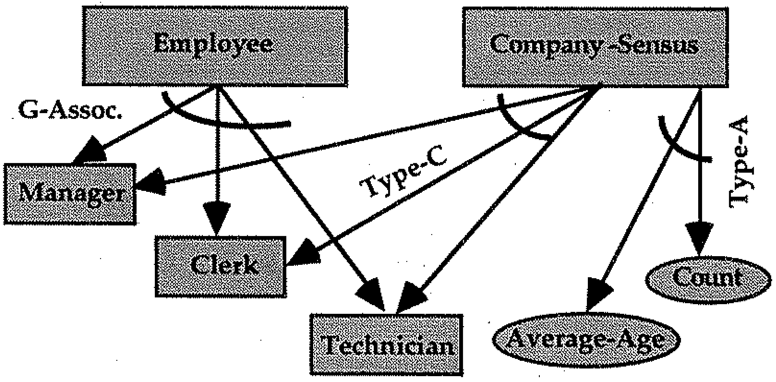
[1] This describes the types of the *Association* and the *Constituent* classes respectively, i.e. either E-class or D-class.

Composition **{Type-C or C-Association}**(E->E)

characterizes the collection of objects of each constituent-class as a whole. Constituent objects cannot inherit any properties of their association-class and they are not objects of that class (see Figure 5.1b). An example of this type of association is shown in Figure 5.2, where the association class Company-Census characterizes the different employee categories within a company. Each of those categories is represented by a class in the database schema and has class Employee as its superclass.

Crossproduct **{Type-X or X-Association}**(E->D)

partitions the collection of objects of the association-class into categories based on the distinct values of one or more of its Type-X attributes. An example of this type of association is shown in Figure 5.3, where the association class Company-Cars categorizes company cars according to their types and manufacturer using Type-X attributes Car-Type and Manufacturer. Each category has its cars counted and cost-evaluated using Type-A attributes Total-Cost and Count. Note that Car-Type and Manufacturer attributes do not necessarily have corresponding classes in the schema.



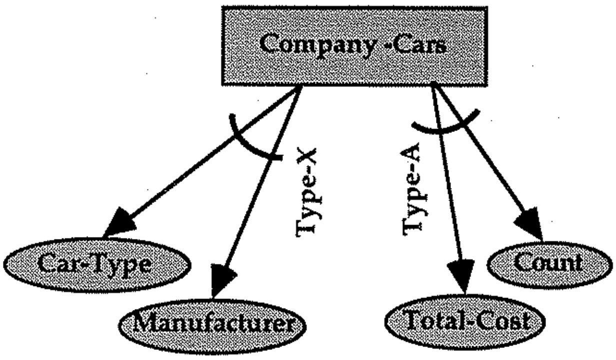
(a)

Census-ID	Category	Average-Age	Count
01	Manager	40	12
02	Clerk	45	23
03	Technician	35	40

(b)

Figure 5.2: The company-census Composition Association

The Type-A attributes associated with a association-class having composition or crossproduct association can be summary attributes such as sum, count, average and minimum. They cannot be used to characterize an individual object since it is not meaningful to summarize a single object. The I, C, and X-Associations model mutually exclusive semantic properties of objects.



(a)

Category-ID	Car-Type	Manufacturer	Total-Cost	Count
01	Truck	Ford	1.2M	6
02	Truck	Fiat	0.8M	3
03	Limousine	Ford	0.05M	3

(b)

Figure 5.3: The Company-Cars Crossproduct Association

In OSAM*, global changes may be required to bring database objects up-to-date. Such changes can emerge from schema evolution or object manipulation operations (update, deletion). Other changes can affect several objects simultaneously. For example, updating or deleting a *constituent* object in type-I/C relationship might affect several objects of the *defined* class (similar to referential integrity in relational database systems).

5.2.2. Class-Versioning in the OSAM* Data Model

In the class-versioning approach, we outlined the supported schema changes related to the IS-A and IS-PART-OF relationships. So, as far as the OSAM* Data Model is concerned, we only dealt with the Generalization and the Aggregation Associations. In what follows, we shall study the applicability of those changes to each of the associations supported by the OSAM* and deduce the appropriate grouping of schema changes accordingly. We consider a change to the schema valid if:

- The change does not contradict the semantics of the association under consideration.

- Handling the change lies within the scope of operations that may be carried-out by class-version handlers and version-collection transformers.

As far as the OSAM* declared class-operations are concerned, they are treated in similar way to that of class-methods in the context of schema evolution. So, in the following discussion, it is implied that changes to OSAM* class-operations belong to Group-I and are applicable to all types of association.

The Generalization and the Aggregation Associations

- The class-versioning approach can be applied to both associations in a similar way to that of the conventional object-oriented data model. However, some changes may be restricted to the aggregation association when combined with any of the remaining association types. This will be revealed in what follows.

The Interaction (I) Association

Class-Structure

- Creating a new version of a constituent-class (as a result of Group-I or Group-II changes) or deleting a constituent-class-version (which may take place upon reorganizing the database) will not cause versioning its Interaction association-class. This is due to the fact that instances of constituent-classes are referenced by their unique `ObjectIDs`. Manipulating those objects, on the other hand, should involve the appropriate transformations to/from the user's class-version.

- Adding/removing Type-I or Type-A attribute to/from the I-Association is not permitted when versioning an association-class since instances of other versions will not conform to the semantics of the new version, and vice-versa. This can be illustrated by the example shown in Figure 5.4. Creating a new version of class `Teaches` eliminating Type-I attribute `Student` will produce conflicting objects from the existing class-versions. Accessing objects of `Teaches` V_i from the new class-version would return instances #1 and #2 of `Lecturer` #100 as teaching `Subject` #A1 for two different settings of `Duration` which is semantically wrong.

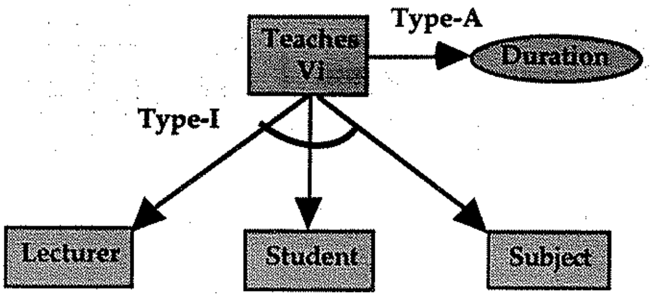


Figure 5.4(a): The Teaches Association-Class (1)

OID	Lecturer-ID	Student-ID	Subject-ID	Duration
#1	#100	#201	#A1	1
#2	#100	#202	#A1	2
#3	#100	#203	#B1	1

Figure 5.4(b): Relation Representation of Teaches Class (1)

Another example is shown in Figure 5.5 where class-version V_i of class Teaches has two Type-A attributes Location and Duration. If we create a new class-version of Teaches by dropping attribute Location, For Lecturer #100 teaching Student #202 Subject #A1, we get two instances (#2, #3) at the new class-version, each of them depicts a different setting of Duration. Getting two such instances for a query would be interpreted as a presence of inconsistency in the database.

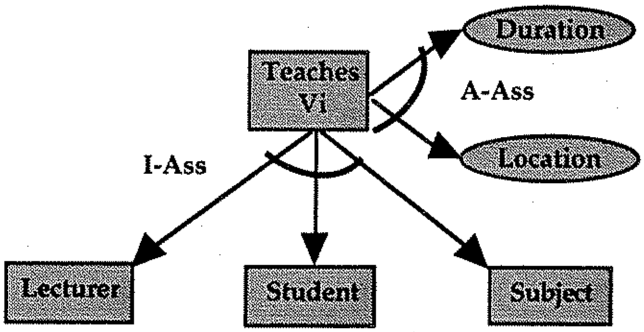


Figure 5.5(a): The Teaches Association-Class (2)

OID	Lecturer-ID	Student-ID	Subject-ID	Duration	Location
#1	#100	#201	#A1	1	EN
#2	#100	#202	#A1	1	CS
#3	#100	#202	#A1	2	EN
#4	#100	#203	#B1	1	CS

Figure 5.5(b): Relation Representation of Teaches Class (2)

Class-Hierarchy

- Removing a constituent-class entering this kind of association from the class-hierarchy is not permitted as it will invalidate the corresponding Type-I attribute in the association-class and would imply dropping that attribute. As we mentioned earlier, dropping Type-I attributes is not permitted.

Attribute-Semantics

- Changing the semantics of Type-A attribute in an I-Association class is a Group-II change that will cause creating a new version in a different version-collection for that class.

Object-Manipulation

- Deleting an object whose class represents a Type-I attribute will cause deleting all association-class instances referencing that object (similar to referential integrity constraint in RDBs).

The Composition (C) Association

The semantics of this kind of association is extended under the class-versioning approach to characterize each constituent-class entering this association on version-basis instead of a class as a single entity. So, each constituent-class-version will have a corresponding instance of the C-Association class in which Type-A attributes hold statistics about that class-version, and the domain notion of the Type-C attribute is extended to include class-versions. The overall statistics related to each constituent-class represented in this association can be produced, collectively, by querying its class-versions instances of the C-Association class.

Class-Structure

- Dropping a constituent-class attribute upon which a statistic is based would cause default settings to be enforced for each Type-A attribute based on the dropped attribute. Accepting a default statistic is mainly application-dependant. If default statistics are not accepted, then it would be appropriate to restrict this kind of schema change.

- Adding/dropping a Type-A attribute to/from the association-class is a Group-I change that will require the creation of a new version of the association-class, in the same version-collection. However, updating an instance of an association-class-version lacking a required Type-A attribute means that a statistic has to be calculated for that instance, either from scratch or derived from its existing Type-A attributes. In this case, the instance should migrate to an appropriate association-class-version that preserves all instance data. This requirement avoids re-calculating lengthy statistics when manipulating objects of other association-class-versions.

- Dropping a Type-C attribute is not permitted as it would invalidate the association-class.

Attribute-Semantics

- Adding/removing a constituent-class or any of its versions to/from this kind of association affects the domain range of the Type-C attribute. If the existing association-class-version specifies the domain range for that attribute, then the change would have the effect of a Group-II change that causes a new version of the association-class to be created in a different version-collection. Otherwise, the change would have no effect on versioning the association-class.

- The domain-range for a Type-A attribute in the association-class should be wide enough to cover all possible statistical values, otherwise, updating the value of such attribute may fail for instances of the association-class that are being manipulated at their current class-version.

- Changing other semantics of Type-A attributes of the association-class is considered Group-II changes and will have a similar effect to Group-II changes mentioned above.

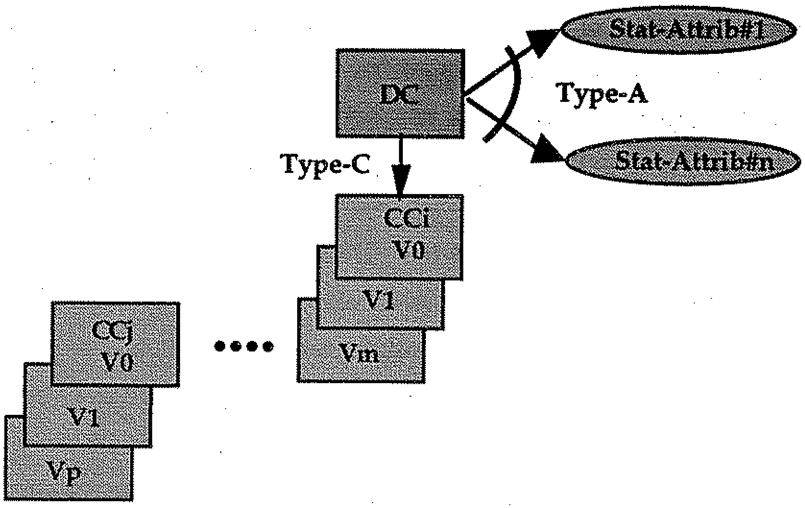


Figure 5.6: Versioning a Composition Association Class

Class-Hierarchy

- Removing a class entering this kind of association from the class-hierarchy has the same effect of removing a constituent-class from the C-association.

Object-Manipulation

- Updating, adding, or deleting an object of a constituent-class-version will affect only its instance of the association-class. Depending on the type of statistics maintained by the Type-A attributes (minimum, maximum, sum, average, etc.), re-evaluating those attributes will be required and may involve scanning all instances of the constituent-class-version, or building upon the existing statistics.

The Crossproduct (X) Association

This kind of association is not based on other classes existing in the class-hierarchy. So, modifying a different association-class should have no effect on crossproduct-association classes, unless the modification is performed on their superclasses. In the latter case, changes to the attributes of the association-class are subject to what follows.

Class-Structure

- Adding/removing a Type-X attribute to/from the association-class causes what might be called statistical disaggregation/aggregation. The OSAM* data model considers the possibility of providing application-related functions that can perform such statistics transformation on instances of the association-class. However, this process involves multiples of objects and requires querying class-objects beforehand. The class-versioning approach, however, deals with the transformation of individual objects. Thus applying the aforementioned OSAM* operations would not be possible. Accordingly, adding or removing Type-X attributes should not be permitted when versioning a crossproduct-association class.

- Adding/removing a Type-A attribute to/from the association-class does not affect the categorization introduced by the existing association-class-versions and in effect would provide more or less statistical information about each category. This is considered a Group-I change. When querying instances of this association based on a Type-A attribute that is being dropped in some versions of the association-class, then a proper default value for that attribute should be provided by a handler of those versions such that the query result becomes meaningful.

5.3. Conclusions

In this chapter, we examined the application of the class-versioning approach to extended object-oriented and semantic data models. We discussed how additional features embedded in those models may cope with and be dealt with in our approach. For object-oriented data models supporting collections of duplicate objects, inconsistency between such objects may arise when any of the duplicates migrates to a different class-version and is then updated. To avoid this situation, we perceived migrating the whole collection of objects and applying updates to all duplicates within the collection. We discussed in detail the application of the class-versioning approach to OSAM* as an example of extended semantic data models. We outlined the limitations imposed on schema evolution for some class associations due to possible incompatibility between class-versions or invalid semantics implied by migrating objects. The OSAM* case study provided a modified taxonomy of supported schema changes for each association type within the context of the class-versioning approach. The reason behind this investigation is to have an insight of the applicability of our approach to data models other than the basic object-oriented model. The case studies highlighted the possibility of facing semantic anomalies and limitations on supported schema changes when adopting our approach to data models other than the basic object data model. The case studies also underlined the requirement of automated management in DBMS for objects in certain cases as with object collections.

This page is intentionally left blank

SUMMARY OF PART II

In this part, we discussed the concepts and main features of the proposed class-versioning approach which is based on grouping versions of the same class into collections. Version-collections are provided with transformation functions that map instances of class-versions across the collections. We highlighted the objectives of the approach, the extension to the data model which it introduces. We presented a taxonomy of the supported schema changes as two groups categorized by their effect on attribute semantics. Consequently, we defined attribute compatibility which governs the grouping of class-versions into their collections and guarantees accessibility of class-instances from any of the class-versions. Intermediate class-versions were also proposed to insure safe mapping of objects and preserving the persistence of object data. Configuring class-versions and version-collections into collection-graphs as a tree or a lattice was also presented. Implementation aspects of the class-versioning approach were also investigated. We discussed the applicability of physical simple and complex object storage models to the approach. We presented the alternative methods of object retrieval and update, and querying class-versions. We elaborated on query optimization features inherent in the approach and presented a seamless programming language extension to query class-versions part of which was implemented in a prototype. Finally, we presented case studies of adopting the approach in extended object-oriented and semantic data models and highlighted semantic constraints and anomalies that may arise when adopting class versioning to such models. We also underlined tasks to be automated in DBMS for managing objects.

This page is intentionally left blank

Part III

Parallel Processing Aspects

Introducing the class-versioning approach to OODBMS incurs additional overheads on system resources, mainly due to the dynamic object transformations during run-time. To support our claim of the practicality and operability of the approach in real systems, we need to study the processing requirements of the approach, and the options it provides in that respect. We also need to examine the possible directions for improving existing systems performance to cope with the added workload. Since parallel processing was not dealt with in the literature from schema evolution point of view, we choose this area to conduct our investigation and we dedicate this part of the thesis to discuss the potential of parallel processing in supporting the class-versioning approach. Chapter 6 studies the aspects of parallelism inherent in the approach and investigates the ways to exploit those aspects to achieve better system performance. Chapter 7 describes our implementation of a prototype for manipulating simple objects using a multiprocessor machine running a distributed operating system. Chapter 8 demonstrates the prototype performance using a sample application that we set-up for testing. We also use simulation to study system performance in manipulating composite objects with different architectural specifications and process configurations. Chapter 8 also demonstrates the performance advantage of combining multitasking, process replication and associative techniques with pipeline and dataflow processing. This part ends with Chapter 9 in which we propose the design of an associative query-engine that realizes our simulation assumptions, made in Chapter 8, regarding processing speed-up with associative techniques.

This page is intentionally left blank

Chapter 6
The Potential of Parallel Processing

In this chapter, we investigate the potential of parallel processing in supporting the class-versioning approach. Section one explores parallelism inherent in our approach. In section two, we discuss the generation of class-versions queries from a user-query formulated for a specific class-version. Sections three and four discuss the manipulation of queries for simple and complex objects respectively, while section five deals with the prospects of data partitioning in our approach. In section six, we present a taxonomy of processes that may take part in object manipulation. Finally, a summary of the chapter's main topics is presented.

6.1. Parallelism Inherent in the Class-Versioning Approach

There are three aspects of parallelism to exploit in the class-versioning approach. The first is query generation for all version-collections or class-versions of classes referenced by the user-query. The overhead of generating such queries depends mainly on the complexity of the collection-graph of the class in question as well as the number of predicates involved in the user-query. We perceive that for interactive environments involving large Collection-Graphs, the overall system performance may be affected by the query generation process. Accordingly, we were motivated to consider the potential of parallel processing in query generation. The second aspect is resolving generated queries each of which may involve several transformation stages and may operate on a large number of objects. This can impose a non-trivial workload on uniprocessor environments and it has thus driven us to study the possible ways of exploiting the potential of parallel processing in speeding up this process. The third aspect is exploiting data partitioning, provided by the grouping of objects of the same class-versions, to improve object sharing and to optimize queries access to objects, thus permitting more queries to be resolved in parallel.

6.2. Query Generation

We enumerated earlier in Chapter 4 three different methods for performing a query on class objects. **The first method** did not involve any query generation and relied on the transformation of all class objects from different class-versions to the user's specified class-version. We shall refer to that method later on in this chapter. Now, we shall elaborate on how to exploit the parallel features embedded in the other two methods that involve query generation for all version-collections or class-versions from the invoked user-query.

Applying **the second method** of query generation for different version-collections was illustrated by the example of Figure 4.3(a) assuming the Collection-Graph depicted in Figure 4.2. In a uniprocessor environment, only a single query can be generated at any time. So, in that example, we may expect the query generation process to take 5T units of time to complete, assuming a single pre-processing invocation takes T units on the average (which can be in the order of tens of seconds, depending on the complexity of user-query). It is clear that the third phase of query generation involves the invocation of the pre-processor twice using the CL_0-query as an input. So, it is possible in a multiprocessor environment to run two processes of the pre-processor on two different processors and generate CL_1 and CL_4 queries simultaneously (see Figure 6.1a). This may reduce the time required for query generation by T units.

In Chapter 4, we also considered applying **the third method** to the same example (see Figure 4.3b) in order to generate queries for different class-versions in a uniprocessor environment. This would take 11T units where 6 additional pre-processing invocations are required to generate class-version queries from the relevant version-collection queries. In this case, we notice that more invocations of the preprocessor can be run simultaneously in a multiprocessing environment if allotting adequate number of processors is possible. For instance, 4 processors would be required for the third phase as shown in Figure 6.1(b). Thus, applying the third method in a multiprocessor environment would cost around 5T units which is the same as the cost of the second method in a uniprocessor environment and which is only T unit more than the second method with multiprocessing.

From this example, we see that parallel query generation can result in more time saving for complex collection-graphs, specially in multi-user environments. As the number of predicates in the user-query increases, more predicate transformations become involved in the pre-processing. This may alternatively be translated into increased T value and would consequently increase the overhead imposed by the third method. However, from the performance point of view, resolving class-version queries is much cheaper than version-collection queries, as we shall see later in the next subsection, and it overshadows the overhead incurred in applying the third method.

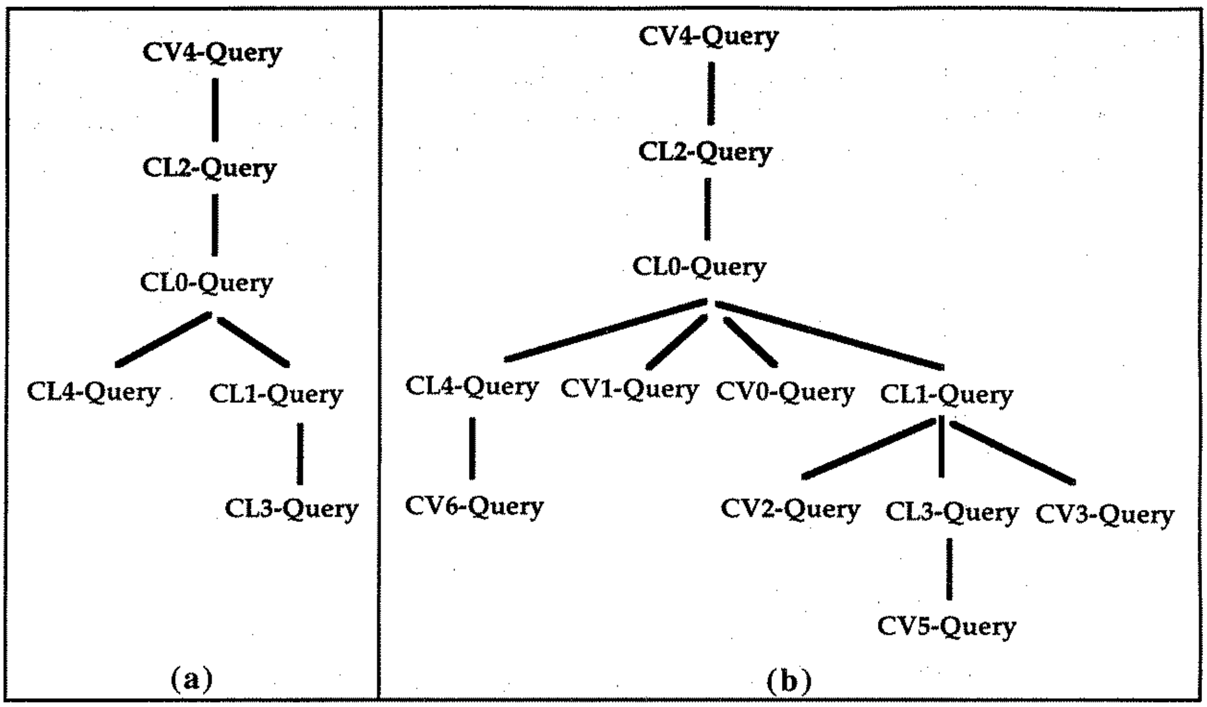


Figure 6.1: Parallel Query Generation from Class-Version CV4

6.3. Manipulating Simple Objects

Query graphs are normally constructed for their usefulness when studying query strategies. In relational databases, the query graph for a relation is the relation itself where the selection predicates are evaluated on the only node in the schema graph. In object-oriented databases, a query can be a combination of two types of predicates: ***simple*** predicate which is a simple attribute, and ***complex*** predicate which is a sequence of attributes on a branch of the aggregation hierarchy of a class. The query graph must, therefore, include all the classes, subclasses, and constituent classes to which any reference is made by the predicates. A single operand query in an object-oriented database may also involve joins of the classes along the inheritance hierarchy of the target class. Such queries are more powerful than single operand queries in relational databases which cannot refer to attributes other than those of the target class (or relation). We investigate in this section the alternative process configurations for queries manipulating simple objects, and we dedicate the following section to manipulating composite and inter-related objects.

Now, we elaborate on the efficiency of object mapping operations for each of the query methods mentioned in Section 4.4. Considering **the first method**, the mapping operations will be applied to all objects in the class, then the user-query is applied to each delivered object. This method, adopted in the CLOSQL system [Monk 92, 93], requires extensive processing resources and reduces object sharing as disqualifying objects will be locked alongside those qualifying during the execution of those operations. The other two methods, in contrast to the first, avoid the massive mapping of class objects throughout that sequence of operations. However, **the second method** for version-collection querying applies the generated queries following the mapping of all class objects to their version-collections. So, we discard this method as well even though it provides improved performance compared to the first method. As far as **the third method** is concerned, it avoids the massive mapping of class objects and transforms only qualifying objects resulting from applying class-version queries. Accordingly, this method takes the lead of the three methods in reducing the processing time required to manipulate objects of other class-versions, and in maximising object sharing as only qualifying objects are locked according to user requirements.

Since the third method seems more favourable, we confine the rest of our investigation to that method. As resolving queries involves all objects in the relevant class-versions, and since object mapping is done for all qualifying objects, so we perceive that it would be advantageous to exploit the potential of parallel processing to resolve class-version queries and execute the set of object mapping operations. We recognize three parallel processing configurations for resolving class-version queries. **The first** is shown in Figure 6.2 and is composed of a set of N processing nodes running in parallel. Each node executes a universal query-resolver and transformer that handles objects from any class-version of the same class, applies the relevant class-version query, and performs the sequence of transformation and checking operations mentioned in Section 4.2. Normally, we would expect this configuration to achieve a speed-up of N-fold compared to a uniprocessor system, assuming no lock-waits are taking place. **The second configuration**, shown in Figure 6.3, applies the pipeline approach where each class-version query or transformation stage is handled by a dedicated processor node. To resolve the queries generated in Figure 6.1(b), we require 4 version-collection transformers; 5 class-version query-resolvers; 1 class-version transformer. It is clear that some version-collection transformer nodes serve more than one transformation route. This has the potential of developing bottlenecks that may block pipeline flow if those nodes become overloaded. In this case, the blocked nodes may become under-utilized even with the introduction of inter-stage buffers, and the targeted system performance may not be achieved accordingly. To overcome this drawback, it would be advantageous to allocate more than one processor node for each shared pipeline stage. However, we think that this solution would complicate managing the allocation of processors and would require more inter-stage communication to locate the node that is willing to serve. **The third configuration,** shown in Figure 6.4, allocates a set of processor nodes for each transformation route and removes dependencies between service activities of class-versions. Although this configuration would provide better performance compared to the second, it requires more processor nodes to form the separate pipelines in the absence of multitasking–24 processors for this example as opposed to only 10 in the second configuration. In systems where the number of available processors is limited, it would be appropriate to postpone establishing pipelines till an adequate number of free processors exists. We assume here that the maximum number of stages in any pipeline, i.e., transformation route, can be supported for any class in the database by an equal number of processors. Alternatively, processor sharing between pipeline stages may be exploited. We perceive two benefits from the latter observation: increasing processor utilization and evenly distributing the processing workload among processors. In other words, if the processor node has a multitasking capability, and knowing that the overall performance of a pipeline is determined mainly by the workload of its busiest stage, then it would be advantageous to assign one stage from several pipelines to each processor so that it can evenly serve active stages and avoids being idle because of blocked pipeline stages. We followed this direction in implementing our object manipulation prototype (described in Chapter 7) and it proved to be efficient. We also investigate the advantages of processor sharing and multitasking for Configurations 1 & 3 in Chapter 8 using simulation.

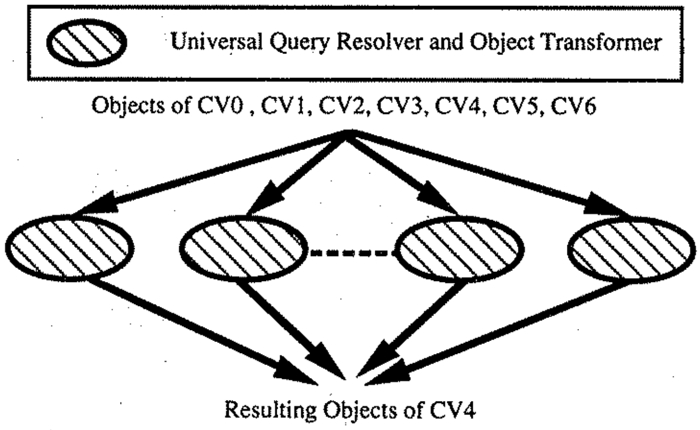

Universal Query Resolver and Object Transformer
Objects of CV0 , CV1, CV2, CV3, CV4, CV5, CV6
Resulting Objects of CV4

Figure 6.2: Parallel Object Mapping for Class-Version CV_4 (Configuration 1)

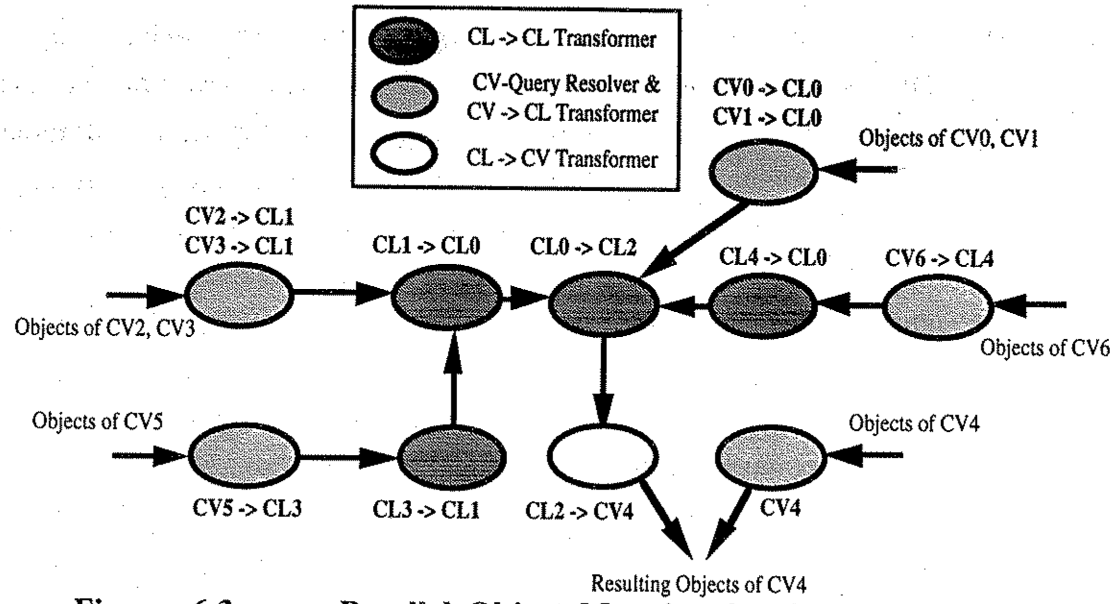


Figure 6.3: Parallel Object Mapping for Class-Version CV_4 (Configuration 2)

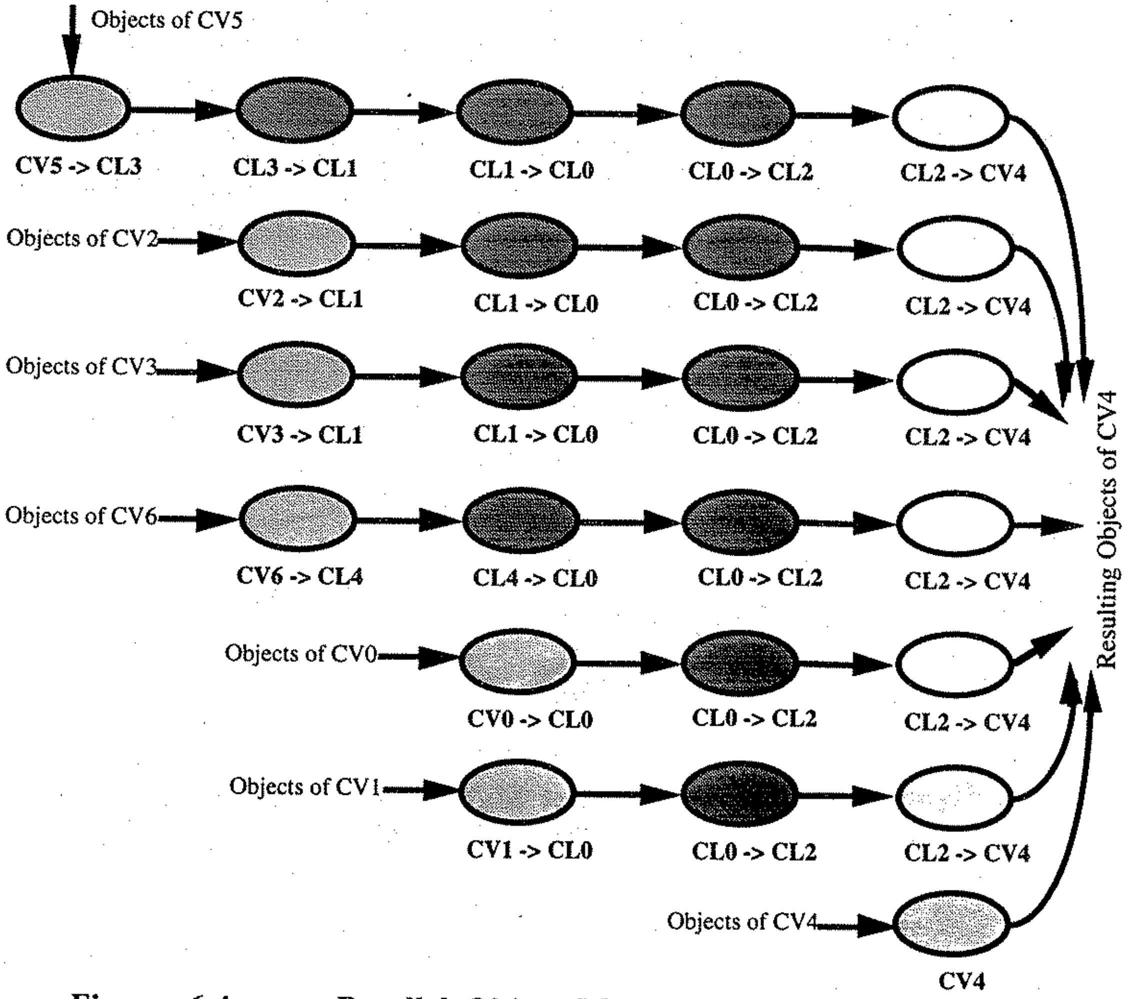


Figure 6.4: Parallel Object Mapping for Class-Version CV_4 (Configuration 3)

6.4. Manipulating Complex and Inter-Related Objects

We discussed in the previous section alternative process configurations for simple object transformation and mapping between class-versions. Querying simple objects does not involve references or relations between objects. On the other hand, querying complex and inter-related objects requires join operations to resolve object dependencies in addition to the restrict (i.e., selection) and transformation operations conducted on instances of each class participating in the query. So, we need to complement the previous investigation and study how process configurations of Section 6.3 would take part in querying complex objects and inter-related objects. In what follows, we present four alternative process configurations for resolving such queries. We would like to note that restricting the first class to be traversed in the query-graph always precedes any join operation and is the first process to execute in all configurations. So, the order of join and restrict processes in those configurations would relate to other class nodes in the query-graph rather than the first node.

(1) **Early Join, Deferred Transformation:** in which search conditions of the first class node to be traversed in the query-graph are applied to instances of its perspective class-versions, then qualifying instances are joined with all instances of the class in the following node. Joined (i.e., assembled) objects are then restricted according to the search condition of the class at that following node. The latter sequence of join and restriction is repeated till all nodes of the query-graph are traversed. Objects emerging from that sequence of operations represent qualifying objects for the generated query of their current class-versions. Before delivering those objects to the user, they may need to be transformed to the user specified class-versions. Such transformation would act on each component object within the assembled object, according to the designated transformation path of the object's class. We denote this configuration by the key expression 'Join-Then-Restrict, Deferred-Transformation', abbreviated to 'JN-RS-DT'. Now, we demonstrate the application of this configuration to Example-4 of Section 3.10. The query shown below specifies the destination class-version CV_2 of class `Employee`, and implies CV_7 of `Children`, and CV_6 of `Spouse`. This requires performing generated queries on all class-versions of the

participating classes. Figure 6.5(a)[1] illustrates the operations performed on Employee (CV$_1$), Children (CV$_5$), and Spouse (CV$_4$) in a response to that query. We adopt forward traversal of the query graph where Employee->Children path is served first followed by Employee->Spouse. Following the shown steps of Restricts and Joins, objects undergo three transformation stages, one for each of the classes Spouse, Children, and Employee. Qualifying objects in the result would conform to the destination class-versions (i.e., CV$_6$ of Spouse, CV$_7$ of Children, and CV$_2$ of Employee).

```
   Select * From Employee(CV2) where Salary > £5000 And
Spouse.Status = "Married" And Children.Sex = "Male" ;
```

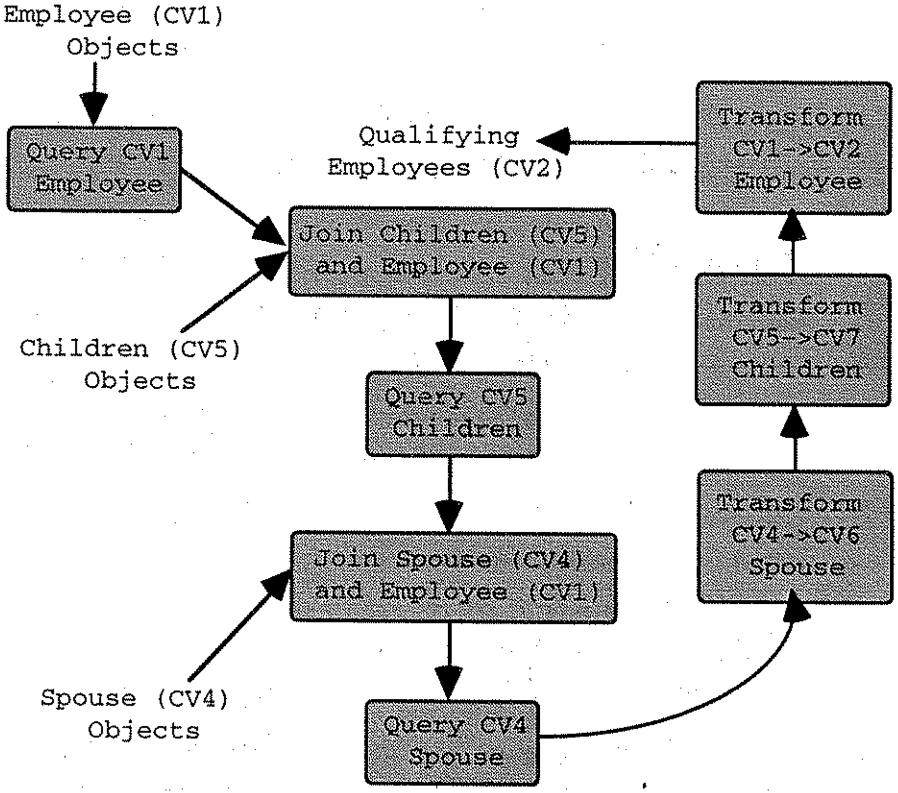


Figure 6.5(a): Configuration#1, Join-Then-Restrict, Deferred-Transformation for a Query on Class "Employee"

(2) **Early Restrict, Deferred Transformation:** this is a variation of the previous configuration where search conditions are resolved for their

[1] Other illustrations of Figure 6.5 are self-explanatory based on the current discussion

perspective class-versions then selected objects are joined according to the relationship (or association) between their classes, whether it is IS-PART-OF or else. Joined objects are then transformed to the destination class-version specified by the user, in a similar way to that of the previous configuration. We denote this configuration by 'Restrict-Then-Join, Deferred-Transformation', abbreviated to 'RS-JN-DT'. The application of this configuration to the `Employee` example is shown in Figure 6.5(b).

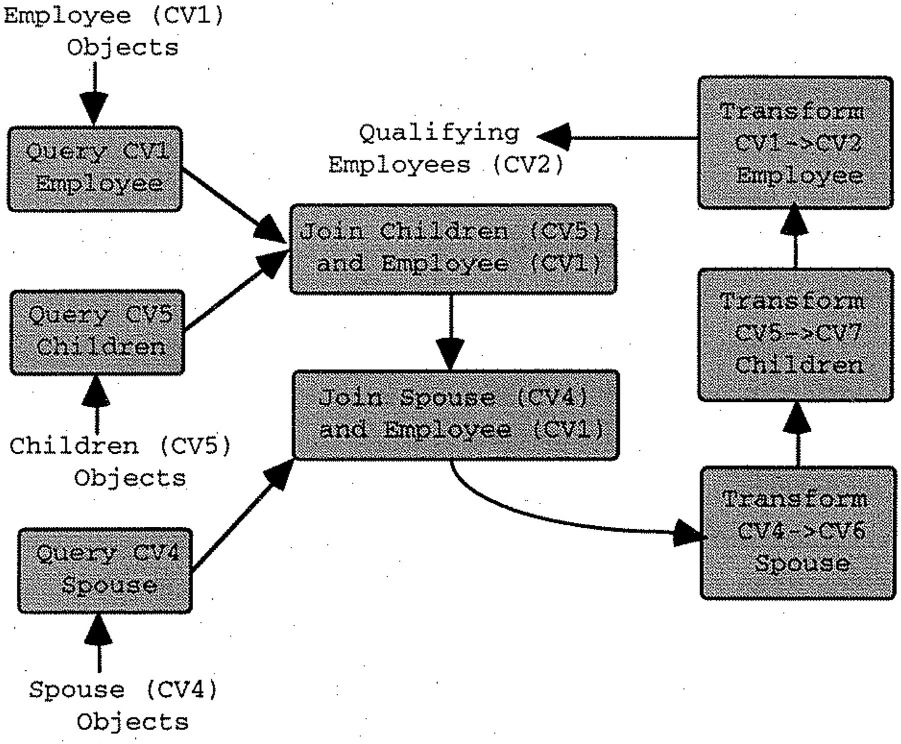

Employee (CV1) Objects
Query CV1 Employee
Qualifying Employees (CV2)
Transform CV1->CV2 Employee
Join Children (CV5) and Employee (CV1)
Query CV5 Children
Transform CV5->CV7 Children
Children (CV5) Objects
Join Spouse (CV4) and Employee (CV1)
Transform CV4->CV6 Spouse
Query CV4 Spouse
Spouse (CV4) Objects

Figure 6.5(b): Configuration#2, Restrict-Then-Join, Deferred-Transformation for a Query on Class "`Employee`"

(3) **Early Join, Early Transformation:** this is another variation of the first configuration as we apply the transformation of objects to the destination versions of their classes immediately following their restrict operation and prior to join operations involving other classes. We denote this configuration by 'Join-Then-Restrict, Early-Transformation', abbreviated to 'JN-RS-ET'. The application of this configuration to the `Employee` example is shown in Figure 6.5(c).

(4) **Early Restrict, Early Transformation:** this is a variation of the second configuration in which we apply the transformation of objects to the

destination versions of their classes prior to the join operations and immediately after their selection. We denote this configuration by 'Restrict-Then-Join, Early-Transformation', abbreviated to 'RS-JN-ET'. The application of this configuration to the `Employee` example is shown in Figure 6.5(d).

We differentiate between the four configurations in terms of: the cost of each operation involved in the query; the amount of redundant processing of shared objects; the degree of parallelism involved in executing query operations. We perceive that all configurations may benefit from pipeline and dataflow processing but with varying degrees of efficiency. Configurations 1 & 3 involve elongated join operations since they involve all (un-restricted) objects of the inner relation. Since a join operation normally takes more time to execute than a restrict operation and assuming that transformation processes come cheaper than joins, then we may expect the Join-Then-Restrict configurations to be slower than Restrict-Then-Join. Another aspect of Configurations 1 & 3 is that objects shared between several parent objects would undergo repeated transformations. This is because a copy of the shared object is merged with each of its parent objects prior to transformation. This results in redundant processing overhead taking place. Configurations 2 & 4 can execute in parallel the restrict operations on all participating classes in the query. This means that they can release more processors earlier than Configurations 1 & 3 and would have fewer number of nodes in any path throughout their processing graphs. Unlike Configuration 2, Configuration 4 can run the transformation operations of all classes simultaneously, while the former is forced to sequential execution otherwise a repeated join would be required following the transformation operations. Configurations 1 & 2 apply transformation operations only to the final qualifying objects, in contrast to Configurations 3 & 4 which transform objects that may still be bound to further restrictions throughout the query graph.

Predicting the exact performance of either configuration, however, would be rather difficult with query-graph nodes being served in parallel and sharing the same processors and with operations involved incurring variant workloads. In Chapter 8, we conduct simulation experiments, depicting pipeline and dataflow operation, to investigate the performance impact of processor allocation and sharing in those configurations.

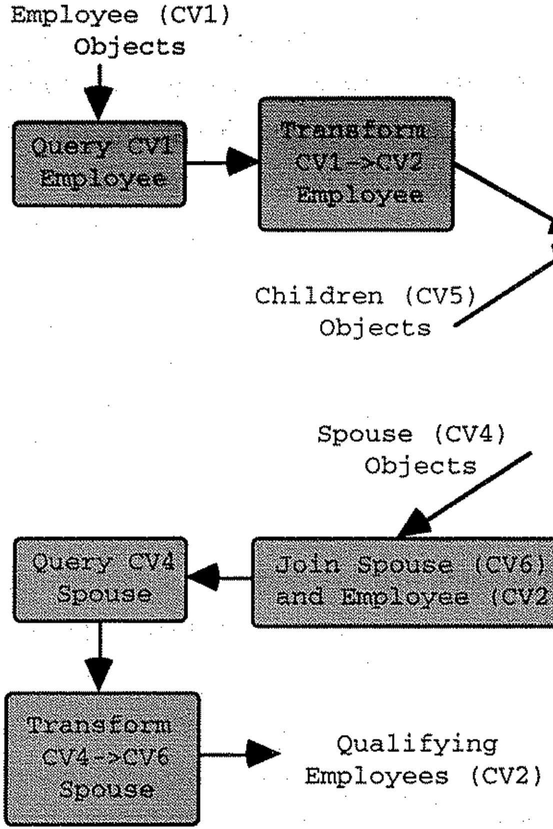

Employee (CV1) Objects
Query CV1 Employee
Transform CV1->CV2 Employee
Join Children (CV7) and Employee (CV2)
Children (CV5) Objects
Query CV5 Children
Spouse (CV4) Objects
Transform CV5->CV7 Children
Join Spouse (CV6) and Employee (CV2)
Query CV4 Spouse
Transform CV4->CV6 Spouse
Qualifying Employees (CV2)

Figure 6.5(c): Configuration#3, Join-Then-Restrict, Early-Transformation for a Query on Class "Employee"

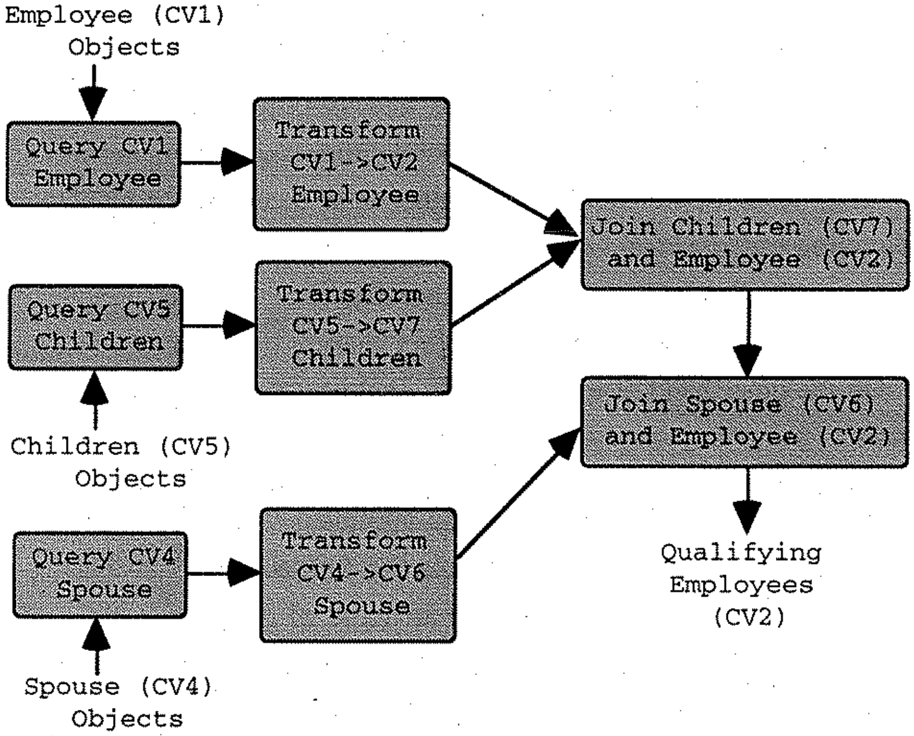

Employee (CV1) Objects
Query CV1 Employee
Transform CV1->CV2 Employee
Join Children (CV7) and Employee (CV2)
Query CV5 Children
Transform CV5->CV7 Children
Children (CV5) Objects
Join Spouse (CV6) and Employee (CV2)
Query CV4 Spouse
Transform CV4->CV6 Spouse
Qualifying Employees (CV2)
Spouse (CV4) Objects

Figure 6.5(d): Configuration#4, Restrict-Then-Join, Early-Transformation for a Query on Class "Employee"

It should be noted that path, node, and class-hierarchy parallelism mentioned in Chapter 2 can all benefit from the pipeline and dataflow processing proposed for simple, complex, and interrelated objects. This can be realized by creating pipelines to resolve user-query for each navigational path in the query graph, and executing those pipelines in parallel. Within each pipeline, processing nodes can be allocated to classes and subclasses relevant to each predicate.

6.5. Data Partitioning

One of the advantages of parallel processing over uniprocessor environments is the enhanced performance they provide when serving multiple users as each user can have separate processors allotted to his usage and thus avoids contention with others over the processing resources. One limiting factor to that advantage is the sharing of data, objects in our case, as users competing to lock a particular data element (be it an object, an object-page of a particular class-version, or all instances of a class) may be forced to wait until the element is freed. Fortunately, improved object sharing comes naturally within the class-versioning approach. We mentioned in Chapter 4 a key feature implied in the approach, that is partitioning database objects horizontally into groups according to their class-versions. Such feature provides three directions in which parallel processing can be applied. The first is to serve class-version queries in parallel, assuming enough processing nodes are available. This would speed-up the execution time of user-queries. The second direction is enabling multiple users to access objects of different class-versions simultaneously. This would have a remarkable impact on system performance specially if users specify class-versions of interest rather than accessing all objects of a class. The third direction emerges from confining locking to accessed class-versions (and making class-versions the largest lock granule) rather than to all versions of a class. Reducing the applied lock granule further improves object sharing between users and minimizes wait times. On the other hand, it imposes additional processing and storage overheads at the same time.

Vertical partitioning of objects is also applicable for class-version instances in our approach, but within the guidelines mentioned in Subsection 4.1.2. That is, if a destination attribute is derived from more than one attribute of the source class-version, then the partitions of all attributes involved in the derivation must be considered as implicitly requested by the user and should undergo the necessary transformations. The drawback of vertical partitioning, however, is the overhead of the join operations required for assembling objects from their partitions.

6.6. Process Taxonomy

Based on process configurations discussed earlier, we categorize processes participating in object manipulation into six modular types: the Class Server (CLSR); the Class-Version Query Resolver (CVQR); the Version-Collection Transformer (CLTR); the Class-Version Mapper (CVM); the Universal Query Server (UQS); the Object Merger (OM). Figure 6.6 illustrates each type of processes in its possible configurations.

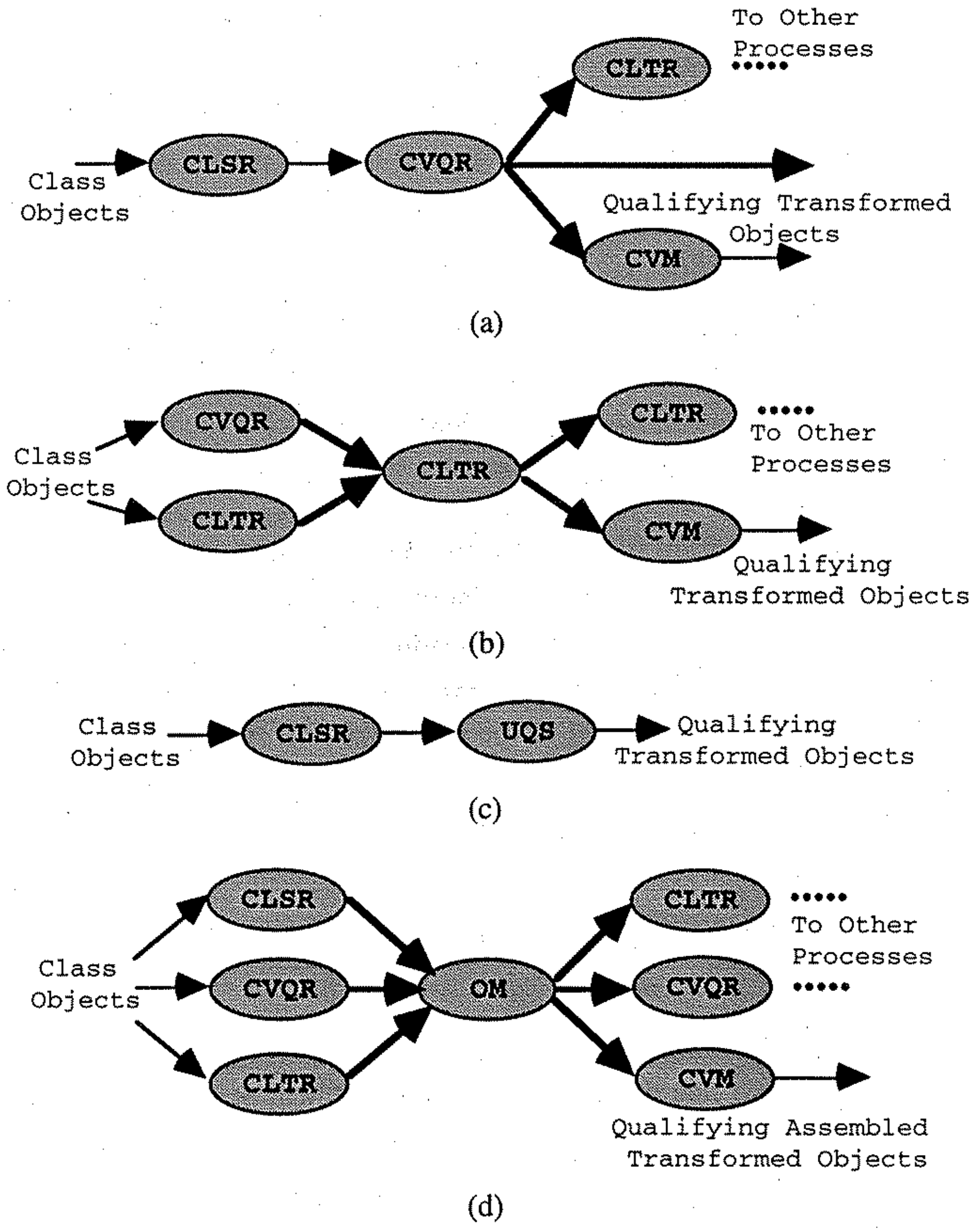


Figure 6.6: Alternative Process Interconnections
The bold arrows depict object flow to/from alternative process types

- **Class Server (CLSR)** forms the central server for class-versions of a particular class. It handles caching objects from the secondary storage, and passes them to subsequent processes serving user queries. More than one CLSR process can serve the same class, each managing one or more class-versions. This would improve query service time if a separate processor is allocated to each CLSR process and requests for object-pages are served simultaneously.

- **Class-Version Query Resolver (CVQR)** resolves queries aimed at a particular class-version. The CVQR receives objects from a CLSR process and checks query predicates against the objects. Qualifying objects are mapped in the CVQR to the version-collection that houses their class-version(s). Mapped objects are then passed on to the following process in the configuration.

- **Version-Collection Transformer (CLTR)** is responsible for the transformation of objects from one version-collection to another. Each CLTR process in a query configuration is dedicated to a particular set of transformers, either forward or backward. Transformed objects may be passed on to another CLTR or to a CVM process, depending on the query configuration.

- **Class-Version Mapper (CVM)** represents the final stage in processing user-query, aimed at a particular class-version. A CVM process takes objects, having the structure of the destination version-collection, from either CVQR or CLTR processes, and transforms them into the structure of the destination class-version specified in the user-query. It then applies the class-version constraints to the resultant objects attributes and passes objects that conform to the user. If query optimization considers condition (4) mentioned in Section 4.5, then the role of the CVM as constraint checker would be redundant since generated queries take into consideration attribute constraints of the destination class-version.

- **Universal Query Server (UQS)** groups the functionality of the later three types into one process. A UQS process will serve all class-versions of a particular class. In doing this, it will resolve the user queries aimed at those class-versions, perform the relevant handlers, and run the appropriate transformers throughout the transformation path leading to any destination version-collection of the same class. Constraint checks are run by the UQS if they are not handled in query optimization phase. UQS also maps the transformed objects from the destination version-collection to the user-specified class-version.

- **Object Merger (OM) joins constituent objects with their parent objects.** An OM is a binary operator that merges objects of two classes at a time. So, a complex class having more than one constituent class would require an OM stage for each of its constituent classes to assemble composite objects. The OM may be required to perform a sort on object-pages of either or both classes being joined, depending on the join predicate.

6.7. Summary

In this chapter, we discussed the potential of parallel processing in serving object-oriented databases within the context of the class-versioning approach. Query generation for different version-collections or class-versions of a class, from the submitted user-query is one aspect that can benefit from parallel execution of the preprocessor. By doing so, we can generate queries emerging from a particular query by running replicates of the preprocessor simultaneously on different processors. Such parallelism can prove to be advantageous for ad-hoc queries targeting large version-collection graphs. Resolving queries on simple and complex objects is another aspect of parallelism in which we proposed dataflow and pipeline processing in order to reduce the overhead of object transformations. We presented different process configurations for querying simple, complex, and inter-related classes. For simple objects, we proposed three process configurations distinguished by the way processes collaborate and the number of processing nodes required in the absence of multitasking. For complex and inter-related objects, we proposed four process configurations, that may integrate any of the simple object process configurations, and distinguished by the precedence of object join and transformation operators. We also discussed the advantages and limitations of each configuration. We commented on multitasking which enables processor sharing between several processes to improve processor utilization and overcoming limitation on the number of available processors. Another aspect of parallelism emerges from horizontal data partitioning which comes naturally by categorizing class instances by its versions, and vertical data partitioning which may be realized with the appropriate physical data model. Processing object partitions simultaneously on different processors can achieve better performance, but, as we hinted, vertical partitions may require expensive merge operations to assemble objects. Confining query access to partitions of targeted objects improves object sharing between users and permits parallel execution of multiple queries. Finally, we presented a classification of process types that may participate in object manipulation and realize its query configurations. We illustrated the alternative configurations in which each process type may fit.

Chapter 7
The MIMD Prototype

Throughout our investigation of parallelism inherent in the class-versioning approach, we outlined alternative process configurations for resolving user-queries. The approach, however, was proposed as a general criteria for supporting schema evolution in OODBMS and did not target any particular system. So, in order to evaluate and demonstrate the performance of those configurations, we recognized the need for a test platform realizing different process types and providing realistic measurements of their activities when resolving a user-query. Accordingly, we sought the implementation of a prototype that would give us a glimpse of the requirements and characteristics of each process type in a realistic processing environment, and that provides us with elementary estimates of overheads involved in manipulating objects[1]. The prototype adopts the first and third methods of querying simple objects mentioned in Section 6.3 and supports transformations with numeric and string operations. Despite the limitations of the prototype, it enabled us to practically envisage the requirements for resolving queries on complex objects based on understanding the capabilities and limitations of the environment and the characteristics of the implemented process types.

In this chapter, we describe the prototype implementation for object manipulation in a distributed parallel processing environment within the context of the class-versioning approach. We start the chapter with a description of the parallel processing environment from both the architectural and the operating system points of view. This is followed by a section describing the realization of processes that may take part in simple-object manipulation. In section three, we describe the supported process configurations and discuss their impact on system performance. Finally, we present concluding remarks about the prototype and the implementation environment.

7.1. The Parallel Processing Environment

Before we proceed with the presentation of the implementation environment, we would like to briefly summarise our experience with the IFS machine (reviewed in Chapter 2) as an example of SIMD architecture. We built a procedural interface between a host SUN machine and the IFS for our C++ extension described in Chapter 4, using the IFS

[1] Performance measurements of the prototype are dealt with in Chapter 8.

simulator SIERRA. We were then faced with some drawbacks of the IFS that would affect its performance in supporting object manipulation. First, the lexical token conversion for each non-numeric attribute required a translation function call from the host computer to the IFS for each value of such attribute fetched into the host, and another function call for writing it into the IFS. We perceived that such requirement would affect the performance of manipulating persistent objects as the IFS would not run in an autonomous mode but would rather impose additional overhead on the host. Another drawback of the IFS is that its transputers are mainly dedicated for controlling the operation of the search engines and the communication between different nodes of the machine. So, having the IFS perform object transformation would not be feasible. We add to these conclusions that being a SIMD machine, the IFS would only serve a single class-version query at a time.

With the experience we had from the IFS, and based on the nature of the sought object manipulation configurations, we were able to formalize the characteristics of the implementation environment that would deliver acceptable performance. These are: having a MIMD architecture; being capable of autonomous operation with minimal intervention from the host; supports multitasking operation; provides a flexible degree of parallelism for any process type. The benefits of those requirements, which are supported by our chosen environment, will be highlighted throughout the discussions that follow.

7.1.1. The Parallel Machine Architecture

The target hardware is a VMEbus based Motorola 68030 machine, with 12 IMP JT68030 cards. These are off the shelf processor cards each with a 20 MHz 68030 + 68882 and 4 megabytes of memory. One of these cards runs UNIX system V.2.2 which is our host operating system. In addition, the machine contains a few other cards which UNIX uses, including a graphics card that can be accessed from UNIX via X-Windows. The rest of the processor cards, which form a parallel processing pool attached to the host processor, run the Equus distributed parallel operating system [Byte 89, Equus]. Equus coexists with its UNIX host operating system and does not affect the running of normal UNIX programs. Only a small part of Equus, the application Launcher, actually runs on the UNIX processor card, while the Equus Kernel is distributed and runs (some 150 Kbytes of code) on each CPU in the parallel processing pool (see Figure 7.1).

7.1.2. Equus Application Structure

An Equus application is made up of "segments" that are separately compiled C modules, and the program is run in parallel by mapping these segments onto the available processors in the pool. The memory image of a segment running on a processor is called, in Equus terminology, an *incarnation* of the segment, which corresponds roughly to the concept of a process used in other systems. In other words, Equus incarnations are communicating sequential processes. A segment can be incarnated more than once, in highly parallel applications. The whole collection of incarnations that forms a running program is referred to as a *wave* . The UNIX host operating system supplies all disk filing services, text, and graphical output. It also launches the first, or primary, incarnation into the parallel pool. The Equus Kernel provides memory management and process scheduling on each parallel processor, as well as managing the making and breaking of communication links between incarnations. Equus is a multitasking system where more than one incarnation can run on a single processor and each incarnation contains separate tasks, or threads. It is also a multi-user system, and waves belonging to different users can share the pool of processors according to a particular scheme maintained by a system wave called the Pool Manager.

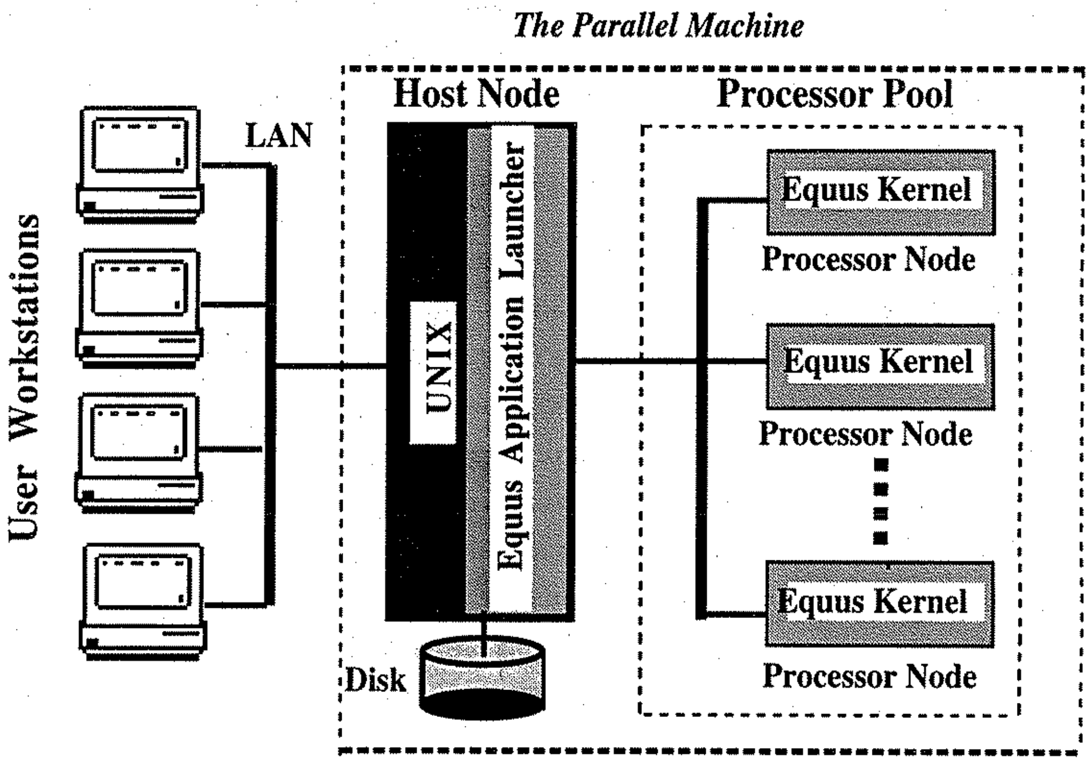


Figure 7.1: The Parallel Processing Environment of the Prototype

To run an application, we need to invoke the Launcher (which runs on the UNIX host card), and this contacts the Pool Manager (which runs in the pool) to find out how many pool CPUs we have been allotted and on which one to launch the first incarnation. The first segment is incarnated, and this primary incarnation is then responsible for incarnating the rest of the segments in the wave.

7.1.3. Communication Channels in Equus

A wave consists of a number of communicating sequential processes–incarnations, which communicate by sending messages to each other. These messages may contain data/or references. A reference is a software entity which may be either a stream, a port, an incarnation handle or a buffer handle. The program entities that support messages are streams and ports; a stream and a port together make up a one-way synchronous/asynchronous communication channel, the stream sending messages and the port receiving them. Only one incarnation can receive messages from a particular port, but many incarnations can send messages to the same port, and an incarnation can have many ports and streams attached to it. Having an incarnation handle gives control over an incarnation and is initially owned by the creator of that incarnation. An incarnation may possess a copy of its own incarnation handle,. allowing it to migrate or kill itself. Incarnation handles may be shared, allowing a degree of fault survivability. A buffer handle gives a remote incarnation temporary access to an area of the sending incarnation's address space to or from which the remote incarnation may copy data.

7.1.4. Equus Wave Generation

Waves distribute themselves around the pool of processors in a decentralized way; any incarnation can create a new child incarnation on another processor by using the *incarnate()* system call. The primary incarnation created by the Launcher spawns one or more child incarnations, which in turn spawn more, and so on until the whole wave is in place. A child incarnation need not be a copy of its parent, for the *incarnate()* system call specifies that a particular segment is to be incarnated on a particular processor. However, the parent does have controlling relationship to its children. The communication network between incarnations is constructed on the fly during program distribution and can be altered at will afterwards. When a parent incarnation terminates, all its children are automatically terminated by the system unless they have another live parent.

7.1.5. Migrating Incarnations

To migrate an incarnation between nodes, its parent uses the *migrate()* remote procedure call; Equus makes frequent use of such calls, in which a process calls a function situated on a remote processor. The parent incarnation can now continue un-hindered as the migration occurs in parallel. As far as the user program is concerned, this migration is wholly transparent, and all messages continue to arrive at the right places. This level of integrity is maintained at the system level by also giving the owner of an incarnation handle, which represents the ability to operate on the incarnation, the responsibility for tidying up the communication links. Supporting incarnation migration in such seamless way enables Equus to dynamically balance the load amongst the processing nodes.

7.2. Realizing Object Manipulation Processes

The prototype of object manipulation exploits, as we shall demonstrate throughout the remainder of this chapter, the parallel machine architecture and the features and facilities supported by the Equus operating system. We shall describe the implementation of process configurations for class-version queries in the following section. In this section, we elaborate on the implementation of different processes that may take part in handling user queries. In doing so, we build on the information provided in Section 6.6 and we highlight the relevant implementation aspects. We shall use Figure 7.2 to illustrate relevant details as we proceed.

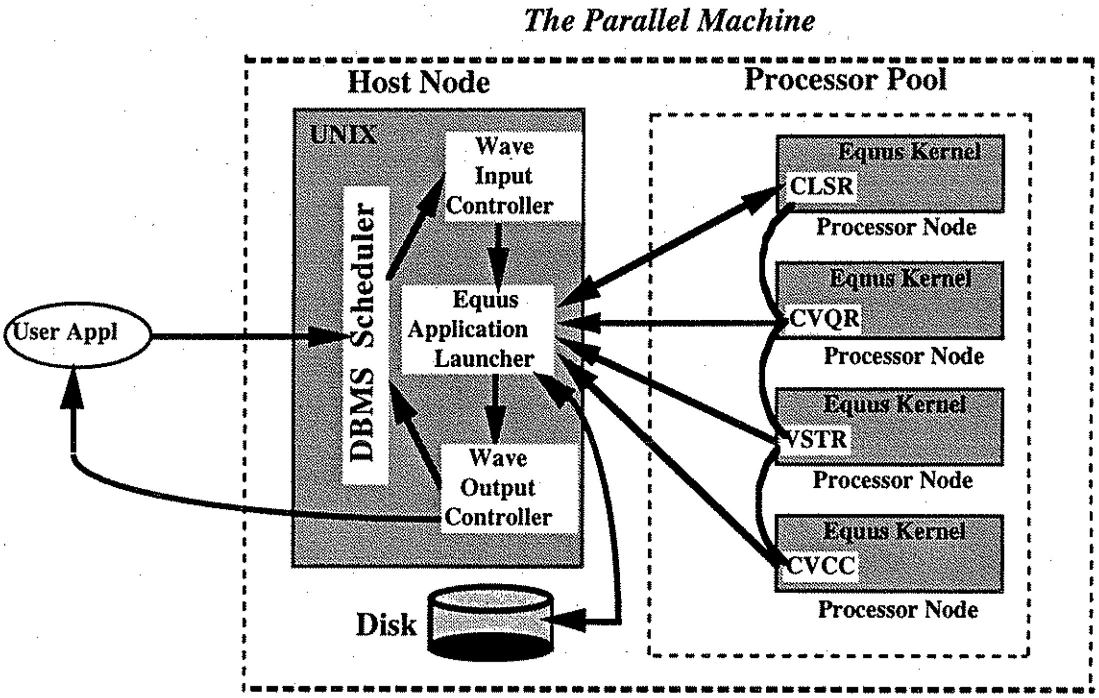


Figure 7.2: The Interconnection of Implementation Modules of the Prototype

7.2.1. The DBMS Scheduler

In the host of the parallel machine, we run a scheduling process that would normally be part of the DBMS. The DBMS Scheduler[2] launches a wave for each class in the database. The primary incarnation in each wave would be a CLSR serving the relevant class. Queries can then be forwarded from the user-workstation via a communication facility to the host. Upon receiving a query, the Scheduler would generate class-version queries and pass them on to the appropriate wave. In order to achieve that task, the Scheduler maintains a matrix marking the path between each version-collection and its immediate neighbours in the collection-graph. Creating and updating the matrix would take place as part of schema editing[3]. Each class-version query includes control information[4] to the CLSR specifying the type of processes to be incarnated, and how those processes are configured.

The communication between the wave incarnations and either the host or the user-application is not straight-forward and requires launching each class-wave from within a UNIX pipe having an input and an output control stages (see Figure 7.2). This is because incarnations under Equus are unable to use the UNIX interprocess communication facility directly. We require this sort of communication facility to enable the class-wave incarnations to receive commands, queries, and data from the host, and to send resulting objects and service statistics to the user-application. Otherwise, we would be forced to use temporary disk files to exchange information between processing nodes, the host, and the user. The UNIX pipe is run on the host side of the parallel machine. The input to the UNIX pipe is forwarded by the Scheduler while its output is forwarded back to either the Scheduler, the disk, or the user-workstation depending on the purpose of the output.

7.2.2. The Class Server (CLSR)

We create only one incarnation of this type for each class. The CLSR[5] is considered the primary incarnation of the wave serving a particular class. Any query related to that class will have its serving incarnations created within the same wave. The prototype maintains a set of control tables governing the manipulation of objects and their holding pages. Those tables are distributed among the CLSR incarnations serving all class-versions in the

[2] requires 56 Kbytes of host main storage

[3] Currently, the matrix is maintained manually as schema editing is not implemented.

[4] Currently, such information is embedded in the user-query sent to the Scheduler.

[5] requires about 64 Kbytes of node memory in addition to its object-cache that may take up to 3 Mbytes.

database. By doing so, part of the DBMS workload is delegated to the pool processors and more host processing resources can be dedicated to other tasks. One table category contains indices to pages belonging to each class-version in the database. At the initialization of the DBMS, each CLSR incarnation caches its own table and the object-pages of the class it represents. Another table category is the lock-queue which keeps a list of users waiting for a particular object-page and requesting a particular lock type. Such a table controls access to object-pages and sharing them between users.

A data page, in the prototype, has a header specifying: the object's class, class-version, and version-collection; page-lock type ('R' for read only, 'W' for write only); the objects count in the page (determines the available space on the page). Each class in the database has its own object-page file that keeps its objects. Objects are grouped in the pages according to their class-version. A control file is affiliated to each class that links object-pages to their class-versions, and class-versions to their version-collections. Each object has a header that specifies whether the object is active or deleted. To simplify the prototype, an object has a fixed size to avoid object migration between data-pages due to expansion in size. Also, intermediate class-versions and object migration between class-versions are not supported in the prototype. The CLSR performs the following tasks:

- ***Caching Object and Control Data:*** The CLSR caches object-pages and control information relevant to its class, from the secondary storage, into its local processor memory and commits them back to the secondary storage whenever requested or at system shut-down. Although the processor node offers only 4 Mbytes of main-storage, which limits the cache size, we know of other multiprocessor machines providing over 40 Mbytes of storage that may support larger caches.

- ***Creating Incarnations*** [6] ***Serving a User Query:*** The prototype currently relies on the user to weigh the processes of a query relatively to each other and to append those weights to the query. The user is also required to specify how processes are to be replicated in pool processors. Incarnations expected to have larger workloads would be allocated to nodes with incarnations of smaller workloads. At incarnation time, the CLSR launches processes according to user-specification on the available pool processors.

[6] The current Equus setting allows for a maximum of 12 incarnations per wave, but this setup can be altered to suit user-application.

- ***Terminating Incarnations Serving a User Query:*** The CLSR detects the end of the query in two ways; either by being notified by any incarnation that the query has been served and that no more object-pages are required; or if the query-serving incarnations demand more object-pages to process while there are no more to send.

- ***Exchanging Object Pages with Query-Resolver Incarnations:*** The CLSR sends object-pages to CVQR or UQS incarnations requesting data for processing. The CLSR gets back processed pages from those CVQRS and overlays them on its own copy if they were involved in a modification operation (i.e., update, delete, insert).

- ***Object Sharing Control:*** The CLSR grants access to a read-locked page for users seeking a retrieval operation only, and revokes user-requests for other operations. It also revokes user-requests for accessing a write-locked page. Users denied access to a locked-page are put in a queue for that page. The CLSR serves the page-queue whenever a CVQR or a UQS incarnation releases the page-lock.

7.2.3. The Class-Version Query Resolver (CVQR)

A CVQR incarnation performs the following activities:

- ***Resolving Class-Version Queries:*** It receives the relevant class-version query from the wave's CLSR and requests all object-pages of its related class-version from the wave's CLSR as no indices are maintained in the prototype, and applies the query to each object in the received page. A CVQR has a general-purpose query manipulation routine that is capable of serving queries of any class-version of any class. This simplifies the creation of CVQR segments which can be generated from a generic segment. Placing the handlers specific to class-versions need to be incorporated in the generic segment leading to a modular design of CVQRS. A query can request either of the following operations to be performed :

  (1) retrieve objects satisfying predicate conditions,
  (2) retrieve objects referenced by the user using `ObjectIDs`,
  (3) insert a new object,
  (4) delete objects satisfying predicate conditions,
  (5) delete objects referenced by the user using `ObjectIDs`,
  (6) replace objects with those submitted by the user (equivalent to update operation).

We note that operations (#2, #5) require amending the query with `ObjectIDs` to be searched for, while operation #6 requires the replacement objects to accompany the query, thus realizing the first method of object updating (see Section 4.3). Operations #2, #5, #6 thus do not include any predicate conditions, and only use submitted `object-Ids` to locate targeted objects in the database. Matching database objects are then retrieved, deleted, or replaced. The rest of the operations (i.e., #1 and #4) require predicate conditions to identify the qualifying database objects.

- ***Acquiring and Releasing Page-Locks:*** The CVQR also specifies the lock type that should be imposed on each object-page during processing. For the retrieval operations (#1 and #2), a read lock will be required and the CVQR asks the CLSR to enforce it. If the CLSR grants the lock requested, the CVQR gets the object-page and resumes its activity. Otherwise, it waits for acquiring the lock and suspends its other activities. For the other types of operations, an exclusive write lock will be required. Processed pages are returned to the CLSR to overlay their existing copies if the operation performed on the page is other than retrieval. This latter activity must be performed prior to releasing page locks.

- ***Applying Class-Version Handlers:*** Objects affected by the execution of the query are structured according to the definition of the relevant version-collection. Version-collection objects are then passed on to the following CLTR or CVM incarnations in the pipeline for further processing, if required. In order to reduce the communication overhead, the CVQR buffers the version-collection objects until the buffer becomes full, then flushes the buffer contents to the recipient.

7.2.4. The Version-Collection Transformer (CLTR)

The CLTR polls its input port to get object-pages of its relevant version-collection, and then applies the transformers and produces objects of a destination version-collection. Transformed objects are passed on in the pipeline to either the following CLTR or CVM incarnation, depending on the pipeline structure. A CLTR incarnation does not know where its input is coming from, or to where its output is forwarded. The routing of object-pages is determined solely by the CLSR at incarnation time, and is handled by the Equus during incarnation lifetime. This makes the design of the CLTR interface modular and independent on the pipeline configuration.

7.2.5. The Class-Version Mapper (CVM)

A CVM is notified in advance by the CLSR with the user's network-address to which it should forward the query-result. The CVM incarnations serving the same user-query are identical since they serve the same destination class-version.

7.2.6. The Universal Query Server (UQS)

The UQS segment is more complex in structure than the previous segment types as it embodies all class-versions' handlers and transformers of every version-collection of the class. It switches its activity from one class-version to another depending on the class-version of the object-page received from the CLSR incarnation.

7.3. Wave Configurations

In designing wave configurations for the prototype, we take into consideration load balancing across the pool processors and the alleviation of the impact of communication delays on processors utilization. Load balancing has different approaches which Pears and Gray [Pears 92] divide into two categories:

- Task balancing-based on partitioning data according to its distribution skewness as opposed to access frequency distribution motivated by parallelism enhancement.
- Resource balancing-concerned with balancing CPU cycles and I/O overhead across the machine nodes.

We take a different approach by considering replicating processes on the same processors allocated for a user-query. Pears and Gray also outline the purposes of replicating data as to increase its availability; improve access to data due to its locality; reduce local processing costs by structuring each replica differently. This approach can be beneficial to our implementation but may also cause degradation of performance as object replicas must be updated and migrated to different class-versions all together to have a coherent database. We did not consider object replication in the prototype where concurrency control in the CLSR manages only single copies of objects.

Before we discuss the supported wave configurations, we shall outline some implementation specific points. In the prototype, created incarnations, except for the CLSRS, serve only one query then terminate. This is because users are likely to have queries each with different incarnation configurations. So, leaving incarnations that have concluded their service in the node memory for future use will cause their accumulation and would exhaust the node memory. We add to this the limitations of our machine related to the available number of processor nodes and node memory. All segments that may be incarnated dynamically in a wave are compiled and linked to the primary segment of the wave, that is the CLSR. Modifications to the wave may be in the form of changes to a segment program; the introduction of a new segment; the removal of an existing segment. Such modifications require re-compiling the affected segment or wave and re-generating the wave's executable module that links all its modules. This procedure proved to be slow with Equus, even for a small number of simple segments. For example, an `Employee` wave, detailed in the next chapter, containing a CLSR, a CVQR, a CLTR and a CVM takes about 3 minutes to compile. In what follows, we shall elaborate on two modes of parallelism supported by the prototype based on pipelining and dataflow processing as well as duplicating universal servers.

7.3.1. Pipeline Processing Mode

The simplest pipeline configuration for this mode consists of a group of stage segments of similar or different types (or both), and contains only one incarnation of each of its segments. A simple pipeline starts with a CLSR followed by a CVQR and terminates with a CVM. In between those segments lies a set of CLTRs selected according to the transformation-path required by the user-query. An example of a simple pipeline is shown in Figure 7.3(a). Based on this conceptual pipeline configuration, other wave configurations can be formed. As we noted in Section 7.1, the pipeline stages can exhibit different processing workloads and an incarnation may be forced to wait for either the delivery of input data from its preceding stage or the readiness of its succeeding stage to accept its processed data. This situation is also bound to happen even with the provision of buffering at both the input and output of the segment specially when a large database is involved. In order to reduce the idle time of pool processors due to such imposed waits, we made use of the Equus support for multitasking on each pool processor and for timesharing its services between several incarnations. We also made use of the fact that the processor scheduler, in addition to allotting a time slot for each active incarnation loaded in its memory (as opposed to frozen segments which receive no service from the processor), will switch from an incarnation, waiting for some event to happen, to another incarnation requesting its service. So, running duplicate incarnations on the same processor (Figure 7.3b) or on different processors (Figure 7.3c) will reduce the idle time of the processor and improve its utilization. It should be noted that the number of duplicates of a pipeline is only restricted to a system-specified maximum limit and the available processor memory.

7.3.2. Grouped Processing Mode

In this mode, each class is served by a single CLSR segment and a single UQS segment. The UQS segment handles all class-versions of the class and combines all transformers and handlers contained in its collection-graph. It also resolves any query aimed at any class-version of the class and performs the necessary transformations leading to the destination class-version. The UQS segment can be incarnated as many times as required, on the same processor or on different processors, to improve system performance and increases processor utilization. The CLSR passes on a class-version query and the transformation path, provided by the Scheduler in the host, to the UQS incarnations. This distinguishes the grouped-processing mode from the pipeline mode in which the transformation path is implied in the pipeline structure. The UQS incarnations are then synchronized according to the CLSR instruction and activated to request object-pages in a similar way to the CVQRS.

This mode of operation becomes most useful when the number of available processors cannot accommodate the pipeline stages. Applications can also use this mode when the number of object-pages to be processed is small or when the user-query specifies a particular object or object-page to be processed. In the latter case, setting up a pipeline will be expensive as pipeline stages exchange objects grouped in pages rather than individually. This makes only one stage of a pipeline active at a time and the pipelining advantage will be lost.

7.4. Multiple User Operation

The prototype supports object-sharing between different user-queries by adopting a page-locking mechanism in which the user maintains his lock on the whole object-page, and releases the lock as soon as he finishes processing the required objects. The locks are not retained during the transformation of selected objects, and this ensures that the object-page becomes available to other users quickly. Comparing this mechanism with other implementations of locks at the object-level, we perceive that the latter requires more resources to maintain, including processing and storage, and may become impractical for very-large databases. This has been considered in commercial relational DBMS, for example the DB2[7] system that implements page-locking as the finest granularity of locks which can escalate up to TABLE and DBSPACE levels.

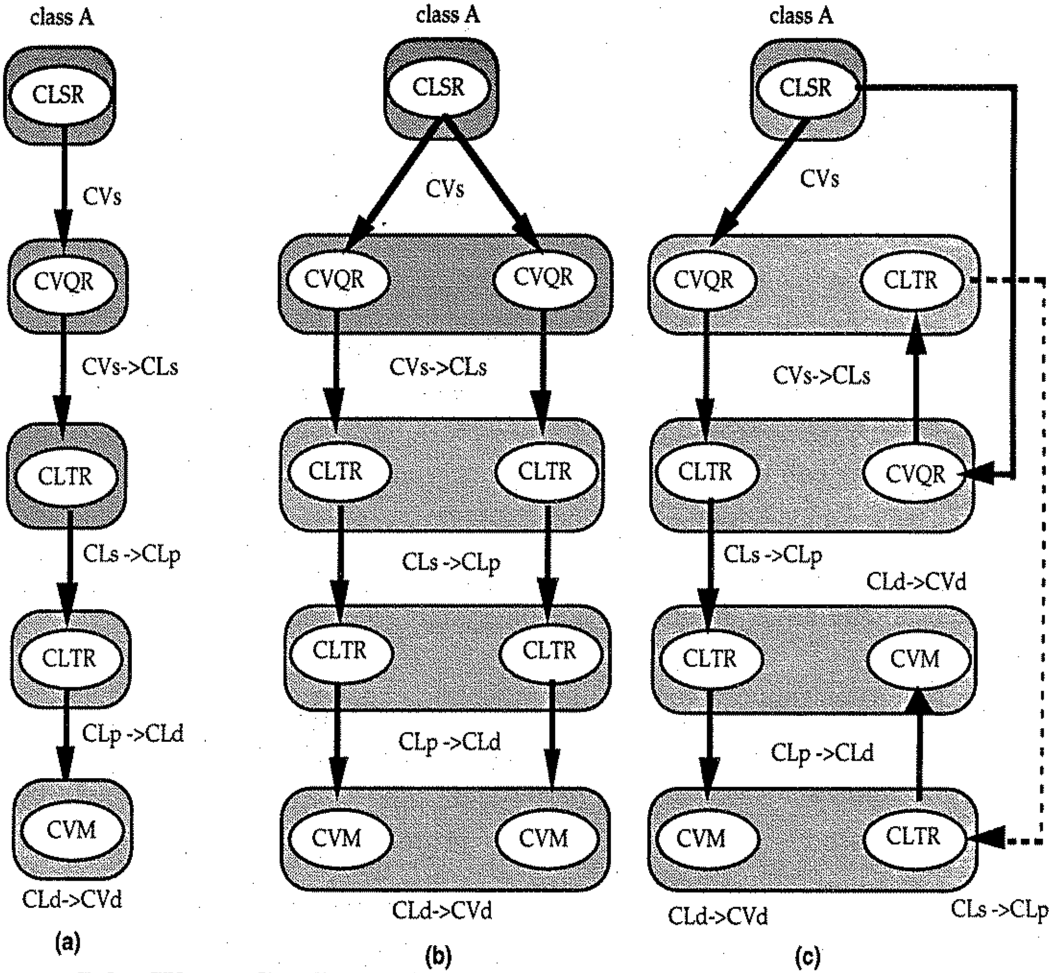


Figure 7.3: Wave Configurations containing (a) a single-stream pipeline, (b) a mirror-duplicate pipeline and (c) a shuffled-duplicate pipeline

CLs -> Source Version-Collection   CVs -> Source Class-Version
CLd -> Destination Version-Collection   CVd -> Destination Class-Version
CLp -> A Path Version-Collection

Another aspect of serving a multiple of users is to divide the processing power of the parallel machine between several users. We accomplish this in two ways. The first is to allow processes in the query configuration of each user, as described in Section 7.3, to be allocated to shared processors that support multi-tasking. By doing so, we are able to increase the processors utilization. The second is to allow query configurations of different users to share the same processors. Thus, we may serve multiple users with a limited number of available processors. We take into consideration here that not all processes require the processor service all the time, and that they may have to wait for some event to

[7] A proprietary of IBM.

occur, such as the arrival of data. In this case, the processor can serve active processes only and achieve better performance and utilization.

7.5. Summary

In this chapter, we presented a potential environment for supporting the class-versioning approach. The architecture of a multiprocessor machine was described and the main features and facilities offered by its distributed operating system (Equus) were highlighted. We elaborated on the design of the prototype of simple-object manipulation that acts as an experimental platform for evaluating different process configurations. Manipulating class-version objects is accomplished by a group of processes (or segments) of different functionalities as classified in Chapter 6. These processes are combined into a single wave the purpose of which is to resolve the user-query and to return, if required, the qualifying objects in accordance with the user-specified class-version definition. The launched wave may replicate (or incarnate) a particular process type to more than one copy, depending on both the user requirements and the available system resources.

The prototype supports the parallel processing configurations #1 and #3 discussed in Section 6.3 with the ability to replicate those configuration for each query. The latter feature is beneficial for horizontally partitioning class-version instances into groups which can be processed simultaneously. The prototype utilizes the processor sharing (i.e., multitasking) facility offered by Equus to distribute replicated processes amongst the processing nodes in such a way that achieves load balancing for the processors and improves their utilization. So, processing nodes need not stay idle waiting for some event to happen (such as receiving objects for manipulation) and they can be exploited by active processes instead. Processor sharing also compensates, to some extent, for the limited number of processors so that multi-user operation can be supported. In the case of transputer-based machines [Harp 89] employing the Occam programming language [INMOS 88], the allocation and the configuring of processing nodes need to be defined at compilation time. We find this feature rather limiting to our approach (but not undermining its applicability) for load balancing since the degree of parallelism given to a process (by duplicating it on more than one processor) cannot be specified on-the-fly and would require recompiling the configuring programs whenever there is a change of the expected process workloads.

The implementation of process types CVQR, CLTR, and CVM is modular in the sense of having a similar structure and I/O handling routines (each process of those types has a single input port and a single output stream). The query resolution in a CVQR is also performed by a generalized routine that is able to handle any class-version objects. In this way, we are able to create segments of each class-version by just placing its main functions (handlers, transformers, etc.) into a generic segment. Accordingly, creating a new class-version would require generating program code for each of the three process types to support manipulating its objects and would not affect existing processes. However, linking new processes to the class-wave in Equus would be required and it is rather slow as we hinted earlier in the chapter. The realization modularity also simplifies the operation of the CLSR which is responsible for creating process incarnations and forming them into the required configuration.

The parallel machine supports only a single disk that is managed by the host card. Accordingly, accessing the disk by pool processors may cause a bottleneck that slows down operations involving secondary storage. Having a disk attached to each processing node, however, would help in saving temporary pipeline results on secondary storage, specially when buffering is not feasible with limited main storage. It would also be advantageous in distributing class-pages throughout the disks so that each CLSR may access its relevant data from its local disk, and in speeding-up CLSR initialization.

We did not utilize the Equus load balancing facility as it relies on the accumulated workload of pool processors rather than on the activity of the currently executing processes and their role within their configurations (e.g., a process may be required to undertake a heavy load but for a limited duration in the query). So, with that facility, process migration may be initiated not for the benefit of the user. We, however, enabled the CLSR to distribute processes in the processor pool according to their pre-specified weighted workload (appended to the query by the user for simplicity but should be delegated to the Scheduler which has access to database control and statistical information to predict those weights).

We did not apply any of the existing benchmarks, such as the OO7 [Carey 93], to our prototype. The reason is that our implementation is not by any means an OODBMS but rather a set of functional modules of an OODBMS. Benchmarks, on the other hand, are normally set up to provide a profile of the performance of an existing DBMS. The OO7 benchmark was designed to test the speed of pointer traversals, the efficiency of updates to indexed and unindexed object attributes, and the performance of the query processor with different types of queries. The prototype, however, considers only simple objects and accordingly the targets of such benchmark are not applicable in our case.

In the following chapter, we shall present the results obtained from querying test data with this prototype. Based on benchmarking the basic operations of the prototype, we shall also investigate the performance of querying simple and complex objects under different system and database parameter settings using simulation.

Chapter 8
Experimenting With Prototype And Simulation

In this chapter, we differentiate between the process configurations, mentioned in Chapter 6, for manipulating simple and composite-objects in terms of performance and the required resources. We also explore the advantages of combining multitasking, process replication and associative techniques with pipeline and dataflow processing to improve query service time and processor utilization, and to reduce the required processing resources. We demonstrate the effectiveness of our proposed load balancing strategy mentioned in Chapter 7.

The prototype presented in Chapter 7 highlighted some implementation considerations in a potential environment, and provided us with basic estimates of processing and communication overheads. In this chapter, we review part of the performance measurements provided by the prototype which relate to our above mentioned objectives. We also adopt simulation for building models of different process configurations serving queries for simple and composite-objects. In building the simulation models, we take into consideration the characteristics of the prototype and its environment as guidelines. In running those models, we improvise processing and communication overheads to investigate system performance under a variation of system and database parameters.

We start this chapter with a description of our experimentation with the prototype and the benchmarks conducted on it. We demonstrate the impact of communication overhead and processor sharing on query service time. This is followed by an introductory section about simulation modelling and the simulation language used. In section three, we describe the basic models involved in the simulation experiments. In section four, we present the considerations and assumptions of the simulation. In section five, we demonstrate the effects of varying database and system parameters on query service time and processor utilization when manipulating simple and composite-objects. We also compare the performance of different process configurations and study the benefits of combining different processing strategies. We end this chapter with concluding remarks about our achievements.

8.1. Experimenting with the Prototype

The purpose of this effort is to measure the duration of basic activities of the prototype, and to evaluate its performance in serving queries using synthetic data for different process configurations. In the conducted experiments, we consider the simple class `Employee` with two class-versions (CV_1 and CV_2) each of which belongs to a separate version-collection (CL_1 and CL_2 respectively as in Figure 8.1). The attributes of each class-version and version-collection are depicted in Table 8.1, while class-version constraints are:

```
4000.0 ≤ Salary_£ ≤ 8000.0 for CV1;
6000.0 ≤ Salary_$ ≤ 12000.0 for CV2.
```

The definition of each version-collection matches that of its respective class-version. So, class-version handlers are not applied in this example. The transformers relating the attributes of both version-collections are shown in Table 8.2. Objects were generated for CV_1 with 40 bytes as object size and `Salary_£` randomly generated with uniform distribution of values between 4000 and 8000. The values of `Name` were cyclically generated from the alphabet, while `Age` was randomly generated with uniform distribution of values between 20 and 50. We executed several queries of different complexity on generated data to verify the operation of the prototype. Here, we present the results obtained from running two simple queries on CV_1 and delivering the result at CV_2. We choose those queries since the number of their servicing processes, alongside their replicas, can be accommodated by the available pool processors[1]. Running complex queries on the prototype, however, required sharing pool processors by query processes and did not enable us to study different process configurations.

```
Select * From Employee(CV1) where Salary_£ ≤ 8000
```

(all objects qualify; 49152 `Employees`);

```
Select * From Employee(CV1) where Salary_£ ≤ 6000
```

(around 50% of objects qualify; 24607 `Employees`).

Both queries were served by two basic process configurations (shown in Figure 8.1) which were replicated in different ways the details of which are listed in Table 8.3. In Table 8.3, we list the allocation of processing nodes to processes and their replicas in different configurations. The first column indicates the convention used for the

[1] Only 9 processors were installed in the parallel machine.

corresponding configuration. We shall refer to this table again when we discuss simulation later on in this chapter.

Timing different activities involved in resolving a user query was done by invoking the Equus time-record function to determine the elapsed time in performing each activity. Since the Equus kernel resident on each processor node handles timer interrupts at a rate of 60 Hz, the recorded time resolution is in the order of 17 milliseconds. To reduce the activity time indeterminacy per object, we ran each activity 1000,000 times[2] and isolated the iteration loop delay to get a more precise measurement. The CLSR activities were measured as follows: 8 milliseconds to send an object-page to a CVQR or UQS; 5 milliseconds to release locks on an object-page; 12 milliseconds to overlay a modified object-page (of size 32 Kbyte). The CVQR serves a 32 Kbyte object-page in approximately 285 milliseconds, while CLTR, CVM and UQS take 325, 265, 965 milliseconds respectively if all objects in the page qualify. The cost of a process or operation related to object-page size may be scaled in proportion to other page sizes. The CVQR takes 3 milliseconds to issue a request to the CLSR to either send an object-page or to release locks on an object-page.

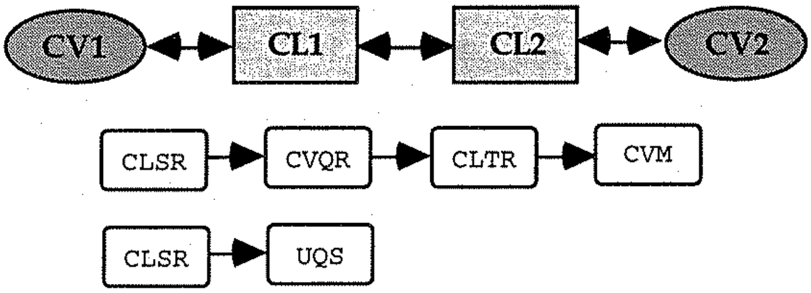


Figure 8.1: The Collection-Graph and Alternative Basic Query Configurations

CV_1 & CL_1	CV_2 & CL_2
Name	First_name
	Middle_name
	SurName
Age	Date_of_birth
Salary_£	Salary_$

Table 8.1: Class-Version Attributes

We assume Age is updated at the beginning of each year.

[2] This excludes incarnation creation time.

In Table 8.4, we list the results of running the first query (Salary_£ ≤ 8000) with different process configurations, each of which depicts a particular pattern for process replication and processor sharing. The wave involved in each run employed a combination of synchronous and asynchronous inter-process communication (we shall elaborate on this choice in Section 8.3). From the table, we notice the superiority of the pipeline mode (involving CVQR) compared to SIMD mode (involving UQS). In either case, replicating processes on different or on the same processors result in improved performance but not by the same factor of increasing processing power. For example, 2UQS-3CPU performs around twice (not triple) as better as a 1UQS-1CPU. This is due to the idle time which a process (UQS in this case) exhibits while waiting for an object-page from its preceding process (the CLSR in this case). However, replicating processes on the same processors improves wave performance as demonstrated by 2CVQR-4CPU$^{¢}$ which uses a similar number of processors as Single CVQR configuration and replicates its processes twice. The former configuration takes around 92% of the time required by the latter to serve the same query. The speed-up achieved by replicating wave processes in this experiment may seem insignificant. We relate this to the fact that the difference between page-processing-time of process types is small. Widening that difference, as we shall see in the simulation experiments later on, reveals more improvement with replication. Another aspect of replicating processes is that balancing the loads of participating processors achieves better performance than that of processors with unbalanced loads. This is evident with the second, third and fourth configurations in Table 8.4. Sorting the processes in a descending loading order as CLTR, CVQR, CVM (and weighted as 3, 2, 1 respectively), we notice that the fourth configuration performs best, giving a weighted loads of 4-4-4 for processors CPU#2, #3, #4 respectively. This is followed by the third configuration of loads 3-5-4, and the second configuration at 4-6-2.

Propagating any message between pool processors incurs fixed control overhead which is independent on message size. So, the larger the size of the message, the less the weight of that overhead. This is reflected in the results of Table 8.5 for service time of the above-mentioned queries with different object-page sizes and using Single CVQR configuration. From the table, we see that with 64-Kbyte page size we achieve 79% and 58% of the service time for 4-Kbyte page size, with all objects and half the objects qualifying respectively. Using either synchronous or asynchronous communication mode solely within the prototype revealed close results to those obtained with mixed communication mode but with slight superiority showing for the asynchronous mode over the rest. Regrettably, there are situations in which asynchronous mode dictates the availability of large buffer sizes to hold messages (intermediate results) between wave processes. A typical situation will be discussed in Subsection 8.5.2.

CL_2 -> CL_1 (Backward Transformers of CL_2)	CL_1 -> CL_2 (Forward Transformers of CL_1)
`Name=First_name||Middle_name||Surname`	`First_name=select_string(Name,1)`[3]
	`MiddleName=select_string(Name,2)`
	`SurName=select_string(Name,3)`
`Age=Current_year_date-Date_of_birth`	`DateOfBirth=CurrentYear_date-Age`
`Salary_£=Salary_$/1.5`	`Salary_$=Salary_£*1.5`

Table 8.2: The Version-Collection Transformers

Query Configuration	Process Types	Replication Factor	CPU #1 [CLSR]	CPU #2	CPU #3	CPU #4
2CVQR-4CPU[†]	CVQR-CLTR-CVM	2	X X	CVQR CVQR	CLTR CLTR	CVM CVM
2CVQR-4CPU[¢]	CVQR-CLTR-CVM	2	X X	CVQR CVQR	CLTR CVM	CVM CLTR
2CVQR-4CPU[§]	CVQR-CLTR-CVM	2	X X	CVQR CVM	CLTR CVQR	CVM CLTR
Single CVQR (1CVQR-4CPU)	CVQR-CLTR-CVM	1	X	CVQR	CLTR	CVM
3UQS-4CPU	UQS	3	X X X	UQS X X	X UQS X	X X UQS
2UQS-3CPU	UQS	2	X X	UQS X	X UQS	X X
Single UQS (1UQS-2CPU)	UQS	1	X	UQS	X	X
1UQS-1CPU	UQS	1	UQS	X	X	X

Table 8.3: Simple-Object Query Configurations[4,5]

[3] The `select-string()` is a user-defined string manipulation function that returns a sub-string located (according to the second operand, e.g., 1st, 2nd, 3rd etc.) in the attribute (referenced by the first operand) with no embedded blank characters.

[4] Larger process configurations (7 or more) are replications of the *Single* configurations on different processors.

[5] X indicates an unused CPU.

8.2. Simulation Modelling

Simulation is a recognized tool for predicting the performance and behaviour of implemented as well as under-development systems. The required investigation may involve new algorithms, techniques, system services, or hardware structure and configurations. Simulation may play a crucial role in modifying, enhancing, or even rejecting the element under scrutiny prior to launching its realization phases. Towards the implementation of the class-versioning approach, we used simulation techniques to model the prototype environment. We produced three models that form the basic test platforms for different process configurations mentioned in Chapter 6. In building those models, we made use of the results of both the system benchmark related to its basic activities, and our own experimentation with the system. The models have been verified and then used to evaluate different process configurations for queries with different system and database parameters.

Processor Configuration	Average Pipeline Service Time (ms)
Single CVQR	21000
2CVQR-4CPU[†]	20814
2CVQR-4CPU[§]	19923
2CVQR-4CPU[¢]	19208
2CVQR-7CPU	11566
1UQS-1CPU	60667
Single UQS	60667
2UQS-3CPU	30417
3UQS-4CPU	20517

Table 8.4: Process Replication and Multitasking Results[6]

[6] for 60 object-pages with 32-Kbyte page size and all objects qualify.

Page Size (Kbytes)	Page Count	Salary_£<=6000	Salary_£<=8000
4	480	18481	26393
8	240	13550	23015
16	120	11555	21650
32	60	10883	21000
64	30	10779	20916

Table 8.5: Communication Overhead Effect on Query Service Time[7]

The Simulation Tool

We used a discrete-event simulation language called **smpl** [MacDougall 87] which is a functional extension of a general-purpose programming language called the **host** language. A **smpl** simulation model is implemented as a host language program, **main()** in our case for C-language, and simulation operations are performed via calls on the functions of the simulation subsystem. This approach to discrete-event simulation provides a simulation capability suitable for small-to-medium scale models as opposed to process-simulation languages which are preferable for large-scale models. The **smpl** simulation subsystem is part of a simulation environment that provides additional tools, including debugging, data collection and plotting, and simulation output functions, together with an interactive interface to simulation model execution.

8.3. The Basic Simulation Models

We assume that message-passing is the employed interprocess communication method between processes serving a user-query. We built three models each of which deals with a specific message-passing method. The first depicts the synchronous message-passing operation where a process waits for its message to arrive at its destination before it resumes its operation. The second deals with the asynchronous operation where a process invokes the transmission of the message but does not wait for the communication to conclude. Instead, the process resumes its activities immediately as long as the number of pending messages in the wave does not exceed the threshold value set by the operating system. In case the latter value is exceeded, any process sending an asynchronous message within the wave will be blocked until the number of pending messages falls below the operating system's threshold value. The third model combines both modes of communication and

[7] for `Single CVQR` configuration with mixed synchronous and asynchronous inter-process communication.

specifies either mode for a process depending on its type and on that of the message destination.

All models consider multitasking to be supported by the operating system. In this respect, round-robin CPU scheduling is employed such that each CPU is cyclically allocated to the process requests in the CPU queue. Each request, in turn, receives CPU service for a fixed quantum of time (or for the time remaining in its current execution interval, if smaller than a quantum). A process that requires an I/O operation, message-passing in our case, will leave its CPU queue until its communication activity is concluded and then returns to the CPU queue if it requires further processing. The models are parameter-driven and can represent different process configurations for more than one query in the same run.

The behaviour of each process in either model is determined by the process position in the pipeline. For example, any process communicating with a CLSR would request an object-page from it, lock the page, and then request the CLSR to release its lock on that page. However, if the process is communicating with a preceding process other than CLSR, then all it has to do is wait for a transmission of object-pages from its preceding stage in the pipeline. Figure 6.6 illustrates the possible interconnections between process types and indicates process types that may communicate with CLSR. A unique feature of the OM type is that it may be dealing with a CLSR (to get component-object pages) and be at the same time connected to a non-CLSR (to get composite-object pages). Though we did not implement the OM type in the prototype, modelling its behaviour was possible by comparing its functionality to that of the implemented process types. In all simulation models, we deal with those behavioural variations to produce a typical process performance.

8.3.1. The Synchronous (S) - Model

In this model, all processes adopt synchronous communication. In Figure 8.2, we illustrate an example of a class-server process (a CLSR) that establishes a wave to serve user-queries related to its class. It creates m-pipelines, some or each of which may be related to specific class-version and may be affiliated to the same or different user-queries. Each pipeline is composed of a CVQR, a CLTR and a CVM processes shown in dark blocks (and so labelled) and are connected with thick lines according to their logical order in the pipeline. Other examples may have UQS and OM processes. The communication between processes is handled by the triangular blocks (labelled COM), each of which serves a specific pair of processes. In the model operation, when a COM block service is requested by the sender, the latter is blocked until the COM delivers the message to the receiver. If the receiver is busy, the COM waits and keeps the sender blocked until the receiver is free. Then the COM block applies the appropriate communication delay and releases the sender. In this way, the synchronous mode of operation is simulated. The COM$_0$ block is reserved for CLSR-to-CVQR (or to UQS) command and object-page delivery, where COM$_1$ block takes care of CVQR (or UQS) -to- CLSR delivery of requests and processed object-pages that should overlay the CLSR's existing copy. The exchange of processed object-pages and commands between the rest of the pipeline processes is handled by the remaining COM blocks labelled 2 and above.

Now we look at the processors side of the model where a set of n-CPUS exist, each of which may be allocated to more than one process at a time, and thus should serve them according to the multitasking scheme supported by the operating system. To demonstrate this situation in the model diagram, we chose to connect each process to its specified CPU via two imaginary buses, the upper one delivers the process's service request to the particular CPU-queue, and the lower returns control of the simulation process to the process which then decides what activity is to be performed next (either a message-passing or the termination of its activities). The round-robin processor scheduling is implemented in the model as follows: each service-request in the CPU-queue is attended by the CPU for a specified *quantum* of time (or the remainder of requested service time if it is less than a quantum) after which the service-request is enqueued again if it requires more quantums or discharged from the queue if it has received the full service required. We ignored the processing overhead of multitasking as well as that of communication invocation since we had no estimates of either of them, and we assumed they require no CPU service. We ignored the overhead of delivering query-results to the user and confined our model to the parallel-machine internal activities.

To summarize how a process performs in the model, each process receives a message to be processed from an input COM block, frees that block, requests a CPU service and waits for its completion, then requests communication service of an output COM block (except for the last process in a pipeline) and waits for the conclusion of transmission, and finally makes itself available for any further requests by a COM block.

As far as running the model is concerned, each of the activities explained so far is represented by an event that is to have a preset invocation time and duration which in effect advances the model's execution clock accordingly.

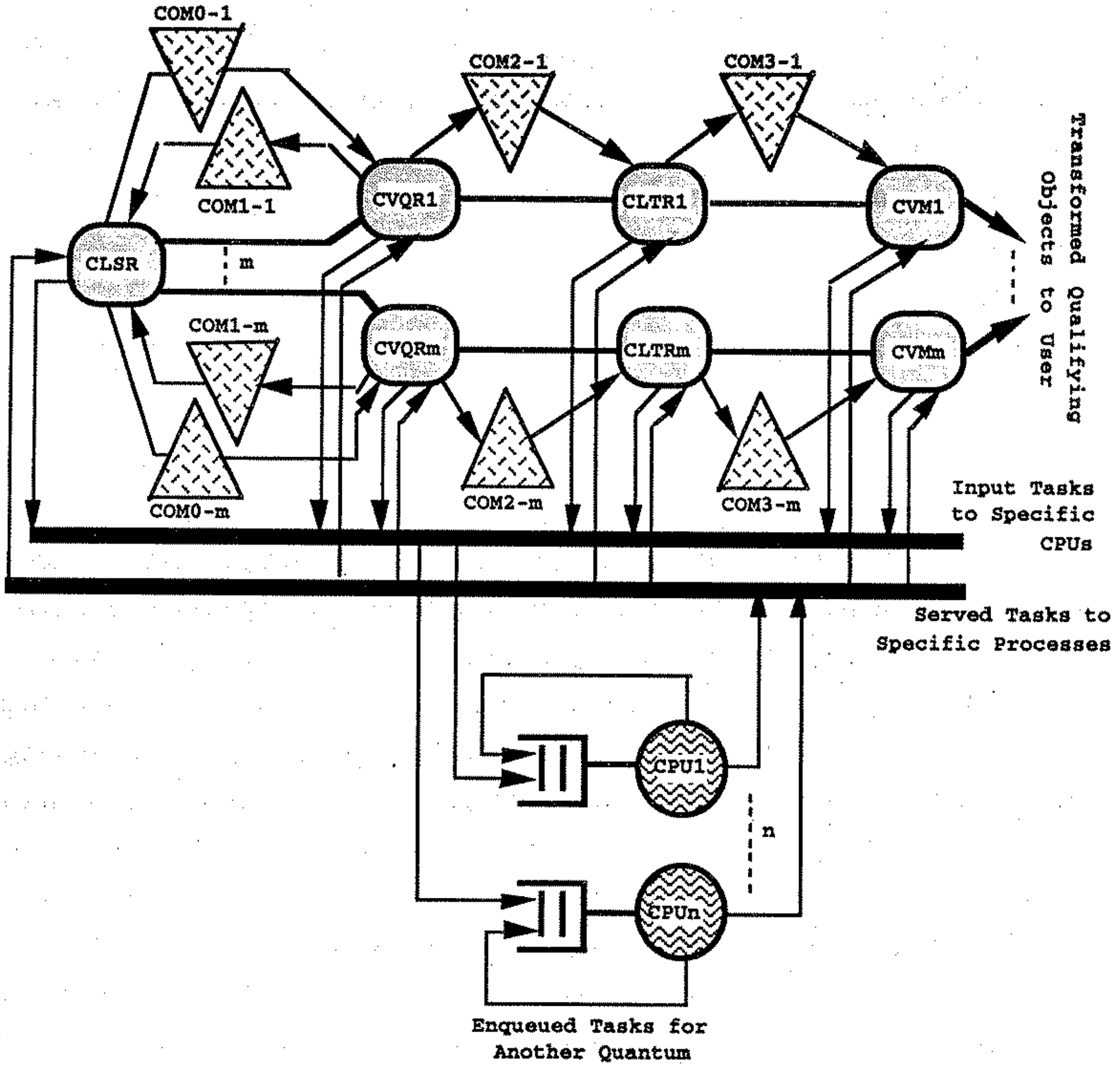

COM0-1
COM2-1
COM3-1
COM1-1
CVQR1
CLTR1
CVM1
CLSR
m
COM1-m
CVQRm
CLTRm
CVMm
COM0-m
COM2-m
COM3-m
Transformed Qualifying Objects to User
Input Tasks to Specific CPUs
Served Tasks to Specific Processes
CPU1
n
CPUn
Enqueued Tasks for Another Quantum

Figure 8.2: An Example of the Synchronous-Model

8.3.2. The Asynchronous (A) - Model

This model is a variation of the S-model where all message-passing between processes is handled by the operating system's asynchronous communication facility (see Figure 8.3). In this mode of operation, the operating system specifies the maximum number of messages that may be pending for transmission at a time within each wave, while each process in the wave defines another buffer threshold for its messages. If either threshold is exceeded, the process involved is blocked until the relevant number of pending messages falls below the threshold. To simulate this operation, we removed the COM blocks connecting process couples, and replaced them with a centralized serving facility of k similar COM servers for each wave (or class of objects).

A process requiring a message-passing reserves a COM server, and if one is available, it is allocated for the process, otherwise the process is blocked and queued. Once the process is granted a COM server, it is released and allowed to resume other activities if it has any. The reserved COM server, on the other hand, is not released until the destination process is ready to receive the message in which case the COM server applies the appropriate communication delay and makes itself available afterwards. As soon as the delay expires, the destination process starts processing the message.

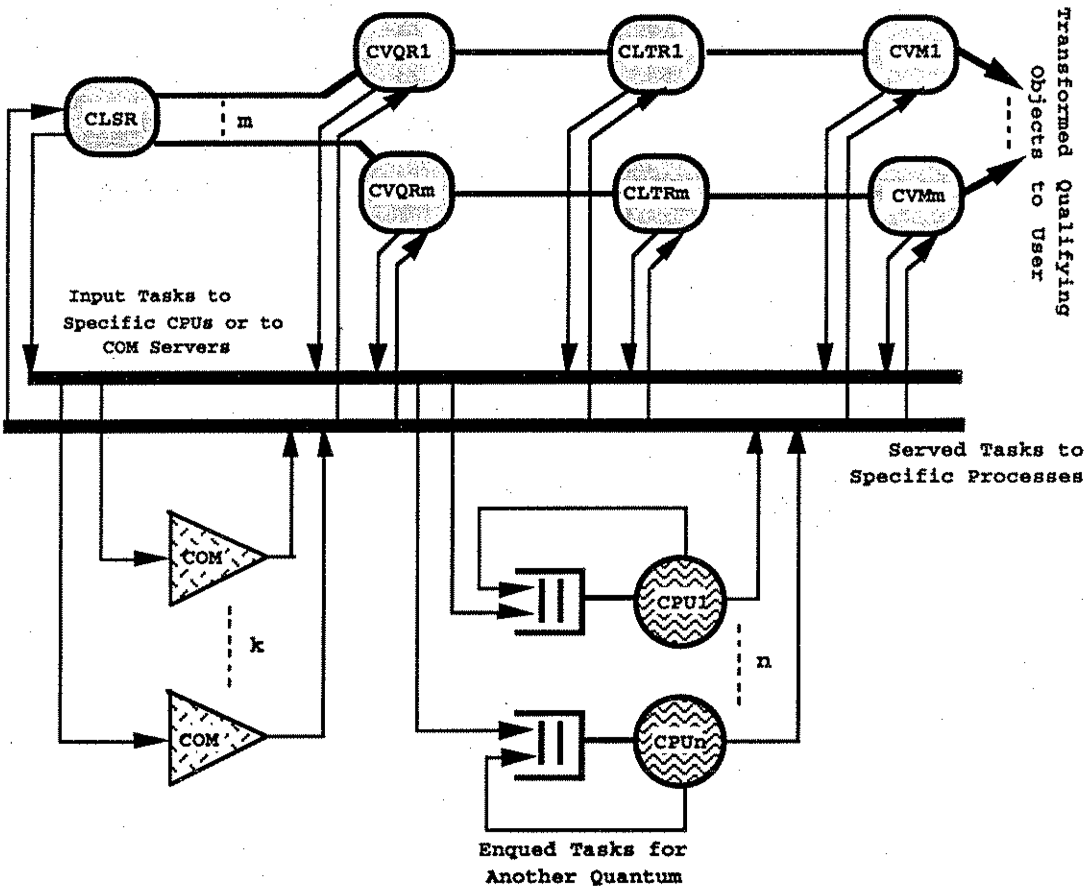

CVQR1
CLTR1
CVM1
CLSR
m
CVQRm
CLTRm
CVMm
Transformed Qualifying Objects to User
Input Tasks to Specific CPUs or to COM Servers
Served Tasks to Specific Processes
COM
k
COM
CPU1
n
CPUn
Enqued Tasks for Another Quantum

Figure 8.3: An Example of the Asynchronous-Model

8.3.3. The Synchronous/Asynchronous (S/A) - Model

This model combines the features of the previous two models (see Figure 8.4). First, it applies asynchronous message-passing only from the CLSR to other processes in its wave. Thus, the CLSR is able to serve pending requests once it initiates the transmission of each message and it does not have to wait for the communication to conclude. Second, the model uses synchronous communication between the other processes in the wave and thus removes the need for buffering output messages.

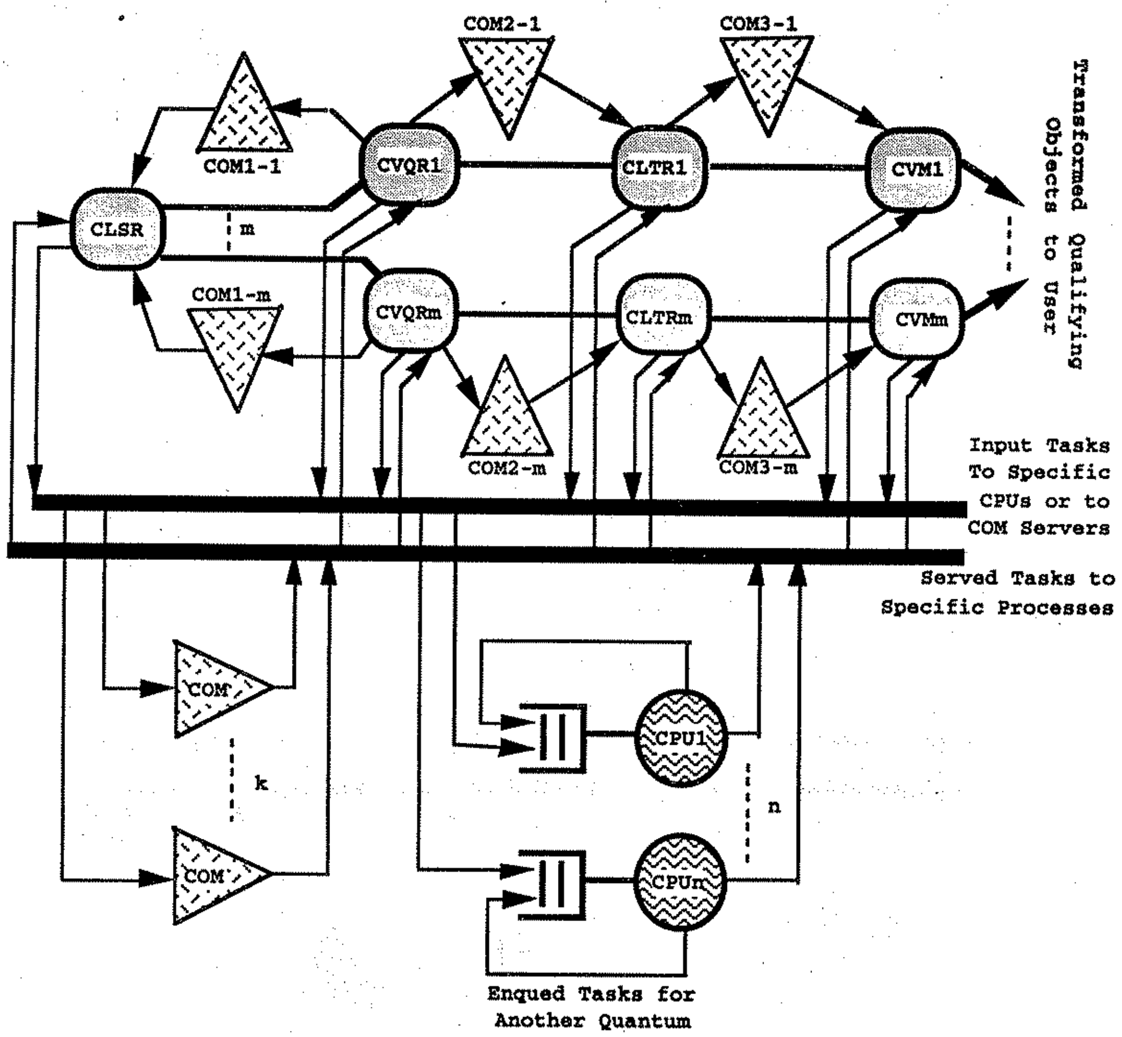

COM2-1
COM3-1
COM1-1
CVQR1
CLTR1
CVM1
Transformed Qualifying Objects to User
CLSR
m
COM1-m
CVQRm
CLTRm
CVMm
COM2-m
COM3-m
Input Tasks To Specific CPUs or to COM Servers
Served Tasks to Specific Processes
COM
k
COM
CPU1
n
CPUn
Enqued Tasks for Another Quantum

Figure 8.4: **An Example of the Synch/Asynch-Model**

8.4. Modelling Considerations and Assumptions

In building the simulation models, we took into consideration the ability to set and alter the following parameters:

a) Application and database characteristics;

- Object size.
- Object-page size.
- The number of object-pages in the database.
- Process configuration.
- Allocation of processors to processes .
- Query hit ratio.
- The number of composite-objects sharing a component or a referenced object.

b) System-related parameters;
- Maximum permitted number of pending asynchronous messages.
- Page-processing-time.
- Inter-process communication cost.
- Mode of communication between processes (synchronous, asynchronous).
- Number of CPUs in processor pool.

Unless otherwise specified, we assume that an experiment is conducted with the default settings:
- Sixty 32-Kbyte object-pages using page-processing-time specifications of Section 8.1.
- All class objects qualify for the query.
- The first CPU (#1) in the wave runs the CLSR process, and may be shared amongst other processes in the configuration.
- Query service time is measured in milliseconds and excludes process creation.
- 25 millisecond for CPU-quantum[8].

Function	no data bytes	1 Kbytes	32 Kbytes	64 Kbytes
Synchronous Invocation	2.863	3.377	14.57	26.1
Asynchronous Invocation	3.86	4.43	14.783	25.493

Table 8.6: Message Invocation Benchmark

To estimate the overhead of communication between incarnations serving user-queries, we considered the benchmark run on the machine by Kindberg [Kindberg 90][9]. Table 8.6 lists some results of the benchmark which are of interest to our implementation (all measurements are in milliseconds). We produced an approximate invocation measurement equation for both types ignoring the slight difference between themselves as our modelling accuracy is in the order of milliseconds. Figure 8.5 plots the respective chart estimating message-passing time for message sizes up to 64 Kbytes.

[8] Equus setting is unknown to us. Accordingly, we conducted simulation runs with different cpu-quantum values ranging from 1 to 50 milliseconds. The variation of the query service times obtained was negligible. The chosen setting is in the centre of the tested range and seems reasonable.

[9] A Ph.D. dissertation introducing the Equus distributed operating system and its implementation using the same parallel machine of our prototype.

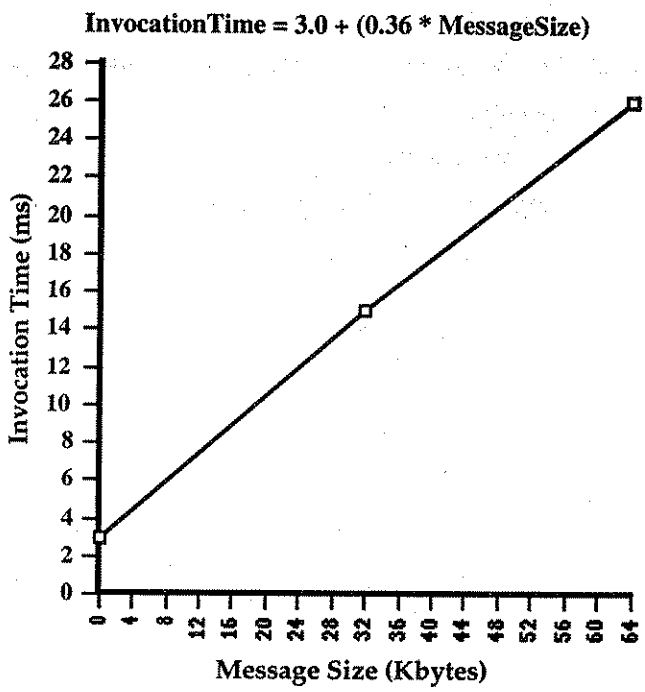


Figure 8.5: Message Passing Time Chart

8.5. Experimenting with the Simulation Models

In the following experiments, we shall consider object selection and retrieval operations only. Regarding object updates and deletion, those operations follow the retrieval of qualifying objects as we hinted in Chapter 7. Carrying out those operations is based on fetching the relevant object-pages into CVQR or UQS and overlaying their objects with those provided by the user and having matching ObjectIDs. This procedure can be applied to simple or to composite-objects and each of their component-objects. Since the update and deletion of objects does not require transformation of affected objects, we think the impact of both operations on system performance is not as serious as that of selection retrieval (unless more attention is to be paid to object sharing between multiple users).

8.5.1. Processor Load Balancing

In this subsection, we verify the benefits (identified in Section 8.1) of multitasking and process replication in improving query service time and processor utilization, and in reducing the impact of inter-process communication on the system performance. Here, we adopt some of the configurations listed in Table 8.3 to resolve the query qualifying all Employee objects. We examined those configurations using synchronous, asynchronous and mixed communication modes. In Table 8.7, we list the simulation output sorted in ascending order of query service times, starting with the best-performing configuration. It is clear that replicating processes improves query service time, while in the absence of processor sharing, the average processor utilization falls. Also, assigning processes to processors in such a way to achieve load balancing improved to some extent both query service time and CPU utilization. For example, the 3CVQR-10CPU configuration achieves the best query service time but produces the worst processor utilization in all communication modes. On the other hand, processor sharing in 2CVQR-4CPU¢ provides better query service time and processor utilization (around 10% better) than 1CVQR-4CPU configuration though both configurations use the same number of processors. Configuration 2CVQR-4CPU¢ also achieves between 8% and 10% less service time and 8% better CPU utilization compared to 2CVQR-4CPU† in different communication modes.

A particular feature is observed in Table 8.7, that is the slight lag of process-sharing configurations in mixed-communication mode behind the same configurations in synchronous mode (and asynchronous mode in a particular case). For example, configuration 2CVQR-4CPU† achieved a slight speed-up over the asynchronous and mixed communication modes. We relate this to the fact that the CLSR in synchronous mode waits for page-send to conclude for either CVQR before it starts serving the other process. This in effect gives the first-served CVQR more un-interrupted processor service than in other communication modes. Consequently, one CLTR and one CVM processes get the same advantage and the pipeline is filled faster than in the other communication modes. Since this configuration presents the worst load balance, the configuration performs almost identically in all modes following the initialization of pipelines. We also find other processor-sharing configurations (2CVQR-4CPU¢ and 2CVQR-4CPU§) maintain the speed-up initiated by the lag of one of pipelines at startup. However, in the asynchronous mode, such configurations suffer less from the communication delays and regain the lead.

This experiment demonstrates that replicating processes on shared processors improves system performance even with almost even page-processing-times of process types as in the case of the Employee class. We anticipate that with larger differences of page-processing-time between process types participating in a query, we can expect better improvement in system performance than that depicted in this experiment. Table 8.7 also shows that the performance of the asynchronous mode is slightly ahead of the other two modes. However, the speed-up of the asynchronous mode was notable when running the simulation with small page-processing-times that are close in magnitude to the page-communication overhead. It is clear that previous conclusions come in concert with the prototype findings in Section 8.1.

Query Configuration	Synch Service Time (ms)	Av. CPU Utiliz.%	Asynch Service Time (ms)	Av. CPU Utiliz.%	S/A Service Time (ms)	Av. CPU Utiliz.%
3CVQR-10CPU	8823	78	7151	82	7497	72
2CVQR-7CPU	10832	81	10420	84	10799	80
2CVQR-4CPU¢	18793	93	18766	93	18972	92
2CVQR-4CPU§	19236	91	18832	93	19348	91
3UQS-4CPU	20300	95	20244	95	20275	95
2CVQR-4CPU†	20579	85	20626	85	20629	85
1CVQR-4CPU	20999	83	20229	87	20999	83
2UQS-3CPU	30358	95	30317	96	30343	95

Table 8.7: Query Configuration Effect on Service Time[10]

8.5.2. Buffering Requirements[11]

Investigating the effect of the number of buffers allocated for an incarnation and the maximum buffer count for a wave in Equus revealed that for some query configurations the number of pending messages can be enormous and would require substantial space in node memory specially when object and result-pages are of large size. In this subsection, we conduct an experiment with pipeline processes having large differences between their page-processing-times. The CVQR, CLTR, CVM, UQS incarnations have page-processing-times of 100, 300, 50, 450 milliseconds respectively. Table 8.8 shows the effect of changing buffer count limit on the performance of query configurations.

For the A-Model, we notice that the single CVQR configuration requires at least 35 buffers to serve pending messages and complete query service. This is because the CVQR serves an object-page every 134 ms[12] (8040 ms for 60 pages) whilst the CLTR serves a page at lower rate of 300 ms. Accordingly, the CVQR eventually fills-in 34 page-buffers $(60-\left\lfloor\frac{8040}{300}\right\rfloor)$ while the CLTR reserves one buffer to forward its last-served page to the CVM. This requires a minimum of 35 buffers to serve the query, otherwise a deadlock situation persists and communication invocations will be blocked (indicated by *fail* in the table)[13].

[10] Service time for 1UQS-2CPU and 1UQS-1CPU is 60616 ms for all modes while CPU utilization is 96% and 98% respectively. The CPU serving CLSR is excluded from utilization calculations of all configurations and modes except for 1UQS-1CPU.

[11] We exclude from this study the S-Model since buffering is not applicable to it.

[12] composed of 3 ms (request page send) + 3 ms (request release page) + 5 ms (release locks) + 8 ms (send page) + 15 ms (page comm.) + 100 ms (CVQR service).

[13] In the prototype, this situation causes the wave operation to suspend.

A sįmilar requirement is recognized for 2CVQR-7CPU configuration since duplicate processes run on separate processors and consequently use communication buffers in the same manner as single CVQR but for less number of pages each. In the case of a 2CVQR-4CPU¢ configuration, we find that the minimum operable buffer limit reduces to 27 as a result of multitasking where two CVQR processes share the same processor and that in effect slows them down. In the case of UQS configurations, we found that the single UQS server is not affected by the buffer setting above 1 since its operation is serialized by communicating with the CLSR using one message at a time to either release page-lock, request or send an object-page for processing. For the duplicate configuration, each UQS requires at least one buffer which totals to 2 buffers as a threshold setting.

(Asynchronous-Mode Model)

Buffer Limit	Single CVQR	2CVQR-4CPU¢	2CVQR-7CPU	Single UQS	2UQS-3CPU
1	fail	fail	fail	29048	fail
2-26					14686
27-34		10753			
35 and above	18203		9211		

(Mixed-Mode Model)

Buffer Limit	Single CVQR	2CVQR-4CPU¢	2CVQR-7CPU	Single UQS	2UQS-3CPU
1	19091	11166	9656	29048	14694
2 and above			9649		14686

Table 8.8: Buffer Limit Effect with Different Query Configurations

For the mixed-mode (S/A) model, buffering is only required by CLSR to send object-pages to CVQR or UQS processes. So, at least one buffer is required to operate a wave with any number of process incarnations. Providing a number of buffers equal to or larger than the number of CVQR incarnations in the wave improves their performance slightly as they do not wait for freeing buffers to get their requested object-pages (see Table 8.8).

Equus requires that the output buffer not to be overwritten until message transmission is concluded. This means that a process should not be scrapped for garbage collection if it has pending messages even if its service has been concluded. So, buffering in Equus does not help reclaiming memory space occupied by a terminated process till its pipeline processes have all completed their work.

8.5.3. Object-Page Size

Here, we investigate the effect of communication overhead on query service time when changing the object-page size. For each experiment, Table 8.9 shows the result of asynchronous, synchronous, and mixed communication modes respectively. In the absence of processor sharing, we can see that the increased communication overhead for reduced page size has more tangible impact on query service time. However, UQS configurations are less affected than those of CVQR where the former record an increase of query service time for 4 Kbyte pages between 13% and 16% of that of 16 Kbyte pages, while the latter exhibit an increase between 29% and 33%. On the other hand, the performance of most of the shared process configurations is less affected by the decreased page size where some configurations record an increase of query service time by 8%. Some of the shared process configurations also achieve improved query service times when page size is reduced (shaded entries in Table 8.9). The reason behind this observation is that reducing page size consequently reduces page processing time in all process types (only *proportionate* page-processing-times referred to in Section 8.1). As page-processing-time of a process reaches a comparative order of the CLSR overheads, the weight of query processes changes in such a way that affects processors load balancing to the advantage of some configurations and not to others. Accordingly, 2CVQR-4CPU§ developed load weights of 4-5-3 which is the best for 4 Kbyte page size, followed by 2CVQR-4CPU¢ at 6-3-3 and 2CVQR-4CPU† at 6-4-2.

8.5.4. Processor and Communication Speed-Up

Changing the processing speed can be accomplished by either altering the clock frequency of the processor and/or by using different types of processor and memory. The first option means that the timing of the instruction cycle will change leading to different execution time of the instruction. So, increasing the instruction cycle time will increase the processing time of the same task and vice-versa. The second option may involve employing a different processor with more (or less) powerful instruction set. For example, a processor may support floating-point arithmetic, rely on auxiliary devices, or have more (or fewer) instruction cycles per instruction (such as RISC processors as opposed to conventional microprocessors).

Query Configuration	16 Kbyte Pages (120 pages)	8 Kbyte Pages (240 pages)	4 Kbyte Pages (480 pages)
3CVQR-10CPU	7010 7275 7324	7617 7875 8247	9061 9397 9389
2CVQR-7CPU	10337 10708 10807	11324 11692 12274	13528 14015 14011
2CVQR-4CPU¢	18158 18593 18548	18171 18135 18135	19526 20011 20011
2CVQR-4CPU§	18605 18681 18676	18613 19231 19226	18575 19963 18584
3UQS-4CPU	20629 20461 20629	21709 21332 21709	23869 23090 23869
2CVQR-4CPU†	20113 20091 20116	19952 19955 19955	19813 20055 20055
1CVQR-4CPU	20344 21061 21061	22471 23191 23191	26955 27916 27916
2UQS-3CPU	30916 30632 30916	32536 31949 32536	35776 34588 35776

Table 8.9: Page-Size Effect on Query Service Time[14]

Reducing the cost of inter-process communication may come partly as a consequence of improved processing speed. This is because the per-message overhead would be reduced irrespective of the volume of data transferred. Another way is to increase the bandwidth of communication between processors.

In our experimentation, we considered the mixed communication (S/A) model in which task processing and communication speed is increased, regardless of the realization of the speed-up. Table 8.10 depicts query service times for different process configurations at different processor and communication speed-up combinations. For shared-processor configurations, we can see that the improvement in communication at the same processing speed had little effect on query service time. The 2CVQR-4CPU¢ was the best achieving

[14] Service time of 1UQS-2CPU and 1UQS-1CPU for all modes is 61208 ms (for 16 KByte pages), 63848 ms (for 8 Kbyte pages), 69128 ms (for 4 Kbyte pages).

between 7% and 12% reduction in query service time, while 2CVQR-4CPU† achieved between 0% and 6%, and 2CVQR-4CPU§ achieved around 2% for all cases.

Doubling the processing speed (from CPU*2 to CPU*4) for CVQR configurations without processor sharing achieved around 40% improvement in query service time, and around 50% with multitasking. UQS configurations, on the other hand, achieved between 46% and 48% improvement in the absence of multitasking. The reason behind such less than expected performance improvement is that processors still encounter idle times due to communication between processes. Accordingly, increasing communication speed-up by 3 fold (CPU*4 and COM*3) achieves the reduction of query service time of CPU*2 by one-half which was expected from doubling processing speed.

Query Configuration	CPU*2	CPU*4	CPU*2 and COM*2[15]	CPU*4 and COM*3
3CVQR-10CPU	4121	2554	3662	2062
2CVQR-7CPU	5926	3649	5404	3069
2CVQR-4CPU¢	9831	5112	9274	4534
2CVQR-4CPU§	9549	4907	9338	4807
3UQS-4CPU	10469	5649	10315	5427
2CVQR-4CPU†	10415	5200	10058	4989
1CVQR-4CPU	11528	7121	10531	5798
2UQS-3CPU	15676	8446	15458	8134
1UQS-2CPU 1UQS-1CPU	31028	16568	30604	15962

Table 8.10: CPU and Communication Speed-up Effect on Query Service Time

8.5.5. Serving Multiple Users

Although the previous experiments did not show a significant advantage of asynchronous communication between the CLSR and CVQR (or UQS) over the other modes, we perceive that it can prove advantageous in a multi-user environment. This is because in the synchronous mode, the CLSR would have to wait for page transmission to conclude before it can serve pending requests from CVQRS (or UQSS) which may belong to a single or multiple users. If page transmission is of a considerable order (15 ms for a 32 Kbyte page in the prototype), then we would expect a notable delay in CLSR. We can roughly demonstrate how this

[15] Page communication cost estimated at 8 ms (as opposed to the normal 15 ms).

conclusion relates to the prototype by calculating the number of page requests served by a CLSR during the service time of a page in a CVQR (or UQS). We use the formula:

$$\frac{CLSR\_Page\_Service+CVQR\_Page\_Service}{CLSR\_Page\_Service}$$

CVQR_Page_Service includes page querying and mapping; requests for a page and/or for releasing locks on a page; communication overheads. CLSR_Page_Service includes sending a page to CVQR (or UQS); release of locks on a page; overheads of communication during which CLSR operation is suspended.

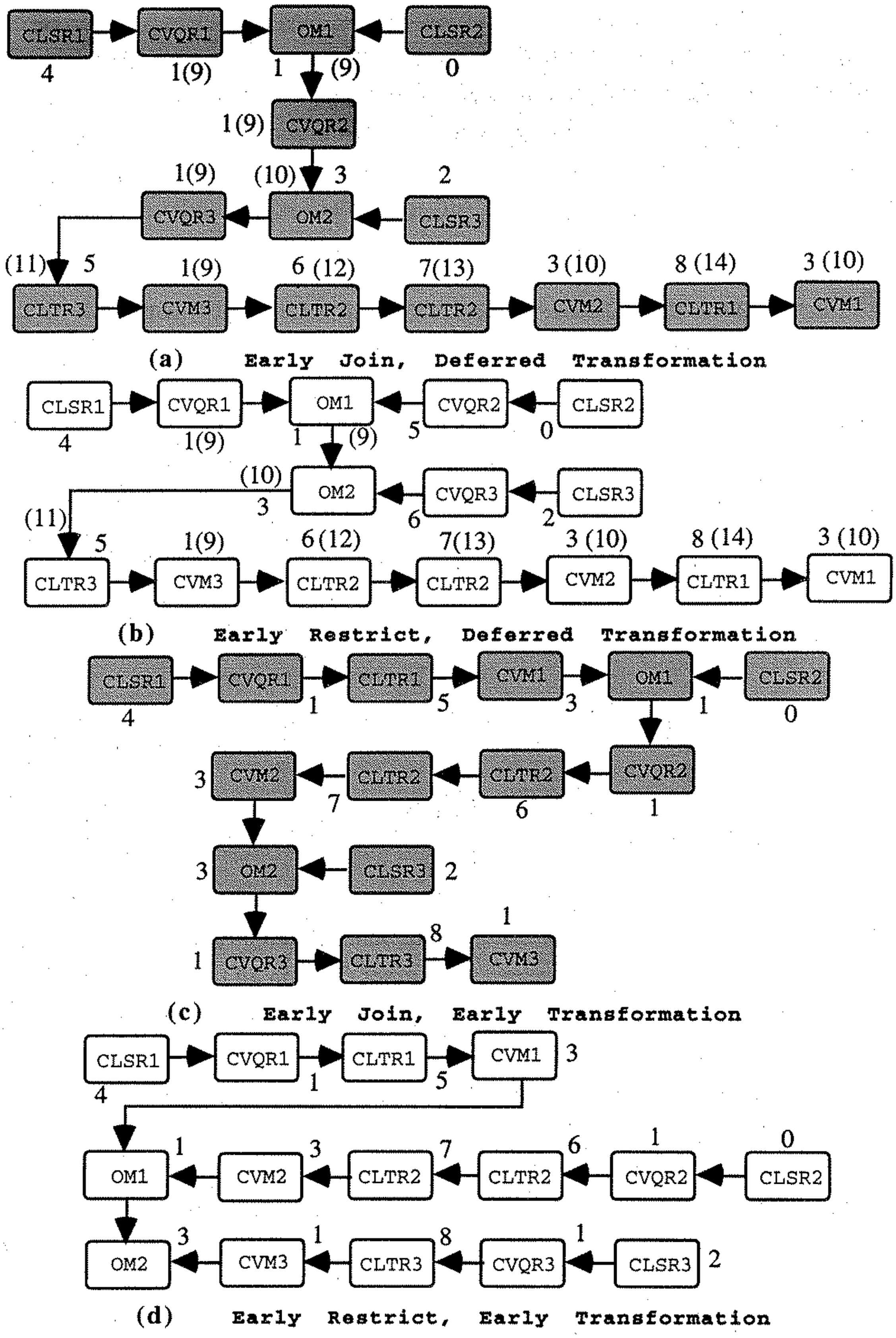
Recalling the activity timings mentioned in Section 8.1, we find that the CLSR serves a page-request from CVQRs (or UQSs) in 8+15+5=28 ms in synchronous mode, and in 8+5=13 ms in other modes. On the other hand, a CVQR serves a page every (3)+8+15+(285+3+15)=329 ms in synchronous and mixed communication modes, and every (3)+8+15+(285+3)+5=319 ms in asynchronous mode (the numbers between brackets represent CVQR activities and the rest are related to the CLSR). This gives the CLSR the chance to serve around 11 pipe-requests ($\frac{329}{28}$) in synchronous mode and 25 pipe-requests ($\frac{329}{13}$) in mixed communication mode and 24 pipe-requests ($\frac{319}{13}$) in asynchronous mode. Studying the UQS case reveals a similar gap between communication modes as far as pipe-requests count is concerned. A UQS serves a page every (3)+8+15+(965+3)+5=999 ms (the numbers between brackets represent UQS activities and the rest are related to the CLSR). This gives the CLSR the chance to serve around 35 pipe-requests ($\frac{999}{28}$) in synchronous mode and 76 pipe-requests ($\frac{999}{13}$) in other communication modes. Keeping in mind that pipe-requests exceeding those limits would cause delays to the operations of user-pipes, we can see that the asynchronous mode of inter-process communication enables more user-pipes to operate simultaneously than in the other modes and is therefore the most effective in multi-user environment followed by the mixed mode.

8.5.6. Manipulating Complex and Inter-Related Objects

In this experiment, we differentiate between process configurations discussed in Section 6.4 in terms of processor utilization and query service time. We apply hypothetical queries, posing different page-processing-times, to two databases of different specifications. The experiments involve one complex class with two component classes. One of the component classes has its objects shared, with varying degrees, between the composite-objects, while the second has exactly one instance per composite-object. We shall refer to those classes using the `Employee` database of Example 4 in Section 3.10 for its simplicity. However, in some cases, we shall take the liberty not to limit the sharing of component-objects to just two composite-objects.

In Figure 8.6, we illustrate the simulation models corresponding to the four process configurations of our query using the forward traversal mode mentioned in Section 2.6. Processes CLSR1, CLSR2, CLSR3 represent class-servers for Employee, Children, Spouse respectively, while CVQR1, CVQR2, CVQR3 are their corresponding query-resolvers. Process OM1 joins an Employee with his/her Children, while OM2 joins an Employee with his/her partner in Spouse. In those models, we assume that the component-object pages are already sorted according to their Object_IDs. This assumption is practical as a data page may be sorted whenever new objects are inserted in it. Such assumption was also considered for the database machine analysis in [Cesarini 87]. We consider the block of objects being exported from any process to include objects resulting from processing a single input object-page (or a block of objects). A block of objects can be of any size and its communication cost is calculated according to its size. In the following experiments, we shall examine the model's performance with different processor allocations and page-processing-times of processes. In the absence of multitasking and process incarnation, each configuration in Figure 8.6 would require 15 processors, while with multitasking and without incarnating only 9 processors are used (their Id numbers are indicated without brackets next to each process). In the case of multitasking with process incarnation, we used 15 processors for each configuration on which duplicate CLTR processes are assigned separate processors while other process types share processors as indicated in Figure 8.6. Processes serving Employee objects are replicated into two pipelines running in parallel (processor Id numbers used by replicas are indicated between round brackets in Figures 8.6(a,b) and are used for replicas in Figures 8.6(c,d)). The other classes would be replicated twice and three times in some cases as we shall indicate whenever applicable.

In Table 8.11, we describe the input parameters to the experiments. Page-processing-times of processes were chosen to be multiples of each other but not less than prototype estimates (in fact they are exaggerated per-object estimates of the prototype database). The size of the databases, comprising 3 classes each, is set to around 16 megabytes with two object size settings: 160 bytes (small-object database) and 640 bytes (medium-object database).



(a) **Early Join, Deferred Transformation**

(b) **Early Restrict, Deferred Transformation**

(c) **Early Join, Early Transformation**

(d) **Early Restrict, Early Transformation**

Figure 8.6: Simulation Models for Employee Database

The assumed sizes are in the ranges considered for some research projects. The object operations benchmark for engineering applications presented by Cattel and Skeen [Cattel 92] considers small databases of size 4 megabytes that would fit in main memory and it scales up its record count by a factor of ten to provide large database size of 40 megabytes. The object-based SOS operating system [Shapiro 89] on the other hand supports arbitrary, medium-sized (of the order of a hundred bytes or more) "SOS objects"[16].

In setting-up the test data, we took into consideration distributing the qualifying composite-objects randomly throughout processed object-pages. We also randomly choose the component-object pages (which themselves may have random number of qualifying component-objects) that need to be scanned in merge operations. This was intended to study processor sharing and communication between processes in a more realistic fashion without having a uniform load pattern for any processor involved in the query or having a fixed size for object blocks passed between processes. The following equations show how output-object-counts are generated for each process participating in the query:

```
output_object_count = input_object_count * expntl(AVG)

expntl(AVG) = - AVG * log(random())

random() returns a decimal number between 0.0 and 1.0 with a uniform distribution.
```

```
The expntl() function returns values greater than 1.0 for random numbers
less than 0.1.  As the count of such cases was trivial in 200 and 400
runs, we had a way around this problem by resetting such values to 1.0
without influencing the experiment outcome.
```

We assume that the Object-Merger (OM) processes have enough main storage to accommodate the entire inner-relation of the join operation (i.e., the component-object pages). We also assume that the CLSR processes cache in all their relevant object-pages prior to initiating the queries. This can be realized in real systems with processing nodes having caches of 7 megabytes or more (for each CLSR). So, we do not involve any secondary storage access time in the experiments. Based on the buffering investigation conducted in Subsection 8.5.2, we employ synchronous communication between all processes, except for the CLSR (S/A model). This is because the CLSR provides service to other processes on request, and it should be able to serve more than one query and more than one pipeline for each query as fast as possible without being blocked by synchronous communication.

[16] An SOS object includes collections of data with some code attached to it.

Employee and Spouse page counts	400, 100
Children page count	200, 50
object size (in bytes)	160, 640
object count per page	100
average hit ratio (AVG) of qualifying objects of either class (pre-join restrict)	0.5
merge success factor (both relations are restricted)	0.5
merge success factor (only outer relation being restricted)	1.0
distribution of qualifying objects of either class in its pages	exponential with random variant
random variant	between 0 and 1, uniformly distributed
cost of restricting a full-page in CVQR (msec)	70, 140
cost of restricting a full-page in CVQR using associative processor (msec)	10, 20
cost of transforming a full-page in CLTR (msec)	140, 280
cost of mapping a full-page in CVM (msec)	35, 70
cost of scanning an inner-relation object for a merge (μsec)	2
cost of merging two objects in OM (μsec)	1, 4
cost of scanning and merging an outer relation page with inner relation page in OM using associative processor (μsec)	100

Table 8.11: Experiment Parameters for Composite-Object Query

For object merge, we assume that 50% of inner-relation pages are checked for each page of the outer-relation (Employee in our case). An inner-relation page would have 25% of its objects scanned for a match of the join predicate for each object of the outer-relation (giving a join cost for a full outer-relation page with full inner-relation pages as high as 1000 and 250 milliseconds for the small-object and medium-object databases respectively). With associative processing, scanning an inner-relation page has a fixed cost in the join operation. The cost difference between 16 and 64 Kbyte pages was negligible for either database, so we assume the same cost of joining two pages of either size which is 100 microseconds. For the small-object database, this totals to 20 and 10 milliseconds for joining a page of Employee with 50% of Spouse and Children pages respectively. For the medium-object database, we consider 25% of the previous estimates following page-count proportion. According to the query-engine requirements described in

Chapter 9, we used the same cpu-quantum setting of 25 milliseconds (as in previous experiments) which enables the query-engine to conclude serving joins of an outer-relation page. Consistency has been maintained between different process configurations (i.e., in the four models) in terms of the impact of each process on the final result of the query. For example, merging objects of restricted Employee and Children in one configuration would result in the same number of objects we get from restricting Children following a join of restricted Employee with un-restricted Children. Adopting that approach ensures a fair comparison between process configurations.

In Table 8.12 we present the results of querying the small and medium-object databases both of which have similar size but different number of objects and page-processing-times. The conducted experiments employ pipeline processing without processor sharing or process incarnating and are intended to highlight the differences between the four process configurations. It is clear from Table 8.12(a)(b) that Configuration #4 is the less affected by the increase of object sharing where its performance improved by only 8% and virtually 0% for small and medium-object databases respectively. This is due to the fact mentioned in Chapter 6 regarding the minimized overhead for repeated transformation of shared objects, which makes Configuration #4 less vulnerable to object sharing. Configuration #2, on the other hand, recorded improvement of 14% and 2% respectively. Both configurations achieved around 10% improvement of CPU utilization for small-object database and no improvement for medium-object database. We notice that both configurations record less CPU utilization than Configurations #1 and #3 since they release some processors early during query processing. In those experiments, we can see that Configurations #2 and #4 did not provide better query service time as expected compared to the other configurations till object sharing was increased. Increasing object sharing in our experiments is achieved by reducing the hit ratio of qualifying constituent-objects (down from 50% to 12.5%) and producing the same query result. This in effect reduces the cost of join operations that normally dominate the performance of all configurations for small-object database (joining a full Employee page costs 500 and 1000 milliseconds with Children and Spouse full-pages respectively). The medium-object database, however, has the CLTRS dominating its query with page-processing-time of 280 ms compared to 125 and 250 ms for Employee page joins with Children and Spouse full-pages respectively. So, having all CLTR processes in the same pipeline achieves better performance than separating them into 3 pipelines (as in Configuration #4 in Table 8.12(b)). In Table 8.12(b), we can see that Configuration #2 performs worse than Configurations #1 and #3 because of the early restrict operations that delay the activation of `Employee` pipeline.

Configuration	Query Service Time	Avg. CPU Util. %	Max. CPU Util. %
Early Join, Deferred Transformation (#1)	110250	21	52
Early Restrict, Deferred Transformation (#2)	101510 87440	6 7	28 32
Early Join, Early Transformation (#3)	99660	23	57
Early Restrict, Early Transformation (#4)	102110 93740	12 13	27 30

(a)

Configuration	Query Service Time	Avg. CPU Util. %	Max. CPU Util. %
Early Join, Deferred Transformation (#1)	26090	13	54
Early Restrict, Deferred Transformation (#2)	40560 39740	5 8	35 35
Early Join, Early Transformation (#3)	27500	13	51
Early Restrict, Early Transformation (#4)	45030 45020	14 14	31 31

(b)

Table 8.12: Altering Object-Share Ratio for (a) Small-Object Database and (b) Medium-Object Database

15 CPUs allocated, each cell depicts results of sharing Children with 2 and 8 Employees respectively. Service time measurements are rounded to tens of milliseconds for easy grasp.

In Table 8.13, we study the effects of employing associative querying and joining objects while changing the number of allocated processors and introducing process replication and multitasking. We evaluated the four configurations for both databases in different cases: 15 CPUs allocated without processor sharing or process replication; 9 CPUs allocated with processor sharing and no process replication; 15 CPUs allocated with processor sharing and process replication. Processor sharing was performed according to our load balancing policy described earlier, but with processes having the largest page-processing-time allocated to separate processors. In some cases, process replication was limited to the `Employee` pipeline by doubling its processes. In other cases, both

Children and Spouse pipelines were duplicated alongside that of Employee. Without process replication and multitasking, we notice an improvement in query service time of around 40% with associative processing[17]. We also notice a drop of CPU utilization (reaching 90% in some configurations) giving way for more tasks to be run on allocated processors. The results of associative processing also show trivial increase in query service time with multitasking and 40% reduction in the allocated processors number and without process replication. The maximum CPU utilization also improves for Configuration #1, #2 by around 100%. Increasing the number of processors again but with replicating processes and employing multitasking achieved the best query service times which are around 25% less than those without process replication, and between 50% and 75% less than those without associative processing. Replicating the inner-relation pipelines (i.e., Children and Spouse processes) twice and three times in Configuration #2 gives it the lead amongst all configurations for both databases. For the first case, its query service time is less than that of Configuration #1 (which nearly provided the best performance in previous settings) by 29% and 17% for the small and medium-object databases respectively. For the second case, those percentages were increased to 40% and 33% respectively. Configuration #4 also achieved superiority over Configuration #1 for the small-object database only. However, increasing its replication and object sharing even more can give it the second place of best performance for both databases.

8.6. Conclusions

In this chapter, we investigated different operational and performance related issues, part of which were examined using the prototype, and then verified by simulation. Experimenting with the prototype indicated an advantage of combining multitasking and process replication with pipeline processing in improving query service time even with slight differences between page processing time of processes serving. It also demonstrated the effectiveness of our load balancing method.

As the prototype was implemented for simple-object manipulation, we employed simulation to examine composite-object manipulation and its alternative process configurations. Using simulation enabled us to examine large process configurations that would not fit into our prototype environment. We looked into the impact of changing some of the system and database parameters on system performance. For example, we studied buffering requirements of different modes of inter-process communication, and examined

[17] The conducted comparison is between measurements in the first row of each cell in Tables 8.12, 8.13.

the effect of reducing object-page size on query service time. We also demonstrated the impact of processing and communication speed-up that may be encountered on different platforms.

Configuration	Query Service Time	Avg. CPU Util. %	Max. CPU Util. %
Early Join, Deferred Transformation (#1)	38160	3	14
	38530	10	24
	28290	9	37
Early Restrict, Deferred Transformation (#2)	41970	5	12
	42080	9	20
	30990	6	34
	20960	10	29
	17620	12	26
Early Join, Early Transformation (#3)	48720	2	47
	48970	9	47
	33130	8	36
Early Restrict, Early Transformation (#4)	62510	1	37
	63100	21	37
	47700	26	48
	31720	29	46
	26390	31	41

(a)

Configuration	Query Service Time	Avg. CPU Util. %	Max. CPU Util. %
Early Join, Deferred Transformation (#1)	16050	2	14
	16310	13	25
	12080	10	22
Early Restrict, Deferred Transformation (#2)	18650	6	12
	18720	9	23
	14300	7	24
	10070	10	21
	8660	11	19
Early Join, Early Transformation (#3)	22000	2	56
	22110	10	56
	14660	8	43
Early Restrict, Early Transformation (#4)	31870	1	39
	32220	21	38
	24450	27	51
	16210	29	45
	13460	30	42

(b)

Table 8.13: Associative Processing for (a) Small-Object Database and (b) Medium-Object Database

In Table 8.13, the first 3 results in each cell are obtained respectively with settings: 15 CPUs allocated/no processor sharing/no process replication; 9 CPUs allocated/processor sharing/no process replication; 15 CPUs allocated/processor sharing/double process replication for Employee pipeline. The last two results in the second and fourth configurations are obtained respectively with double and triple replication of inner-relation pipelines.

We demonstrated the advantages of employing multitasking, process replication, and associative processing alongside pipeline and dataflow processing. Pipeline and dataflow processing achieved a speed-up of serving queries compared to sequential processing of uniprocessor environment. Multitasking enabled processors to switch between active processes and avoid being held by blocked or idle ones. This reduced processors idle time waiting for either data to be processed or for communication between processes to conclude. Process replication increased the degree of parallelism offered to each process and enabled balancing processors loads by distributing process replicas among allocated processors. Combining those techniques and methods achieved load balancing throughout processing nodes of the parallel machine, and reduced query service times. We showed that with such combination, it is possible to reduce the number of allocated processors with negligible penalty on performance. We demonstrated the benefit of weighting page-processing-time of processes relative to each other in the same pipeline to achieve efficient process distribution. Associative processing of object querying and joining reduced their execution time which used to dominate and overwhelm pipeline operation. This achieved reduction of query service time and reduced the utilization of processors responsible for those operations making them available to serve more processes.

The general conclusion of our experimentation using both the prototype and simulation is that large collection-graphs can be efficiently served with limited number of processors. This indicates the applicability and operability of the class-versioning approach in real systems.

In the following chapter, we shall present a design of an associative processor that is capable of delivering the improvised speed-up of object querying and joining. The design proves the practicality of the assumptions made in the simulation experiments and accordingly provides credibility to the conclusions drawn from our analysis.

Chapter 9
The SIMD Query-Engine

We perceive that the structure of either the CPU in von-Neumann architecture or the processing node in a MIMD machine (regardless of its configuration) can have a sound impact on the performance of object-oriented applications. This would be true if the node design is customized to the navigational as well as the pattern-matching operations on objects which characterizes the object-oriented applications. Here, we demonstrate how SIMD principles can achieve the aforementioned processor customisation and validate the simulation speed-up assumptions regarding associative techniques. We propose a design of a content-addressable query-engine, abbreviated to QE, as a step in that direction. We dedicate this chapter to the application of the query-engine for object manipulation. Appendix-A presents further details on the design, the estimation of query cost, and adapted logic and mathematical algorithms.

9.1. Introduction to Associative Memories

DeCegama [DeCegama 89] states a fundamental distinction between associative memory[1] (AM) and random access memory (RAM). That is: AM is content-addressable, allowing parallel access of multiple memory words, whereas RAM must be accessed sequentially by specifying the word addresses. Ertem [Ertem 89], on the other hand, points to the AM ability to signal the absence of a piece of data, unlike an explicitly addressed memory, where some data is always read, whether or not it is what is wanted.

Foster [Foster 76] presents the basic associative memory as a two-dimensional array of identical processing elements, or cells. The unit cell of the AM is several bits long and is capable of performing the standard functions of read/write like a RAM cell, but also contains sufficient logic to enable its bit content to be compared with the corresponding bit in a Comparand Register, or ignored depending on the setting of the corresponding bit of a Mask Register (see Figure 9.1). Information and commands are broadcast from the Central Control Unit (CCU) to all cells of memory in parallel. Each cell has associated with it a tag bit T_i (response store). A matching cell for a compare command issued by the CUU will set its tag bit, while a not-matching cell will reset that bit. As commands issued to memory cells will only affect cells with set tag bits, additional capability can be introduced to the

[1] Also called content-addressable memory (CAM).

AM by linking each tag bit to its immediate neighbours so that transferring (shifting) the activity from one cell to its neighbour become possible. This feature was realized in some implementations of the AM. For example, Ogura [Ogura 89] presents a design capable of performing associative operations on data extending over eight successive word locations (cells), while Choi [Choi 89] supports in his design forward linking of tag bits, for unlimited data element size.

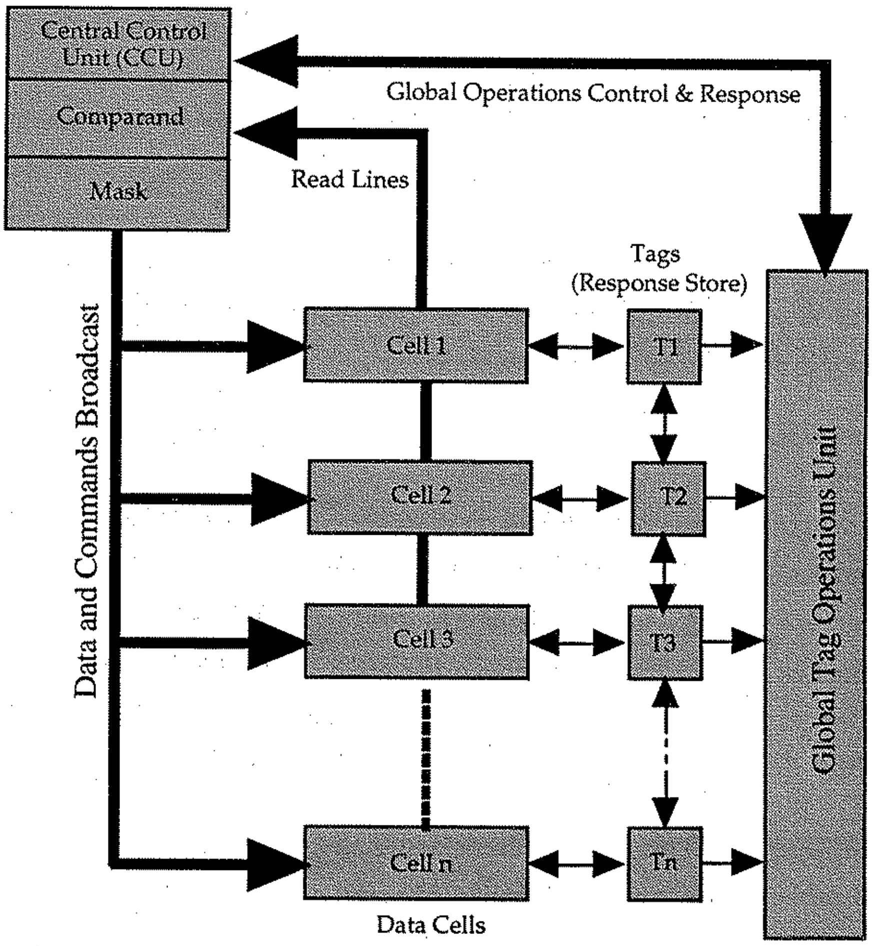

Central Control Unit (CCU)
Comparand
Mask
Global Operations Control & Response
Read Lines
Tags (Response Store)
Data and Commands Broadcast
Cell 1
Cell 2
Cell 3
Cell n
T1
T2
T3
Tn
Global Tag Operations Unit
Data Cells

Figure 9.1: The Conventional AM Organization

Parhami [Parhami 73] categorizes AM organization into four different classes, based on how bit/word slices are involved in the operation:

(1) The bit serial [Hwang 85]: which operates with one bit slice at a time across all the words. The time required for an operation to complete (also called the *cycle time*) using devices of this class is a linear function of the number of bits involved in the operation (except possibly for read and write).

(2) The word serial (the IFS/2 system, STARAN [Batcher 74]): which operates with all bits of one word slice at a time. The speed of operation in such devices depends on the size of their memory array.

(3) The fully-parallel (PEPE [Berg 72], SAP [Ng 88], Ogura [Ogura 89], IXM2 [Higuchi 91], GAM [Choi 89]): which operates with all bits of all word slices simultaneously. The speed of devices in this class depends on the operations implemented in hardware and on the hardware elements used. The cycle time of such devices increases as more complex search and arithmetic operations are to be supported because of carry or borrow propagation.

(4) The block-oriented (SURE [Leilich 78]): which operates on mass storage as data is being read (i.e., on-the-fly). The speed of this class depends mainly on the access time of the storage device involved and the used search criteria.

Parhami [Parhami 90] also recognizes a trade-off between storage capacity, speed and cost when choosing AM organization. Comparing these four organizations suggests that fully-parallel AM provides the highest speed (least cycle time) but the least storage capacity.

9.2. Modified Processor Structure

In our proposal, we follow the direction of adopting associative memories (AM) in supporting querying and manipulation of composite-objects. Unlike the IFS, the query-engine is not a stand-alone backend structure, but is intended for integration in processor nodes of parallel machines, or with the CPU in uniprocessor machines. Acting as a co-processor, the query-engine receives object-pages to be processed and user-queries or operations to be performed. The CPU is then freed to execute other tasks till the query-engine concludes its work. As we illustrate in Figure 9.2, the query-engine module would have direct memory access (DMA) to the node's local memory and access to the secondary storage via the node's I/O Controller. Such configuration speeds-up data movement within the node without posing an overhead on the CPU.

9.3. Object Manipulation Requirements

Querying composite-objects or inter-related objects (i.e., those referencing instances of other classes) within the context of the class-versioning approach is accomplished through the following sequence of operations:

(1) Restricting (or selecting) composite and constituent objects as specified by the query search conditions.

(2) If not clustered, objects may need to be joined where a composite-object is merged with its constituent-objects and referenced objects are merged with their referencing object.

(3) Transforming objects to the required class-version if their current class-version is different.

Querying simple objects, on the other hand, may require only the restriction and transformation operations, i.e., operations (1) and (3).

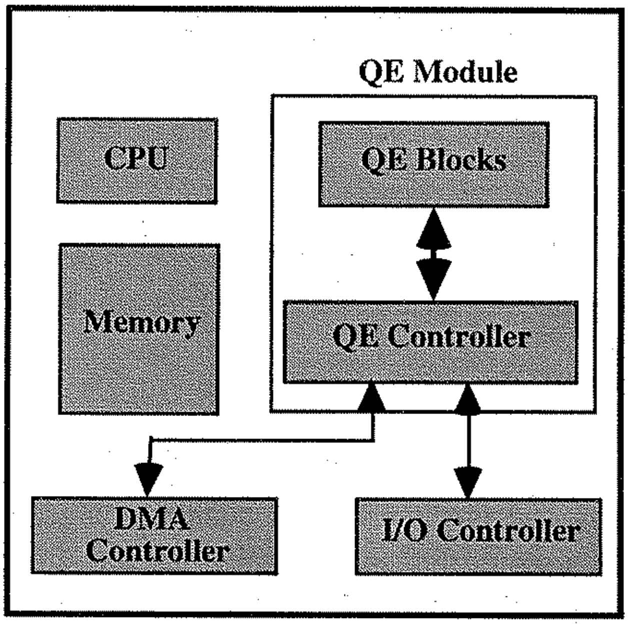


Figure 9.2: Modified Processing Node Structure

We perceive that associative techniques can be advantageous in improving the execution time of restriction and join operations. So, providing a specialized co-processor to carry out such operations would relieve the CPU to serve other tasks (e.g., object transformations). In this way, we could indirectly reduce the impact of run-time object transformation, dictated by the class-versioning approach, on the performance of existing or future systems. Accordingly, we took into perspective the following basic tasks when designing the query-engine:

- Identifying object boundaries, e.g., object header (and trailer if used by the user application). This aims at reducing the control information stored in the data-page header and the overhead involved in processing such information and locating objects of a specific class-version.

- Locating attributes within objects. This aims at reducing the control information stored in the object header and the overhead involved in locating specified attributes.
- Handling multiword attributes. This involves the manipulation of both numeric and alphanumeric data (such as text strings of variable length).
- Moving activity between neighbouring words. This provides for accessing specific words within attributes. The benefit of this is obvious when processing particular control information in the object header, or seeking partial attribute matches.
- Moving activity between neighbouring attributes. This provides for processing more than one predicate involved in the same query, without the need to store intermediate results.
- Parallel-by-word operation. This is the ultimate performance that can be achieved by AM organization and can speed-up data-intensive applications, especially when AM forms a central functional block in the computer architecture.

The current AM architectures lack the hardware support for all those tasks in the same design, and in attempting to achieve some of them, other drawbacks emerge. For example: [Choi 89] imposes a restriction on data by reserving a specific bit pattern for data element headers. Also, [Oruga 89] and GAM [Choi 89] limit the movement of activity to one direction. The query-engine design presents a solution to those limitations as we shall explain in the following sections. Hardware details of the design can be found in Appendix-A.

9.4. Query-Engine Architecture and Operation

The main functional elements of the query-engine are similar to those of the basic AM, namely: the Comparand, the Mask, the control unit and the associative words [Foster 76] (see Figure 9.3). Directing input commands to either Data words, the Mask, or the Comparator is achieved by issuing the proper selection command to the query-engine device(s).

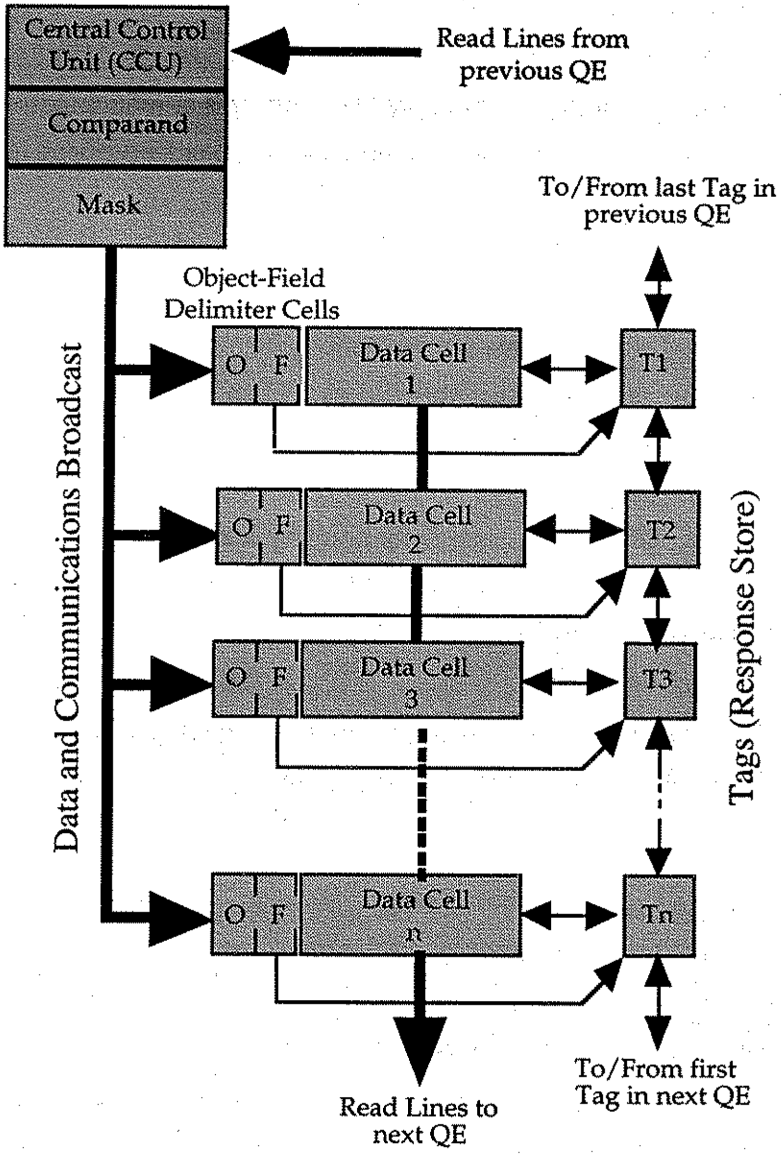

Central Control Unit (CCU)
Comparand
Mask
Read Lines from previous QE
To/From last Tag in previous QE
Object-Field Delimiter Cells
Data and Communications Broadcast
O
F
Data Cell 1
Data Cell 2
Data Cell 3
Data Cell n
T1
T2
T3
Tn
Tags (Response Store)
Read Lines to next QE
To/From first Tag in next QE

Figure 9.3: The Query-Engine Organization

The query-engine incorporates in each memory word (which is 8-bit long) additional associative cells of two types: the ***Object-Delimiter*** type, used to mark the first word of object header (and optionally its trailer), and the ***Field-Delimiter*** type, used to mark the first word of each attribute and to navigate throughout objects. The object and field-delimiter cells can be manipulated as normal data cells. However, the field-delimiter cell has additional feature of combining its state outputs in word control circuitry that incorporates the Tag bit. A memory word may have one field-delimiter cell at the most, but may have more than one object-delimiter cell. Allocating an object-delimiter cell to each constituent-class would provide optimum performance for accessing complex-object's components. However, a single object-delimiter cell per word would suffice to minimize the overhead of control gates per associative word, but with increased navigation overhead.

Selecting a memory word is accomplished by setting its Tag bit[2] to '1'. Two commands affect the setting of this cell: ***Compare*** and ***Set***. The Compare command matches the contents of each of the selected words with the Comparand according to the Mask setting, and if no match is found, the corresponding Tag bit is reset to '0', or otherwise it is left at its current state. The Set command sets all Tag bits in the query-engine to '1'. Each Tag cell is linked to its immediate neighbours via its control circuitry that permits propagating the Tag setting to the next/previous data word or field delimiter word. Navigating within the selected objects, those having words with their Tag bits set to '1', can be achieved with four navigational commands:

Forward Navigation

- ***link-next-word*** (LNW) to select the next-to-current selected word, and de-select the current one.
- ***link-next-field*** [3] (LNF) to select the next-to-current selected field, and de-select the current one.

Backward Navigation

- ***link-previous-word*** (LPW) to select the previous-to-current selected word, and de-select the current one.
- ***link-previous-field*** [3] (LPF) to select the previous-to-current selected field and de-select the current one.

With such flexibility of navigating within the stored objects in the forward/backward direction, the predicates in a multiple-field search condition can be evaluated in any order, irrespective of their physical locations within objects.

The query-engine has two modes of operation which determine how the associative words are affected by the launched operations (either navigational or data manipulation):

- The sequential mode causes only the top-most selected word in the engine to be affected.
- The parallel mode affects all selected words, simultaneously.

[2] The query-engine's Tag bit corresponds to that in conventional AM but with additional control circuitry.

[3] This control command is not effective if the current selected word is a field-delimiter.

Controlling the mode of operation is realized via the input Mode line of the query-engine. Intermixing sequential and parallel modes of operation is supported within the same logical unit of work (LUW). In the following sections, we shall demonstrate how this design fulfils the requirements stated earlier.

9.5. Object Representation

The information stored in the object mainly depends on the user application needs. However, the delimiter words of the object header and subsequent attributes should be marked using the Object-Delimiter and the Field-Delimiter bits. Accordingly, there would be no need for embedding control information in the object to locate such boundary words. In our case, that is supporting class versioning, we choose to record the Class-Id, the Class-Version-ID and the ObjectID in the object header, and assign an end-of-object (EOO) word to the object with its Object-Delimiter bit marked. Table 9.1 illustrates the attributes of a simple-object stored starting from the Field-Delimiter word, while object control information is stored in the object header.

Word Contents	Object Delimiter	Field Delimiter	ASCII Data
Class-ID	1	0	7D
Version-ID	0	0	01
ObjectID	0	0	FE
	0	0	A0
Name 'J'	0	1	4A
'o'	0	0	6F
'a'	0	0	68
'n'	0	0	6E
Rank 'A	0	1	41
EOO	1	0	10

Table 9.1: Simple-Object Format

The representation of clustered composite-objects would be similar to that of simple-objects where each constituent-object would be formatted as if it were a simple-object, but in a contiguous space to the composite-object. If such objects are stored in breadth-first (see Table 9.2), then the composite-object would have constituent ObjectIDs stored as separate fields. If composite-objects are stored in depth-first (see Table 9.3), then constituent ObjectIDs are removed from the composite-object and the EOO word of the latter follows the last constituent-object. The representation of un-clustered composite-objects would be similar to that of breadth-first but constituent-objects may be located anywhere in the database and not restricted to the vicinity of their composite-objects.

Word Contents	Object Delimiter	Field Delimiter	ASCII Data
Class-ID	1	0	7D
Version-ID	0	0	01
ObjectID	0	0	FE
	0	0	A0
Name 'J'	0	1	4A
'o'	0	0	6F
'h'	0	0	68
'n'	0	0	6E
Rank 'A	0	1	41
Spouse-ID	0	1	FE
	0	0	B0
Child-ID	0	1	FE
	0	0	C0
EOO	1	0	10
Class-ID	1	0	7E
Version-ID	0	0	02
ObjectID	0	0	FE
	0	0	B0
Name 'J'	0	1	4A
'o'	0	0	6F
'a'	0	0	68
'n'	0	0	6E
EOO	1	0	10
Class-ID	1	0	7F
Version-ID	0	0	04
ObjectID	0	0	FE
	0	0	C0
Name 'D'	0	1	4A
'a'	0	0	6F
'v'	0	0	68
'e'	0	0	6E
EOO	1	0	10

Table 9.2: Composite-Object Format in Breadth-First

9.6. Simple-Object Manipulation

Here, we present two examples of SQL[4] queries to demonstrate how search operation based on predicate matching is performed using forward/backward navigation capabilities of the query-engine. The examples involve LIKE predicates (i.e., partial pattern matching) using the 'symbol%' and '%symbol'[5] format, and illustrate how the Tag bits are affected during query processing. It should be noted that loading either the Mask or the Comparand register (or both), using a Write operation, is omitted in the illustration for brevity but

[4] Reference to SQL can be found in [Vossen 90].

[5] The % symbol is a placeholder for a sequence of zero or more arbitrary symbols.

implied for each Compare command. Object retrieval, update, and delete operations are described in detail in Appendix A.

Word Contents	Object Delimiter	Field Delimiter	ASCII Data
Class-ID	1	0	7D
Version-ID	0	0	01
ObjectID	0	0	FE
	0	0	A0
Name 'J'	0	1	4A
'o'	0	0	6F
'h'	0	0	68
'n'	0	0	6E
Rank 'A	0	1	41
Class-ID	1	0	7E
Version-ID	0	0	02
ObjectID	0	0	FE
	0	0	B0
Name 'J'	0	1	4A
'o'	0	0	6F
'a'	0	0	68
'n'	0	0	6E
EOO	1	0	10
Class-ID	1	0	7F
Version-ID	0	0	04
ObjectID	0	0	FE
	0	0	C0
Name 'D'	0	1	4A
'a'	0	0	6F
'v'	0	0	68
'e'	0	0	6E
EOO	1	0	10
EOO	1	0	10

Table 9.3: Composite-Object Format in Depth-First

Example 1:

We consider this sample query involving forward navigation operation of the query-engine:

```
Select * From EMPLOYEE Where NAME Like 'John%' And RANK='A'
```

We assume that all object classes have an associated identification number (`Class-ID`) which is 125 for class `Employee`. The steps of processing the query (see Table 9.4) start by issuing a Set command to set all Tag bits to '1' and thus selecting all words in the query-engine. Then a Compare command is performed to select object delimiter words SOO (**S**tart-**O**f-**O**bject), followed by a LNW to select the next word which contains `Class-ID`. A Compare for class 125 is performed followed by a LNF command to pass the search result to the `Name` field. Four Compare commands interleaved by a LNW are performed to search for the pattern 'John'. Then a LNF command is issued to pass the search result to the Rank field delimiter word and ignore the trailing data of attribute Name. Another Compare command is issued to check a match for Rank 'A'. Passing the final query result to the end-of-object (EOO) delimiter word is possible with a LNW command which will then transfer the Rank search result to the EOO word. A Compare for the (EOO) pattern can then be executed to reset the Tag bits elsewhere in the object to '0'. Only qualifying objects will have their Control Cells of that word set to '1'. If an exact match for Name (i.e., Name = 'John') is required, then after checking the pattern matching for 'John', a LNW command should be issued instead of LNF, and this should link to the field delimiter of Rank, otherwise, the exact match fails.

Example 2:

Now, we consider this sample query which involves the backward navigation operation of the query-engine:

```
Select * From EMPLOYEE Where NAME  Like '%Don'
```

In this example, it is possible to match the trailing data of the predicate only with backward linking of object words. Table 9.5 illustrates the sequence of commands issued to resolve this query. Note that checking the field delimiter bit of the start-of-field word (SOF) is required to reset the Tag bits elsewhere in the object to '0'. In example 1, the latter procedure was included in the Compare operation for the first word in the fields Name and Rank.

In the case of resolving nested queries and/or having multiple OR'ed interrogands in the same query, it would be necessary to allocate flag words either in attribute headers/trailers or in the object header to store intermediate results. Evaluating the final result of the query can then be achieved by processing those flag bits and selecting only qualifying objects. Considering the case of selection conditions being correlated using solely AND operators, the navigation capability of the query-engine would suffice to move the compare activity to interrogand attributes in sequence and the qualifying objects would then be selected automatically in a similar way to that of example 1. So, in the latter case, using flag bits will not be required. It should be noted that the methods of querying simple-objects described in this section can well be applied directly to composite-objects only if they are clustered with their constituent-objects.

				Tag bit contents															
word contents	object delimiter	field delimiter	data	Sel	Compare SOO	LNW	Compare 125	LNF	Compare J	LNW	Compare o	LNW	Compare h	LNW	Compare n	LNF	Compare A	LNW	Compare EOO
class id	1	0	7D	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
version id	0	0	01	1	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0
object id	0	0	01	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	FE	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
name 'J'	0	1	4A	1	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0
'o'	0	0	6F	1	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0
'a'	0	0	68	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
'n'	0	0	6E	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
' '	0	0	40	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
'D'	0	0	44	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
'o'	0	0	6F	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
'n'	0	0	6E	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
rank 'A'	0	1	41	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
EOO	1	0	10	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
class id	1	0	7D	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
version id	0	0	01	1	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0
object id	0	0	01	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	FF	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
name 'J'	0	1	4A	1	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0
'o'	0	0	6F	1	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0
'h'	0	0	68	1	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0
'n'	0	0	6E	1	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0
' '	0	0	40	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
'B'	0	0	42	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
'e'	0	0	6C	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
'n'	0	0	6E	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
rank 'A'	0	1	41	1	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0
EOO	1	0	10	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1

Table 9.4: Query Processing for Example 1

				Tag bit contents													
word contents	object delimiter	field delimiter	data	Set	Compare SOO	LNW	Compare 125	LNF	LNW	LNF	Compare SOF	LPW	Compare n	LPW	Compare o	LPW	Compare D
class id	1	0	7D	1	1	0	0	0	0	0	0	0	0	0	0	0	0
version id	0	0	01	1	0	1	1	1	0	0	0	0	0	0	0	0	0
object id	0	0	01	1	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	FE	1	0	0	0	0	0	0	0	0	0	0	0	0	0
name 'J'	0	1	4A	1	0	0	0	1	0	0	0	0	0	0	0	0	0
'o'	0	0	6F	1	0	0	0	0	1	1	0	0	0	0	0	0	0
'a'	0	0	68	1	0	0	0	0	0	0	0	0	0	0	0	0	0
'n'	0	0	6E	1	0	0	0	0	0	0	0	0	0	0	0	0	0
' '	0	0	40	1	0	0	0	0	0	0	0	0	0	0	0	0	0
'D'	0	0	44	1	0	0	0	0	0	0	0	0	0	0	0	1	1
'o'	0	0	6F	1	0	0	0	0	0	0	0	0	0	1	1	0	0
'n'	0	0	6E	1	0	0	0	0	0	0	0	1	1	0	0	0	0
rank 'A'	0	1	41	1	0	0	0	0	0	1	1	0	0	0	0	0	0
EOO	1	0	10	1	0	0	0	0	0	0	0	0	0	0	0	0	0
class id	1	0	7D	1	1	0	0	0	0	0	0	0	0	0	0	0	0
version id	0	0	01	1	0	1	1	1	0	0	0	0	0	0	0	0	0
object id	0	0	01	1	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	FF	1	0	0	0	0	0	0	0	0	0	0	0	0	0
name 'J'	0	1	4A	1	0	0	0	1	0	0	0	0	0	0	0	0	0
'o'	0	0	6F	1	0	0	0	0	1	1	0	0	0	0	0	0	0
'h'	0	0	68	1	0	0	0	0	0	0	0	0	0	0	0	0	0
'n'	0	0	6E	1	0	0	0	0	0	0	0	0	0	0	0	0	0
' '	0	0	40	1	0	0	0	0	0	0	0	0	0	0	0	0	0
'B'	0	0	42	1	0	0	0	0	0	0	0	0	0	0	0	0	0
'e'	0	0	6C	1	0	0	0	0	0	0	0	0	0	1	0	0	0
'n'	0	0	6E	1	0	0	0	0	0	0	0	1	1	0	0	0	0
rank 'A'	0	1	41	1	0	0	0	0	0	1	1	0	0	0	0	0	0
EOO	1	0	10	1	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 9.5: Query Processing for Example 2

9.7. Query Translation

As we mentioned in Chapter 4, accessing objects of other class-versions requires generating queries for those versions from the user-query. Those queries should then be translated into sequences of query-engine operations and passed on to the query-engine controller (see Figure 9.2). The translation process can be carried-out by a preprocessor dedicated for that purpose. Here, we demonstrate with an example the translation process and how the query-engine executes generated operations to resolve such queries. We assume the following query is being launched by the user for class-version CV_X :

```
Select * From EMPLOYEE(CVx) Where NAME  Like '%Don'
```

where NAME is the first attribute in object.

From that query, a new class-version query for CV_y is created as follows:

```
Select * From EMPLOYEE(CVy) Where FULLNAME Like '%Don'
```

where FULLNAME is the second attribute in object.

The operation-generator within the preprocessor produces the command sequence shown in Table 9.6 which are grouped into five segments. This illustration shows how the query-engine will resolve both queries and retrieve the qualifying objects. At first, objects belonging to the targeted class-version are selected (the top section in Table 9.6). This is followed by locating the interrogand attribute in the selected object (the second section in Table 9.6). Attribute evaluation is then performed (the middle section of Table 9.6). Operations in all three sections are executed in the parallel-mode to affect all related objects at the same time.

To retrieve qualifying objects, starting from the object header down to the last word in each object, we need to move the word selection activity, from the current position in object and backwards towards the object-delimiter word (the fourth section of Table 9.6). It should be noted that LPW and LPF commands for backward navigation within objects are executed in the parallel-mode. Reading-out selected objects is performed in the single-mode, where object words are retrieved one at a time. Upon retrieval, the word is de-selected by comparing its contents to an illegal value (in this case it is Hex. 3XX which means a word with both delimiter-bits set to '1' while the data part is insignificant and may take any value). The read operation is illustrated in the lower section of Table 9.6.

CV$_x$ Command	Sequence	CV$_y$ Command	Sequence
SET	select all words	***SET***	select all words
WRITE	Comparand	***WRITE***	Comparand
WRITE	mask	***WRITE***	mask
COMPARE	object-delimiter word	***COMPARE***	object-delimiter word
LNW		***LNW***	
WRITE	Comparand	***WRITE***	Comparand
WRITE	mask	***WRITE***	mask
COMPARE	version ID (x)	***COMPARE***	version ID (y)
LNF	first attribute in object	***LNF***	first attribute in object
WRITE	Comparand	***WRITE***	Comparand
WRITE	mask	***WRITE***	mask
COMPARE[$]	field-delimiter word	***COMPARE***[$]	field-delimiter word
LNW		***LNW***	
LNF	second attribute in object	***LNF***	second attribute in object
COMPARE	field-delimiter word	***COMPARE***	field-delimiter word
LPW	last word in first attribute	***LNW***	
WRITE	Comparand	***LNF***	third attribute in object
WRITE	mask	***COMPARE***	field-delimiter word
COMPARE	'n'	***LPW***	
LPW		***WRITE***	Comparand
WRITE	Comparand	***WRITE***	mask
COMPARE	'o'	***COMPARE***	'n'
LPW		***LPW***	
WRITE	Comparand	***WRITE***	Comparand
COMPARE	'D'	***COMPARE***	'o'
		LPW	
		WRITE	Comparand
		COMPARE	'D'
LPF	top word of first attribute	***LPF***	top word in second attribute
WRITE	Comparand	***WRITE***	Comparand
WRITE	mask	***WRITE***	mask
		COMPARE	field delimiter
		LPW	last word in first attribute
		LPF	top word of first attribute
COMPARE	field-delimiter word	***COMPARE***	field-delimiter word
LPW	last word in object header	***LPW***	last word in object header
LPF	object-delimiter word	***LPF***	object-delimiter word
WRITE	Comparand	***WRITE***	Comparand
WRITE	mask	***WRITE***	mask
COMPARE	object-delimiter word	***COMPARE***	object-delimiter word
WRITE	Comparand (illegal value)	***WRITE***	Comparand (illegal value)
WRITE	mask	***WRITE***	mask
LNW	select next word to be read	***LNW***	select next word to be read
COMPARE	top-most selected word	***COMPARE***	top-most selected word
*	*	*	*
*	*	*	*
LNW	select next word to be read	***LNW***	select next word to be read
COMPARE	top-most selected word	***COMPARE***	top-most selected word

Table 9.6: Generated Command Sequence for Queries on Class-Versions X, Y

[$] This Compare operation is intended to de-select previous associative words so that only one word at the most is selected in the object at any time.

The cost of running such query is estimated using the formula produced in Appendix A. We estimate that a single basic operation takes 5 nanoseconds to complete (minimum cycle time), while switching the read operation between qualifying objects is proportional to the number of associative words (N) in the query-engine. Assuming that 100 objects occupy a 4 Kbyte data-page, the query cost per page for either CV_x or CV_y is around 45 microseconds with all objects qualifying. In a RAM-based system, similar queries would cost around 300 microseconds based on 1 microsecond service time per character in the predicate. Although this example presents a simple query for a small page size, it still depicts the speed-up achieved by the query-engine (almost 7 fold). In this example, we notice that the performance of the query-engine is almost flat (i.e., the cost difference between class-version queries is negligible) when adopting the parallel mode of operation and executing a read operation on a substantial number of objects.

9.8. Composite-Object Manipulation

Here, we consider un-clustered composite-objects where join operation is required to merge composite-objects with their constituent objects. We can find different situations that would require different handling algorithms in the query-engine. That is, we may be dealing with composite-objects having only single or multiple instances of their composite attributes. In the first case (e.g., Employee-Spouse), composite-objects are excluded from further checks once they are joined with the first encounter of eligible constituent objects. In the second case (e.g., Employee-Children), composite-objects require repeated checks till all (or some of) the constituent objects of each composite-object are located (depending on the query). In this section, we consider the first case and develop the suitable algorithm for assembling the composite-object. The algorithm for the second case can be devised based on the discussion that follows. In Figure 9.4, we illustrate the former case with an algorithm of joining one page of un-restricted composite-objects, loaded in query-engine Block#1, with pages of unrestricted constituent-objects, loaded in sequence into query-engine Block#2. We start with simultaneous selection of objects in both blocks according to the predicates of the query. Qualifying objects are marked by setting a bit in the object header. Marked objects in Block#1 (the outer join relation) are served one at a time by retrieving them in sequence into the query-engine controller. The latter extracts the constituent ObjectID from the composite-object and loads it into the Comparator of Block#2. Objects in Block#2 are then searched for that ObjectID and if the required object is located, its marking is checked. If the object is marked, it is retrieved and joined with its composite-object in the query-engine controller, otherwise the composite-object mark is cleared and thus excluded from further checks. If the constituent-object is not located, a counter in query-engine controller is incremented to account for the composite-object in latter check passes. Following the service of a composite-object, which would be the top-selected in Block#1, the next qualifying composite-object is made the top one.

The previous steps would be repeated for each marked composite-object in Block#1. Following that service pass, the counter is checked. If it is zero, then the join is concluded, otherwise, another page of constituent objects (if any) is loaded into Block#2, checked against query predicates, and marked. The join operation then resumes for the remaining marked composite-objects. Each successful join for a composite-object following the first pass results in decrementing the counter of the query-engine controller. When the counter is back to zero, the join ends and another page of composite-objects (if any) may be loaded into Block#1 and be served as well. When operating the query-engine in a multitasking environment, the cpu-quantum allocated to each executing task should be adequate to load the pages to be joined into the query-engine, and to complete the join of both pages.

Now, we consider the case of joining restricted composite-object pages with restricted constituent-object-pages. The algorithm applied here (shown in Figure 9.5) is a simplification of that described earlier as we assume object-pages contain only objects qualifying to query predicates. The cost of executing this algorithm, based on the specifications mentioned in Section 9.7, is calculated using the following equation in which we account for: retrieving the composite-object from Block#1, selecting constituent-objects from Block#2 that match the `ObjectID` (2 words) referenced in the composite-object, retrieving the referenced constituent-object. The equation multiplies the cost of those operations by the count of composite-objects in Block#1 to provide the service time per composite-object page.

```
    Object_count_per_page  *  [(2  *  Object_Retrieval_cost)  +
                    (2 * Selection_in_Block#2_cost)]
```

and results in 100 microseconds for joining all objects in Block#1 with objects of Block#2. In this calculation, we ignored the overhead of the query-engine controller, but this should not be of a significant order. In a RAM-based system, such an operation would cost around 5 milliseconds (assuming 25% of Block#2 objects are scanned for each `ObjectID` matching and that an object-scan costs 2 microseconds). This means a speed-up of the query-engine of 50 fold.

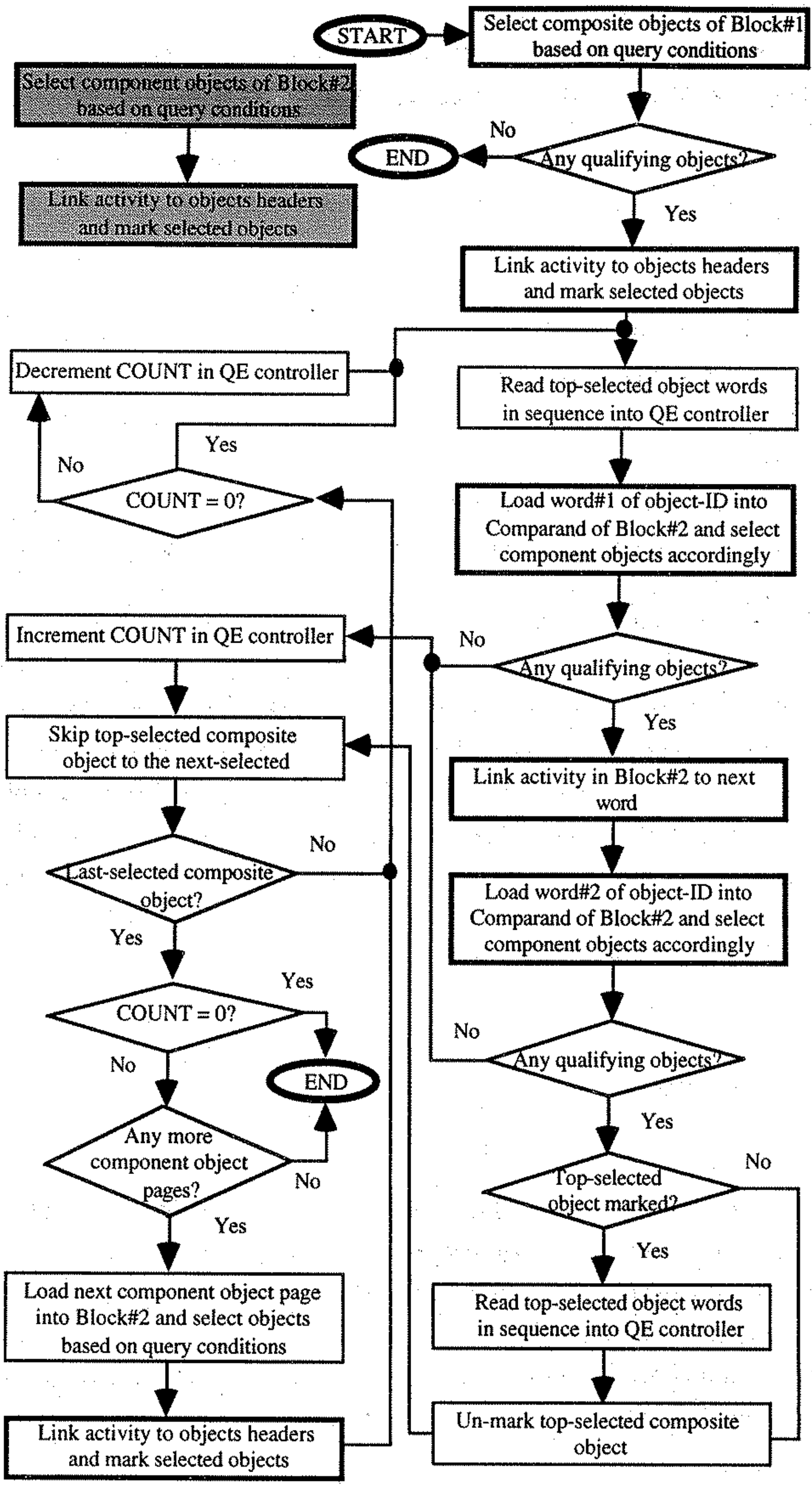

START
Select composite objects of Block#1 based on query conditions
Select component objects of Block#2 based on query conditions
Link activity to objects headers and mark selected objects
No
END
Any qualifying objects?
Yes
Link activity to objects headers and mark selected objects
Decrement COUNT in QE controller
Read top-selected object words in sequence into QE controller
Yes
No
COUNT = 0?
Load word#1 of object-ID into Comparand of Block#2 and select component objects accordingly
Increment COUNT in QE controller
No
Any qualifying objects?
Yes
Skip top-selected composite object to the next-selected
Link activity in Block#2 to next word
No
Last-selected composite object?
Load word#2 of object-ID into Comparand of Block#2 and select component objects accordingly
Yes
Yes
COUNT = 0?
No
No
END
Any qualifying objects?
Yes
Any more component object pages?
No
Top-selected object marked?
No
Yes
Yes
Read top-selected object words in sequence into QE controller
Load next component object page into Block#2 and select objects based on query conditions
Link activity to objects headers and mark selected objects
Un-mark top-selected composite object

Figure 9.4: Assembling Un-restricted Composite Objects

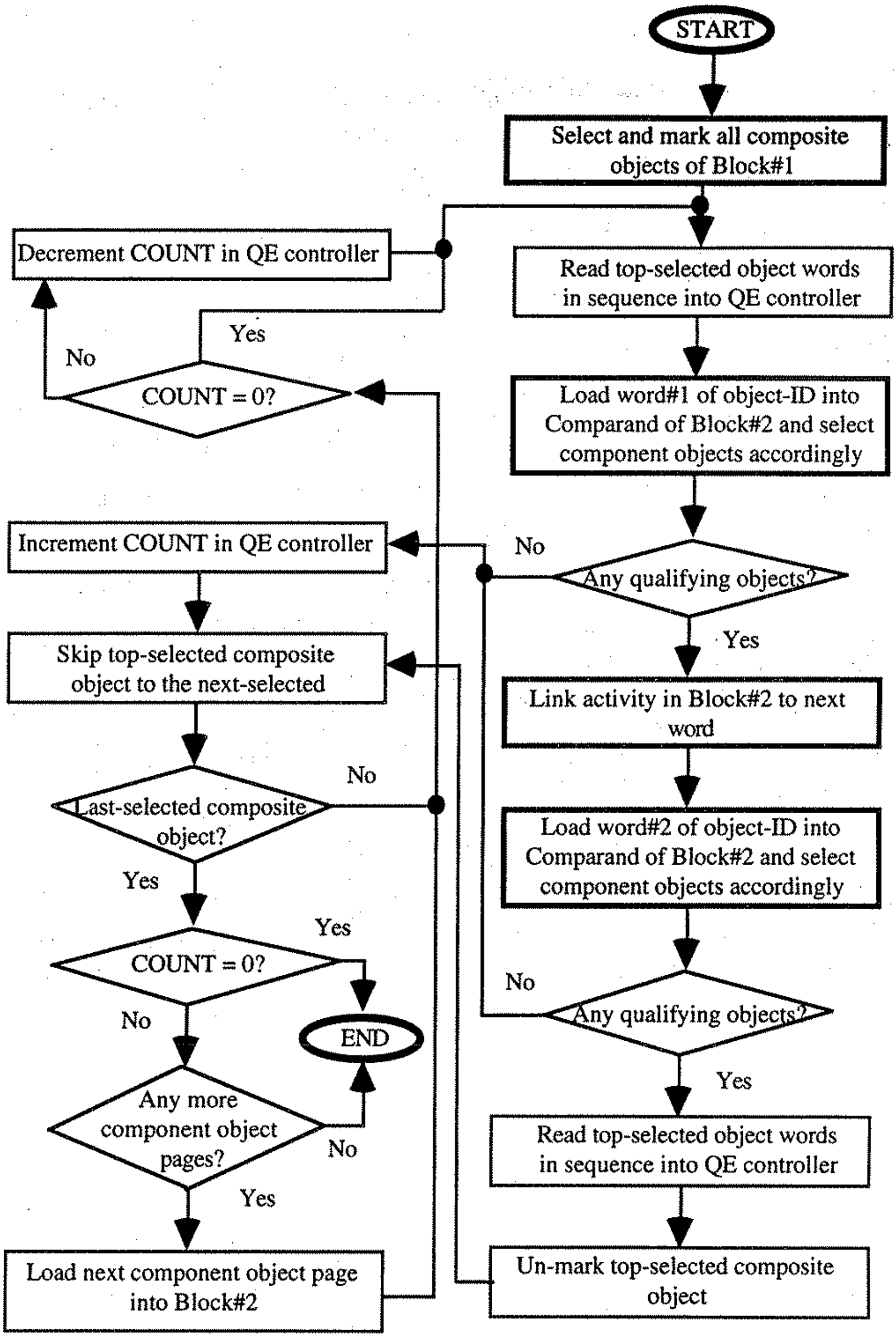

START
Select and mark all composite objects of Block#1
Read top-selected object words in sequence into QE controller
Load word#1 of object-ID into Comparand of Block#2 and select component objects accordingly
Any qualifying objects?
No
Yes
Link activity in Block#2 to next word
Load word#2 of object-ID into Comparand of Block#2 and select component objects accordingly
Any qualifying objects?
No
Yes
Read top-selected object words in sequence into QE controller
Un-mark top-selected composite object
Decrement COUNT in QE controller
COUNT = 0?
Yes
No
Increment COUNT in QE controller
Skip top-selected composite object to the next-selected
Last-selected composite object?
No
Yes
COUNT = 0?
Yes
No
END
Any more component object pages?
No
Yes
Load next component object page into Block#2

Figure 9.5: Assembling Restricted Composite Objects

9.9. Conclusions

In this chapter, we presented the design of an associative memory called query-engine that can perform search and update operations, involving multi-word attributes, on cached objects in parallel or sequentially. Unlike existing associative processors, the query-engine supports both bi-directional navigation (forward and backward) within objects and facilitates direct manipulation of objects and their attributes with minimal navigation overhead. The query-engine does not impose restriction on object or attribute size or reserve any bit patterns to identify object or attribute headers. Query-engine devices can be cascaded to achieve the required cache frame or data-page size. Rather than using separate specialized processors for operations such as object selection and join operations, the query-engine performs such operations in addition to a variety of logic and mathematical algorithms. Objects can be relocated in the query-engine without the need to change their references or employ indirection in contrast with RAM-based systems. Supporting navigation between and within objects eliminates the need for storing intermediate query results (in some cases) and provides better performance for nested and complex queries. The associative operation of the engine eliminates the need for maintaining multiple indices on object-attributes. Such features make the query-engine capable of both resolving user queries related to persistent objects, and manipulating those objects locally instead of transferring them to the host main memory for processing. We demonstrated, by the means of examples, the speed-up achieved by the query-engine for both object selection and join. In resolving a simple selection query, it was 7 fold faster than an assumed RAM-based system. Such speed-up is expected to increase as queries become more complex. In joining objects of two pages, a speed-up of 50 fold was achieved.

SUMMARY OF PART III

In this part, we presented the parallelism inherent in the class-versioning approach which may be exploited to improve system performance. We demonstrated how query generation for version-collections and class-versions benefits from the parallel execution of the preprocessor in run-time when large Collection-Graphs are involved. We also discussed the alternative parallel processing configurations for manipulating simple, composite and inter-related objects and highlighted their advantages and disadvantages. We elaborated on the benefit of data partitioning that comes naturally with dividing class instances into class-versions. A taxonomy of processes that would participate in manipulating objects within the context of the class-versioning approach was presented.

We presented a prototype for simple object manipulation as a test platform to investigate the performance of different process configurations with synthetic data. The prototype highlighted some practical aspects for realizing processes of different types. It also highlighted the advantages and limitations of the adopted parallel machine and operating system that qualify as a potential implementation environment.

We demonstrated by means of experiments on the prototype a potential of combining multitasking and process replication, with pipeline processing, in reducing query service time, balancing processor load, improving processor utilization, and compensating for inter-process communication overhead. The prototype provided estimates of the basic activities of different process types that helped in setting up simulation experiments. Simulation was employed to investigate simple and composite object manipulation with different database and system characteristics. It enabled us to indicate potential advantages of combining associative processing with the aforementioned techniques to improve query service time and processor utilization. The combined processing techniques also reduced the processing resources required to perform dynamic object transformation during run-time, and showed the feasibility of serving large collection-graphs on parallel machines with limited number of processors.

The realization of the proposed associative processing capabilities was presented in the form of a SIMD query engine that would act as a co-processor to node processors in multiprocessor machines or to the CPU in von-Neumann architectures. We highlighted the advantages of the query-engine over existing associative processors and the benefits it provides for querying and manipulating simple and composite objects.

Part IV

Conclusions, Appendices and Bibliography

In this part, we present conclusions emerging from this dissertation, and highlight future research directions that may build upon what we have accomplished. This is followed by Appendix-A giving more details of the design and application of the query-engine. Finally, we provide a bibliography of the references used.

This page is intentionally left blank

Conclusions and Future Work

The main theme of this dissertation was class versioning as one of the approaches realizing schema evolution in OODBMS. The research was motivated by limitations and deficiencies of the existing proposals and implementations of that approach and the lack of performance investigation in conjunction with the approach realisations. So, in this research, we tackle both functionality and performance aspects of such approach. From the functionality point of view, we proposed a class-versioning approach that resolves some of the drawbacks of existing systems and present additional features suitable for a wide range of applications. We enumerated those drawbacks as: limited support for schema changes related to attribute semantics; compromising object data as a consequence of object migration or transformation between class-versions; confining schema modifications to the latest versions of classes resulting in linear versioning. Although not all of those drawbacks were encountered in any single system, none of existing systems had the potential of resolving all of them. We also perceived the necessity to investigate the performance of systems adopting class-versioning especially when they perform object transformation dynamically during run-time. So, we dedicated part of our effort to study performance issues pertaining to our approach and to show how parallel processing can improve the performance of systems adopting our approach.

The proposed class-versioning approach groups versions of the same class into version-collections. A version-collection contains a union of all attributes defined in its affiliated class-versions. It is also provided with transformation functions to map objects, cast according to its attributes definition, to other neighbouring version-collections. Version-collections of a class are configured into *Collection-Graph* of two distinct forms: *Tree* and *Lattice*. A class-version lacking an attribute of its version-collection is provided with a handler that assigns a default value for the attribute or calculates its value from existing attributes data. So, an object of a class-version may be transformed to other class-versions in the same version-collection or in other version-collections of the same class by first mapping the object to its local version-collection (using handlers if required) and then projecting it to other neighbouring class-versions, or mapping it to other version-collections using transformation functions. Due to the restrictions on schema modifications in existing systems, many important issues that we dealt with in our proposal did not arise (or were ignored) and were not addressed in those systems. We proposed extending attribute semantics from merely characterising the type of values eligible for an attribute to specifying the range, type, and units of measurement (if any) of those values.

This gave rise to the issue of attributes compatibility between class-versions and, consequently, the placement of class-versions in version-collections and the configuration of version-collections in the collection-graph. We provided a taxonomy of supported schema changes, based on attribute semantics, that would specify when a class-version may join a particular version-collection. We differentiated between collection-graph forms in terms of the changes they support to attribute semantics and existing transformation functions. We examined what we called *Transformation Path Sensitivity* which is concerned with the conservation of object data throughout the transformation paths between any two class-versions. We highlighted the persistence of object data as an outstanding issue when objects are allowed to migrate between class-versions. We proposed the concept of *Intermediate Class-Versions* that would accommodate migrating objects if their destination class-version drops any of the objects attributes or tightens their attributes domains. We identified the motivations for objects to migrate within the context of the class-versioning approach. We also highlighted the possible inconsistency of data processed by user transactions as a consequence of object migration. The prevention methods for such a case were described. We explained how the class-versioning approach can support schema versioning.

We extended our research to study some implementation aspects pertaining to our approach. We examined the applicability of simple and composite-object physical storage models to our approach, and we came up with some in-applicable, some applicable but with performance penalty, and some suitable and recommended. We presented alternative methods of object retrieval and update, and we differentiated between them from performance point of view. We showed that querying class-versions also had alternative methods that differ in the way they access instances of class-versions for a user-query formulated for a specific class-version. We discussed those methods and examined their performance implications. Query optimization is another implementation aspect that would affect system performance. We presented the features inherent in our approach that would contribute to the query optimization task. We described how to extend the C++ programming language to support seamless querying and manipulation of persistent objects with different class-versions and to eliminate impedance mismatch. Part of the language extension was implemented in our prototype of parallel object manipulation. Studying the applicability of the class-versioning approach to extended object-oriented and semantic data models was conducted using case studies and revealed limitations imposed by some associations between classes to support certain schema changes endorsed by our approach.

Optimising the use of intermediate class-versions requires more investigation to have their creation based on global analysis of the collection-graph rather than on individual class-version basis. This can minimize changes to class-version handlers and transformation functions of version-collections during run-time. Permitting cycles in the collection-graph requires producing algorithms to pick the best path connecting any two class-versions from performance point of view. A schema editor may well be designed to help evaluate the practicality of adopting the class-versioning approach in frequently evolving environments. Applying the class-versioning approach to relational database systems is worth investigation specially when schema evolution in such systems would affect large databases. Studying schema evolution for active databases may be a challenging research direction where events and actions specifications may evolve.

We presented alternative configurations of query graphs for manipulating simple, composite and inter-related objects within the context of the class-versioning approach. We highlighted the advantages and disadvantages of each configuration. A taxonomy of process types required to realize such configurations was presented. We pursued the support of parallel processing for schema evolution where object transformation between class-versions is performed dynamically during run-time. In doing this, we were motivated by the increased number of processes in query graphs that carry out such transformation, and their added processing requirements. We combined multitasking, process replication and associative techniques with pipeline and dataflow processing to improve the performance and reduce the required processing resources of manipulating persistent objects. We highlighted other aspects of parallelism inherent in the class-versioning approach. One aspect is parallel generation of class-version queries from user-query. Another aspect is data partitioning which comes naturally with the class-versioning approach as objects are horizontally split between different class-versions and thus limits the access of class-versions to their affiliated objects. We built a prototype for simple-object manipulation based on the classified process types. The prototype was implemented on a multiprocessor machine using the distributed operating system Equus. The prototype showed a performance advantage of combining multitasking and process replication with pipeline and dataflow processing. Having the ability to dynamically incarnate processes and relocate them on the processing nodes, we were able to increase processor utilization, balance the load across the nodes, and reduce processors idle time. Rather than relocating processes on processing nodes during run time, a task that could prove expensive, we proposed weighting the expected processing time of the processes serving a user-query relatively to each other prior to launching them such that a process (or any of its replicas) with large processing weight is allocated to a processor used by smaller weight processes. We demonstrated the effect of this approach with results of actual runs of the prototype. We employed simulation to study composite and inter-related object manipulation and built different simulation models based on the implementation aspects drawn from the prototype. We conducted simulation experiments involving different system and database parameters settings such as: larger number of processors than that available for the prototype; improvised processing times; processors allocation to different process configurations; speed-up of processing and communication; communication-buffers size; constituent object sharing factor amongst composite objects; process configurations for manipulating simple and composite objects; performing object querying and joining using associative techniques. The experiments produced query service times and processor utilization measurements showing the effect of each variant on the system performance. The experiments indicated a possible performance improvement with the combined processing techniques. The speed-up of associative querying and joining of objects reduced the processing time on node processors of the simulated parallel machine. This, in part, reduces overall object manipulation cost as complex queries and expensive joins become cheap. On the other hand, by reducing the processing time of object selection and join operations, we were able to share node processors between more processes of large query graphs and reduce the required resources for object transformation. Accordingly, implementing the class-versioning approach on machines with limited number of processing nodes would be feasible and practical.

In this research, we carried out performance investigation of the class-versioning approach in general terms without targeting a specific OODBMS. Integrating the approach with an existing OODBMS is a practical step towards evaluating the approach from both functionality and performance points of view. This task would reveal any limitations of the class-versioning approach to cope with the OODBMS and vice versa. Offloading part of the OODBMS activities to a backend parallel machine and distributing those activities amongst processing nodes (e.g., locking and caching instances of class-versions implemented in the prototype) is another challenging area of potential research. With the integration experience, simulation may take into consideration the activities of relevant modules in the OODBMS and realistic processing time of the OODBMS and query processes. The design of the simulation models, on the other hand, considered several system and database features and permit embedding algorithms that need be investigated. So, we regard our simulation models as a *valuable tool for further investigation of processor load balancing schemes, the performance of transactions of mixed query types that would involve simple and composite-object updating and deleting, the impact on system performance of object sharing between multiples of users, object caching methods for large database environment where node processor memory cannot fit all class objects. A future research direction may look into building analytical models to support the query optimizer in deciding how query processes should be replicated and how to distribute them on available processors to achieve load balancing. Such models would process statistical information available in the OODBMS control tables about classes and objects associations with each other.*

The thesis presented the design of a SIMD query-engine that realizes the perceived associative processing speed-up in manipulating simple and composite objects and gives credibility to the conclusions drawn from our simulation experiments. The query-engine is an auxiliary processor to each node processor of multiprocessor machines (rather than being a shared facility) or to the CPU in von-Neumann architectures. It combines the functionality of separate specialized processors proposed for parallel systems and provides features and capabilities that are not in the whole present in any one of the existing designs. We conducted deterministic analysis of query cost and produced a set of formulae for calculating that cost. We presented logic algorithms adapted for multi-word fields, demonstrating the potential of the query-engine in serving different application domains. By achieving tens of times speed-up over RAM-based systems in querying and joining objects, we perceive that the query-engine highlights a potential future research area for associative processing in object-oriented domain.

The current design of the query-engine requires further VLSI investigation to quantify propagation delays within the engine and to provide accurate time measurements for the engine's operations. The investigation should also provide realistic storage capacity estimation for the query-engine, in terms of the number of bytes to be accommodated in a single device. The design of the query-engine controller requires investigation to have an overall view of the query-engine operation and to allow a comprehensive evaluation of the engine's performance in conjunction with existing CPUs. Also, the impact on the engine's and the user-application performance by extending the number of object-delimiter bits per associative word requires further investigation. Adapting mathematical algorithms for the query-engine, such as addition; subtraction; etc. need to be worked-out and be compared in terms of *performance with those designed for sequential operation on von-Neumann architectures and other associative processors. Pattern matching for query predicate such as `Like '%Redwood%'` is worth considering and requires the set-up of an algorithm to resolve it. Extending the algorithms of object joins is also required to cover sets and bags of constituent objects. Comparing the effectiveness of the query-engine with RAM-based systems would then be conducted in that respect.*

This dissertation targeted class versioning in object-oriented databases from both functionality and performance points of view. It highlighted several areas worth investigating when proposing the support of schema evolution. Although we did not specify particular application domains that would best benefit from our research, we think that the outcome of our effort contributes in general to all application domains of existing systems supporting class versioning. We hope that new systems would consider adopting our class-versioning approach and benefit from the parallel processing investigation presented in this thesis.

Appendix A
Query-Engine Design and Adapted Algorithms

This appendix is complementary to the discussion in Chapter 9 regarding the query-engine. Here, we elaborate on the logic design of the engine and its operation. We put more emphasis on the cost of using the engine, from a performance point of view, to resolve database queries and we show how the engine may carry out some of the logical and mathematical algorithms proposed in the literature.

A.1. Query-Engine Structure

In this section, we present the logic circuit design of the main functional elements comprising the query-engine device. We shall also describe the engine layout and its operation.

A.1.1. Circuit Design

As we mentioned earlier in chapter 9, the query-engine incorporates Comparand, Mask and Data cells similar to those of the basic associative memory proposed by Foster [Foster 76]. Figure A.1(a)(b) shows the circuit design of those cells. The data cell is composed of 9 gates which can be reduced to 7 (as opposed to 5 in RAM) if we allow gates O_1 and O_2 to be implemented as wired-ORs. The query-engine also incorporates in each memory word additional associative cells of two types: the ***Object-Delimiter*** type, used to mark the first and the last words in the object, and the ***Field-Delimiter*** type, used to mark the first word of each attribute. The object and field-delimiter cells operate as normal data cells, with additional feature for the field-delimiter cell where its state outputs are combined in the word control circuitry. A memory word may have one field-delimiter cell at the most, but may have more than one object field-delimiter cell depending on whether the object is simple or composite, and on how many constituent-objects expected to occur per composite-object. From the description of the operation of the query-engine that follows, we shall see that accessing constituent-objects can be achieved using a single object-delimiter cell per word to navigate through the engine. However, the engine performance would be further improved if each constituent class has a dedicated object-delimiter cell to locate its instances. The optimum performance is realized when the number of object-delimiter cells matches the number of constituent classes per complex class. Since delimiter words (those having either of their delimiter cells set to '1') are likely to be separated by a number of data words (non-delimiter words), i.e., an attribute is likely to occupy more than one word, so we introduce a minimized version of those cells that comprises none or a single gate (see Figure A.2) depending on the word type, that is: object/field-delimiter or data word. The built-in control unit of the query-engine device decodes input commands into 9 control signals. These are: ***Compare*** (CMP), ***Select Data*** (SD), ***Select Mask*** (SM), ***Select Comparand*** (SC), and [***Set*** , ***Link Next Word*** (LNW), ***Link Previous Word*** (LPW), ***Link Next Field*** (LNF), ***Link Previous Field*** (LPF)][1].

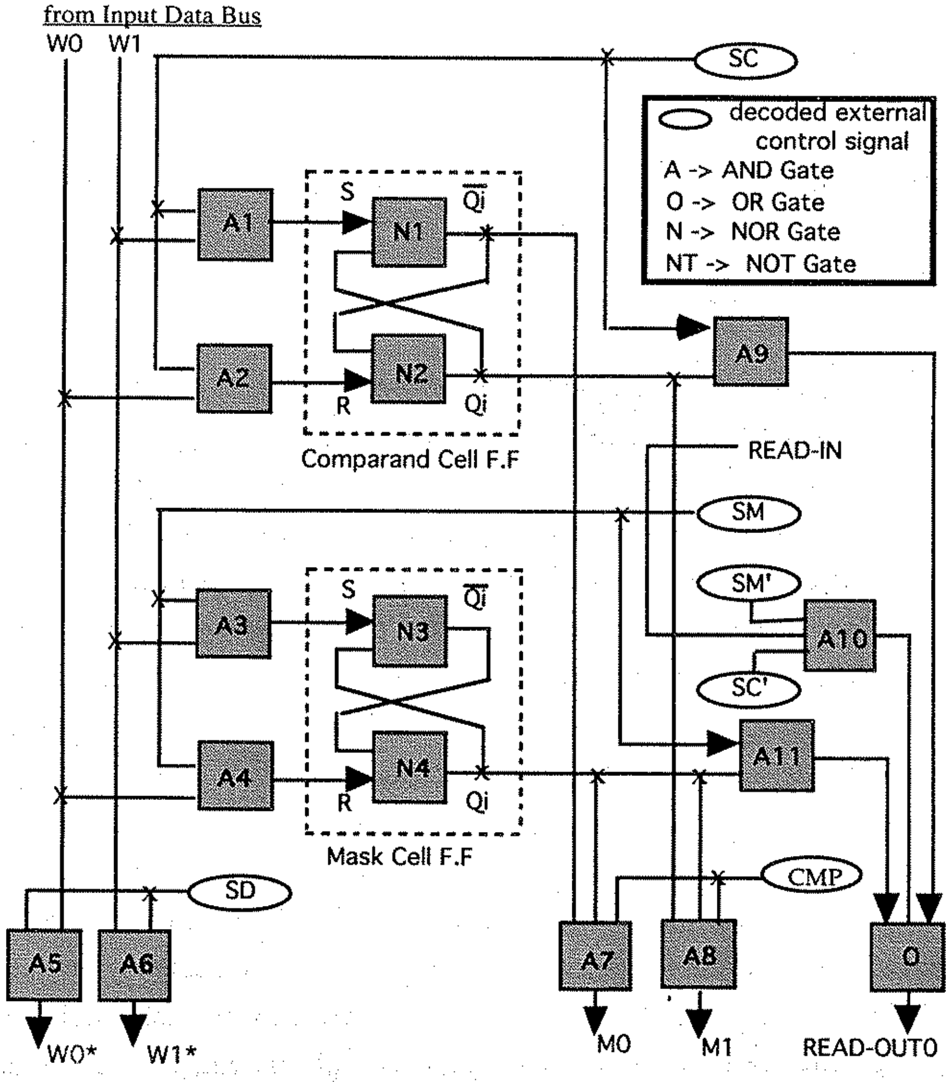


Figure A.1(a): Comparand and Mask Cell Circuitry

[1] Those are mutually exclusive positive-edge pulses.

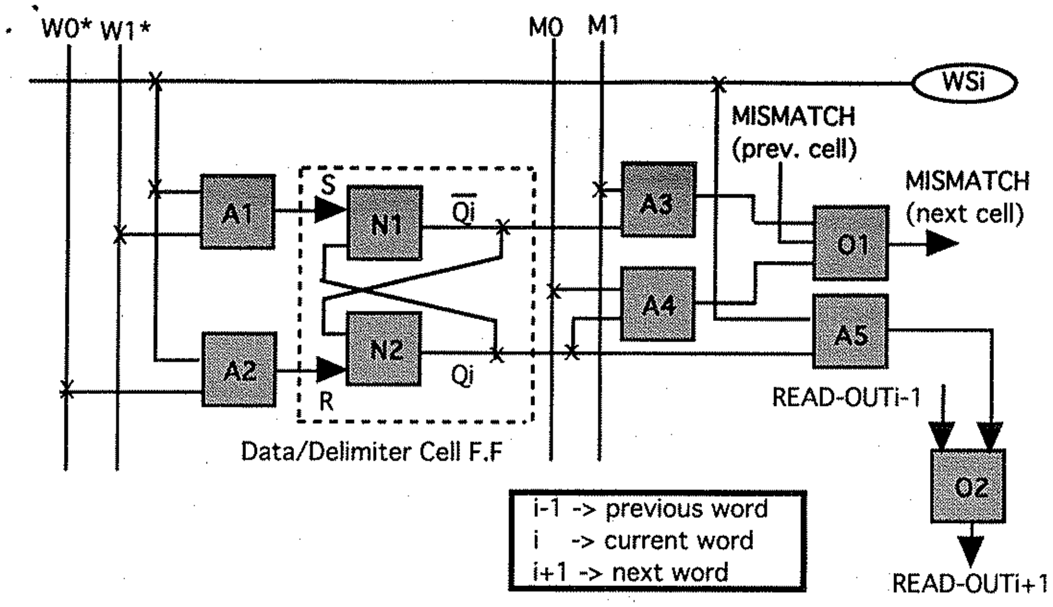


Figure A.1(b): Associative Data/Delimiter Cell Circuitry

The query-engine Tag bits manipulate some of the decoded control signals as well as some internal control signals which link the associative words together. The Delimiter minimization concept also applies to the Tag bits, resulting in two versions comprising 18 and 13 gates (see Figure A.3, Figure A.4). Note that gate O_4 is counted as three 2-input OR gates in Figure A.3 with inputs from A_1 and A_8 being wired-ORed, and is counted for two 2-input OR gates in Figure A.4. Gate O_5 is also counted for two 2-input OR gates. Gate O_6 can be implemented as wired-OR to improve cycle time so that RESULT line can convey the Tag setting to the chip output as fast as possible. Mentioning the RESULT line, the NONE/SOME line does the same function of the former and in addition it controls word selection in Sequential Mode. This incurs a propagation delay of one gate per associative word making the NONE/SOME line much slower to rely upon for checking comparison results.

Based on the previous optimizations, the query-engine can be manufactured with four intermixed types of words as shown in Table A.1. Depending on the distribution of those types of words in the query-engine, the *average control gates overhead* (ACGO) per associative memory word and the *maximum number of unused words* (MNUW) (fragmentation between objects or attributes) can be determined. For example, choosing an organization pattern of one Type-I word followed by seven Type-IV words gives a ACGO of 16 and a MNUW of 7, thus tailoring the organization of the query-engine to suit a particular application.

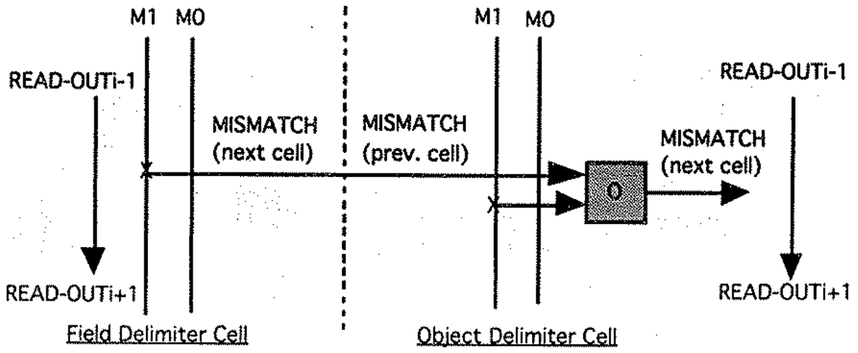


Figure A.2: Modified Delimiter Cell Circuitry

Word Type	Field Delimiter	object Delimiter	Control	Data	Total Word Size
Type I	7	7	18	56	88
Type II	7	1	18	56	82
Type III	0	7	13	56	76
Type IV	0	1	13	56	70

Table A.1: Associative Word Types (all sizes in gates)

A.1.2. Chip Layout

The query-engine can be realized in a reasonable package with the current VLSI technology which can achieve high storage capacity. As depicted in Figures A.5 and 9.3, the query-engine has two data buses to convey data to either the engine's Central Control Unit (CUU), or the associative words. Assuming each bus is 8 lines wide, then to write a single word, 3 steps are required to set the (16) W1, W0 lines of the data cells and the (4) W1, W0 lines of the delimiter cells. On the other hand, reading a single word requires: one step for the data bits, and another optional step (determined by the user application) for the delimiter bits. The query-engine chips can be cascaded as, shown in Figure A.6, by connecting the i+1 lines of each chip to the corresponding i-1 lines of the next. For the first chip in the chain, QF_{i-1} and $ISET_{i-1}$ should be reset to '0', while $NONE/SOME_{i-1}$ lines should be set to '1' and $RESULT_{i-1}$ set to '0'. For the last chip, QB_{i+1} and $ISET_{i+1}$ should be reset to '0'.

Given 6 transistors per gate (non-inverted); 88 gates per associative word, then for a 4096 associative words, we require around 2.2 million transistors that can be fitted on a single device. Scaling the device capacity to larger object-page sizes can also be realized with the current technology. The query-engine design was verified using the simulation package OrCAD run on a Viglen desk-top computer.

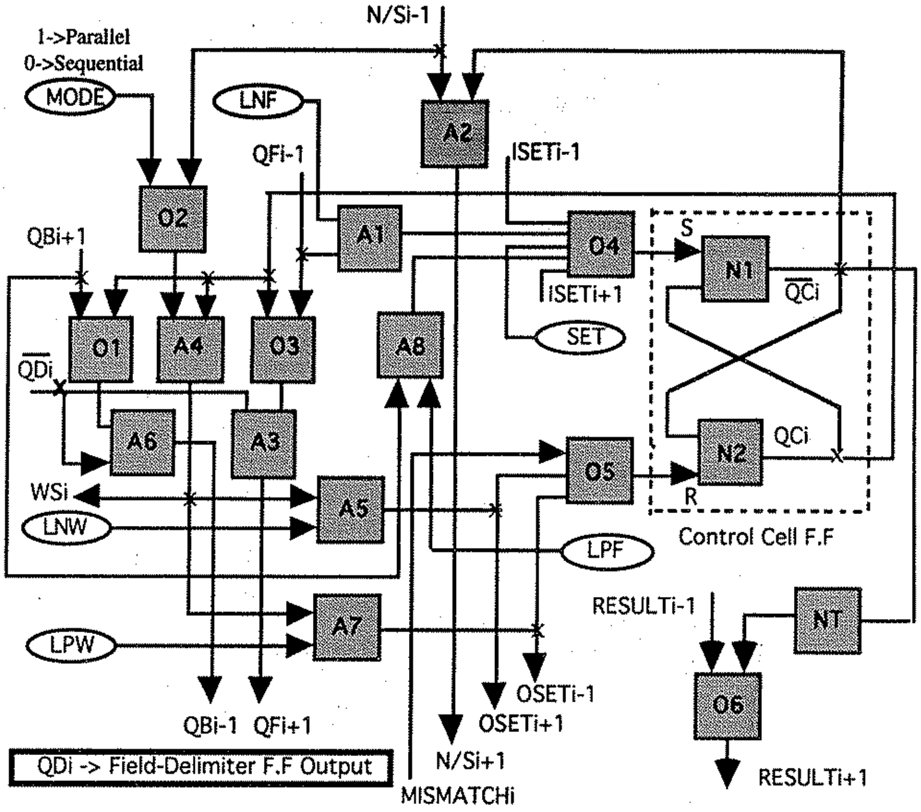


Figure A.3: The Basic Control Circuitry

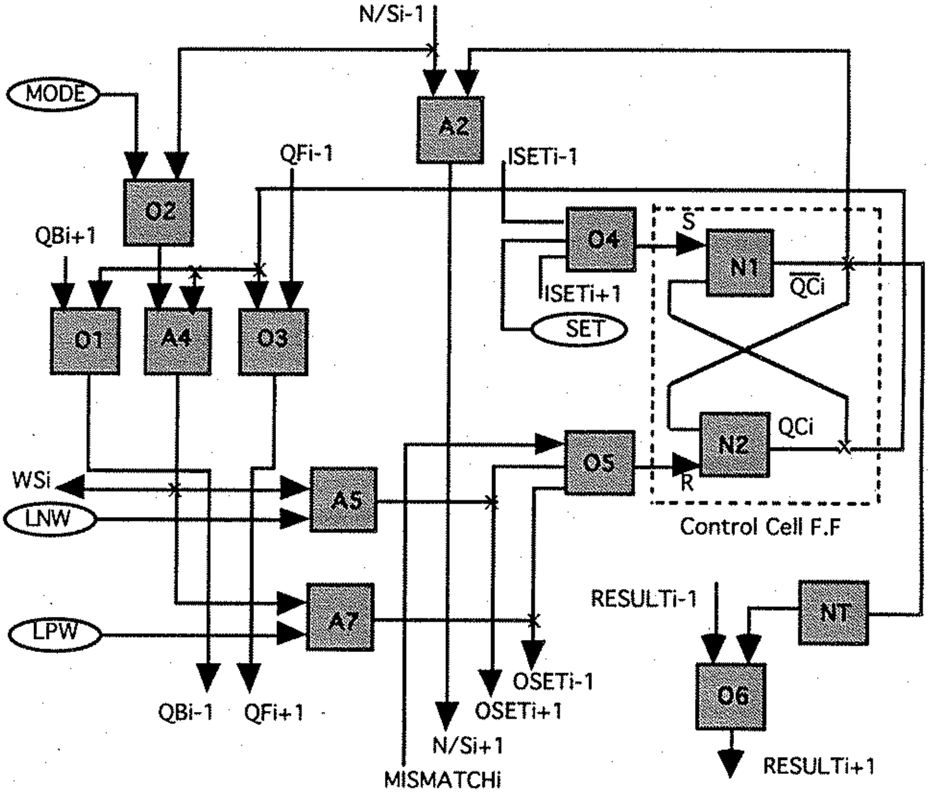


Figure A.4: Modified Control Circuitry

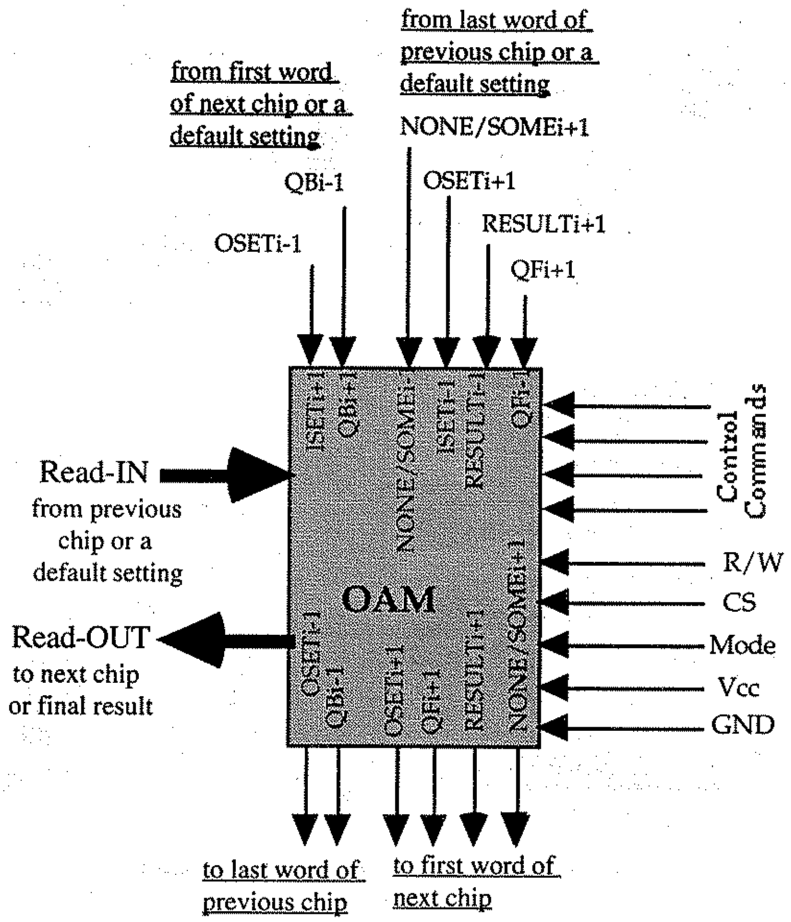

from first word of next chip or a default setting
QBi-1
OSETi-1
from last word of previous chip or a default setting
NONE/SOMEi+1
OSETi+1
RESULTi+1
QFi+1
ISETi+1
QBi+1
NONE/SOMEi-1
ISETi-1
RESULTi-1
QFi-1
Read-IN
from previous chip or a default setting
OAM
Read-OUT
to next chip or final result
OSETi-1
QBi-1
OSETi+1
QFi+1
RESULTi+1
NONE/SOMEi+1
Control Commands
R/W
CS
Mode
Vcc
GND
to last word of previous chip
to first word of next chip

Figure A.5: The Query-Engine Chip Layout

A.2. Operating The Query-Engine

In this section, we describe how object retrieval, update and delete operations are performed in the query-engine. We also elaborate on managing free space in the query-engine that can be a crucial issue if cached persistent objects are to reside in the query-engine during run-time rather than in the processing node's main storage.

The Retrieval Operation

Qualifying objects are retrieved starting with the top-most object and according to the designated direction. Also, the retrieval process can start at any point within selected objects, depending on the requirement of the application, and may proceed in either direction (i.e., forward or backward) depending on the word-linking command used (i.e., LNW or LPW). The output read lines of the query-engine carry the contents of the top-most selected word and the NONE/SOME line will enforce this.

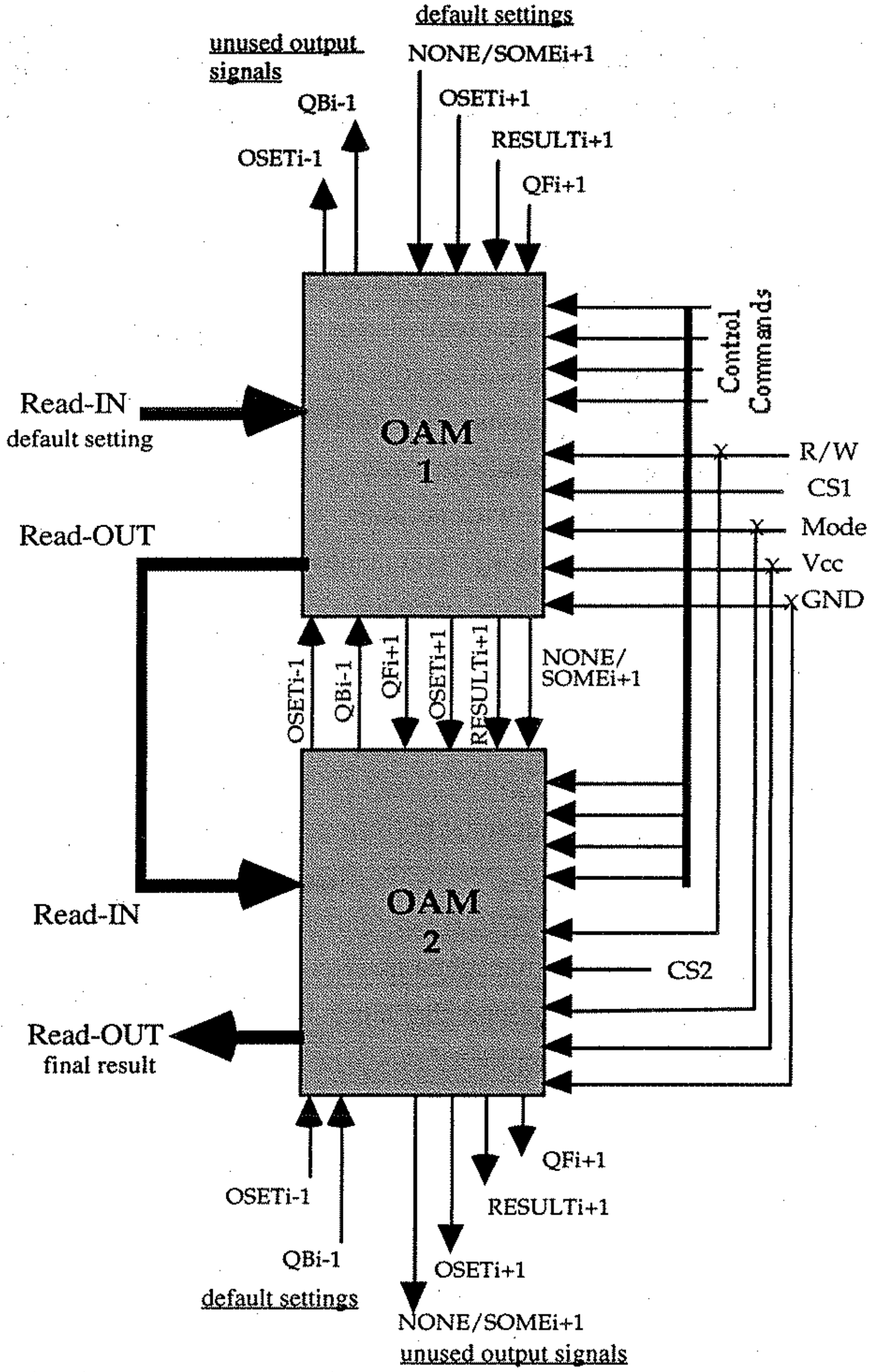


Figure A.6: Cascading Query-Engine Devices

However, the NONE/SOME line has the potential of elongating the retrieval of selected objects due to its incurred delay of activating the next to the top-most selected object. Such delay would be of a variable amount depending on the distance (counted by associative words) separating both objects. So, instead of accessing objects at fixed intervals (equal to the maximum possible delay), the query-engine controller may sense the setting of the next object-delimiter at the output lines, then resumes its activities. This situation means that the query-engine would provide better performance as the number of selected objects increases.

The Update Operation

The query-engine has two modes for the write operation: single-word (Sequential Mode) and multi-word (Parallel Mode). The first mode is realized as long as the Mode line is reset to '0' during the write operation. In this case, only the top-most selected word will be affected. By activating the Mode line (setting it to '1'), the second mode comes in effect. Depending on the control command issued, namely SD; SM; or SC, the write operation is directed to the required cells. Loading the query-engine with an object-page is executed by first selecting all associative words using the SET command. While in the Sequential Mode, data is written to the top-selected word, then the word is de-selected (by comparing it with an illegal value). The latter sequence of operations would be repeated for each byte of object data.

The Delete Operation

Deleting selected object(s)can be performed by writing a special bit pattern in the object header (for example '0's in the first word). Object's freed space can be referenced by the object's own ID or by replacing it with a special ID introduced specifically for memory management. Removing an attribute from the database schema can be enforced by screening attribute's data when accessing objects. Later reorganization of query-engine storage can be performed to reclaim unused space within objects.

Managing Free Space

The query-engine should be initialized prior to loading it with data. During initialization, all words are written to '0's while Object and Field-Delimiter Cells are set to '1's. So, by selecting the first empty object-delimiter word (containing '0's in its data cells) in the query-engine, and while in the sequential-mode, each word of the object header can be written with SD, followed by a LNW command to select the next empty word. To insert a new field, the LNF command should be issued to select the first available field-delimiter word, then followed by a SD and LNW to insert each of the field's data words. In this way, the free space is always kept at the bottom of the query-engine as a contiguous area. In update-intensive applications, relocating objects due to their need of acquiring extra space is possible. This would require freeing previous space occupied by the object. Such mode of operation would cause fragmentation of free space. Reallocating freed object space to other objects is possible by introducing a management technique for free space. One example of such techniques is the binary buddy system [Knuth 68] which was adopted for the object memory in [Hyatt 93]. It might also be required to compact free space if memory utilization falls below a certain threshold while fragments become too small to accommodate any object. This in turn requires reorganising the objects in the query-engine.

A.3. Estimating Operation Cost

We present in this section deterministic analysis of the execution cost of database queries and different adapted mathematical and logical algorithms. Our analysis is based on the elementary operation cost depicted in Table A.2. We shall use those operations in producing synthetic formula for query cost. We assume the following:

Target attribute position (i.e., sequence) in object	$P \geq 1$.
Attribute size (in terms of associative words)	$S \geq 1$.
Word position in attribute (excluding attribute header)	$W \geq 1$.
Object size (in terms of associative words)	$T \geq 3$.
Average interrogand size (in terms of associative words)	$Z \geq 0$
Average number of forward-processed interrogands in the query	$F \geq 0$
Average number of backward-processed interrogands in the query	$B \geq 0$
Average number of interrogands in the query	$I = F + B$
Average distance between interrogands within object (in terms of fields)	$D \geq 1$
Average number of qualifying objects	$N \geq 0$

Examples of recent (i.e., the 90's) fully-parallel devices showed a diversity of cycle times due to the adoption of different technologies. Shin et al [Shin 92] produced a 256-word x 8-bit CAM for real-time image processing with 50 ns cycle time. Tamura et al. [Tamura 90] produced a 64-word x 20-bit CAM for virtual memory management that operates at 3.6 ns per address translation. Bergh et al [Bergh 90] produced a 128-word x 64-bit fault-tolerant CAM device operating at 60 ns cycle time. Accordingly, we chose to set a general estimate of the query-engine's cycle time that is not biased towards a particular technology. Given a gate propagation delay of 1 nanosecond (which is moderate) and 5 gate levels on the average per operation, we estimate a basic operation to take 5 nanoseconds (that is the engine's cycle time). However, the formulas deduced later on refers to the basic operation cost in *units* as a general estimate which can be scaled or translated to the exact cycle time proven by the implementation.

Operation Description	**Code**	**Cost (units)**
Any command to the Central Control Unit (CUU)	CM	1
Comparing data word/bit in associative words (Compare) = CM (Write Comparand + Compare) = 2 * CM (Write Comparand + Write Mask + Compare) = 3 * CM	 CW_1 CW_2 CW_3	 1 2 3
Selecting object header (Set + Compare) = CM + CW	SO	4
Moving activity (link) to next/previous word/field (LNW/LNF/LPW/LPF) = CM	MA	1
Selecting target attribute header (starting from object header) (P * [LNW + LNF + Compare]) = P * [CM +CM + CW_3]	SA	5P
Updating intermediate query-result (Write to attribute header) = CM	UF	1
Selecting target word (starting from target attribute header) (W * MA)	SW	W
Selecting target attribute header (starting from target word) (W * MA)	SH	W

Table A.2: The Cost of Elementary Operations

We decompose queries into their basic modular operations and estimate the cost of each operation based on the query-engine elementary activities described earlier. We use the brackets [] to indicate repeated activities within query operations. In some cases, rewriting the Mask may be eliminated, which results in less cost than what is estimated here.

- **Object Header Processing**

Set	Compare (class-ID)	LNW	Compare (class-version ID)
CM	CW_3	CM	CW_3

8 units

- **Locate the Header of each interrogand starting from current attribute**

Write (Comparator)	Write (Mask)	[LNF]	[Compare] (field-delimiter)	[LNW]
CM	CM	D*CM	D*CW_1	(D-1)*CW_1

3*D + 1 units

- **Locate the Trailer of each interrogand starting from its Header**

LNW	LNF	Compare (field-delimiter)	LPW
CM	CM	CW_1	CM

4 units

- **Compare interrogand (forward traversal)**

Write (Mask)	[Write(Comparator) + Compare(Assoc. Word)]	[LNW]
CM	Z*CW_2	(Z-1)*CM

3*Z units

- **Compare interrogand (backward traversal)**

Write (Mask)	[Write(Comparator) + Compare(Assoc. Word)]	[LPW]
CM	Z*CW_2	(Z-1)*CM

3*Z units

- **Link-back to object-header starting from last interrogand**

Write (Comparator)	Write (Mask)	[LPF]	[Compare] (field-delimiter)	[LPW]	Compare (object-header)
CM	CM	I*(D+1)*CM	I*D*CW_1	I*D*CM	CW_3

I*(3*D + 1) + 5 units

- **Link-back to attribute-header starting from current attribute word**

LPF	Compare (field-delimiter)
CM	CW_3

4 units

- **Retrieve qualifying objects (forward traversal)**

Write (Comparator)	Write (Mask)	[LNW]	[Compare] (illegal setting)
CM	CM	N*T*CM	N*T*CW_1

2*(N*T + 1) units

- **Resolve a query on simple objects or clustered complex objects**

Process Object Header	I* [Locate Interrogand Header]	B* [Locate Interrogand Trailer]	F* [Compare Interrogand Forward]	B* [Compare Interrogand Backward]	Link Back Object Header	Retrieve Qualifying Objects

15 + I*(6*D + 3*Z + 2) + 4*B + 2*N*T units

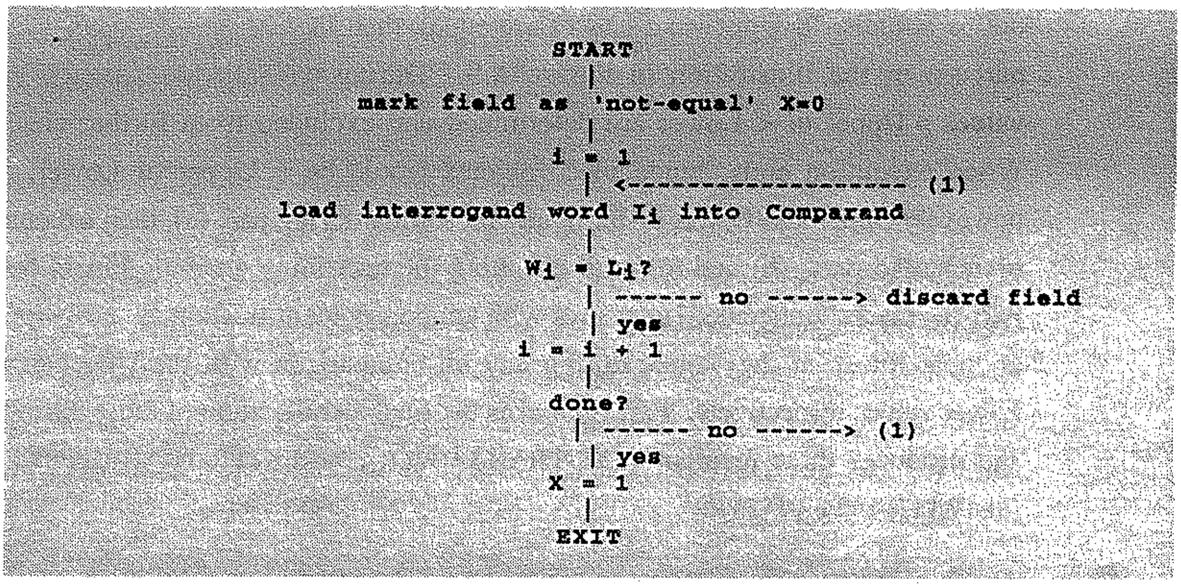
The delay incurred in propagating NONE/SOME signal between selected objects is equal to the number of associative words separating those objects multiplied by the gate delay (1 nanosecond approximately). This delay needs to be added to the overall query cost.

A.4. Adapted Algorithms

In Chapter 9, we demonstrated the application of the query-engine in querying both simple and composite objects. In doing so, we highlighted the advantage of using the associative techniques in restricting sets of objects in parallel. Compared to existing associative processors, the query-engine provides better navigation capabilities which lead to faster accessibility to object attributes and reduce reliance on storing temporary query results. In this section, we demonstrate the ability of the query-engine to execute logic algorithms giving it a wider range of applications. Our presentation will be based on the logical algorithms introduced by Foster [Foster 76] and adapted for associative processors. The main feature of Foster's algorithms is that they were based on single-word attributes. Here, we extend some of those algorithms to operate on multi-word attributes in parallel-by-word mode. The algorithms are: ***Exact Match*** , ***Compare Magnitude with Comparand*** , ***Five-Way Split.*** , and ***Maximum-Minimum***.

Exact-Match

In this algorithm (see Figure A.7), we compare each word (W) in the attribute with the corresponding interrogand word (I) loaded in the Comparand register. The comparison is done in parallel along all objects in memory. We assume that the result of this operation is to be recorded in a bit (x) within a reserved word in each object. At the end of the comparison cycle, matching attributes will have the Tag bit of their last word set to '1'. An attribute that contains any mismatching word with the interrogand will not have any of its Tag bits set to '1' and is discarded from subsequent comparisons . Subsequently, matching objects will have their x-bit set to '1' while other objects will have their x-bit set to '0'.



```
                         START
                           |
           mark field as 'not-equal' X=0
                           |
                         i = 1
                           | <------------------- (1)
      load interrogand word Ii into Comparand
                           |
                       Wi = Li?
                           | ------ no ------> discard field
                           | yes
                       i = i + 1
                           |
                         done?
                          | ------ no ------> (1)
                           | yes
                         X = 1
                           |
                         EXIT
```

Figure A.7: The Exact-Match Algorithm

Note: The 'discard field' operation is performed by the associative word circuitry, and results in resetting the word's Tag bit to '0'.

Compare Magnitude with Comparand **(Three-Way Split)**

This algorithm determines if a multi-word attribute is 'EQUAL-TO', 'LESS-THAN', or 'GREATER-THAN' a given interrogand. Each word (W) of the attribute is compared with the corresponding inverted interrogand word (I), one bit at a time using the Mask register to enforce this mode. Here, we assign two bits (x,y) within each object to hold the temporary results of comparisons. Only words of an 'EQUAL-TO' attribute are checked against the Comparand. Decided attributes, those marked as 'LESS-THAN' or 'GREATER-THAN', are excluded from subsequent checks. The LNW command is used to point to the word due to be checked, while the LPF is used to point-back to the result word at the field (i.e., attribute) header. Figure A.8 illustrates algorithm steps.

Five-Way Split

This algorithm is an extension to the Three-Way Split algorithm. Here, we compare the attribute against two boundary values, an upper and a lower value. In the algorithm, we need to determine if a multi-word attribute is 'EQUAL-TO-UPPER', 'EQUAL-TO-LOWER', 'LESS-THAN-LOWER', 'GREATER-THAN-UPPER', or 'BETWEEN-UPPER-LOWER'. The algorithm, shown in Figure A.9, relies on the query-engine controller to load the inversion of both the upper and the lower boundary values (separately) into the Comparator to determine if the attribute is out-of-bounds. If Upper and Lower are equal, then the algorithm executes only part-(a) of the illustration in Figure A.9. During execution, attributes with values > Upper or < Lower are determined and excluded from further checks. When execution ends, 'undecided' objects would be = Upper or Lower. If, however, Upper and Lower differ in any bit setting during part-(a), then execution branches to parts-(b) and (c) until checking the remaining attribute bits is done. Part-(b) marks 'undecided' attributes with temporary identification as 'near-upper' or 'near-lower' if they agree during any bit comparison with the Upper or the Lower values respectively. Part-(c) makes the final identification of those attributes and produces three results: 'LESS-THAN-LOWER', 'GREATER-THAN-UPPER', or 'BETWEEN-UPPER-LOWER'.

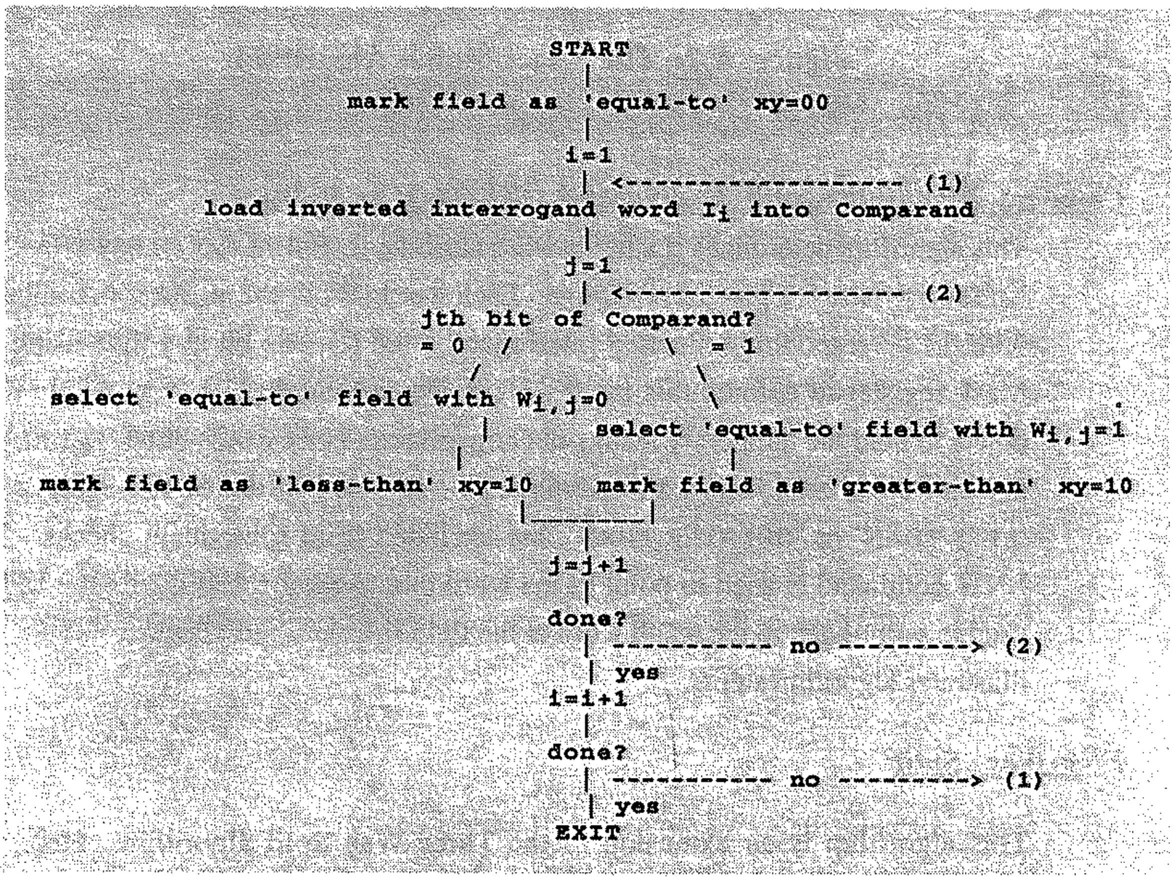

START
mark field as 'equal-to' xy=00
i=1
<------------------- (1)
load inverted interrogand word I_i into Comparand
j=1
<------------------- (2)
jth bit of Comparand?
= 0
= 1
select 'equal-to' field with $W_{i,j}$=0
select 'equal-to' field with $W_{i,j}$=1
mark field as 'less-than' xy=10
mark field as 'greater-than' xy=10
j=j+1
done?
----------- no ---------> (2)
yes
i=i+1
done?
----------- no ---------> (1)
yes
EXIT

Figure A.8: Three-Way Split Algorithm

If the result bits (x,y,z) are maintained in the attribute header word, then the 'mark field' operations would have fixed cost as navigation from any word within an attribute and its header is performed with a single LPF and a Write Data command, costing 2 units. The 'load word' and the 'select' operations, on the other hand, are realized by a Write to the Comparand and a Compare commands respectively, costing 1 unit each. Based on this description, we can make rough estimate of the cost of this algorithm which is 12 units for each bit of the interrogand, and 12 * (Z * 8) units per query. We excluded the cost of tasks performed by the query-engine controller (e.g., inverting the interrogands, and keeping track of loop iterations with i, j).

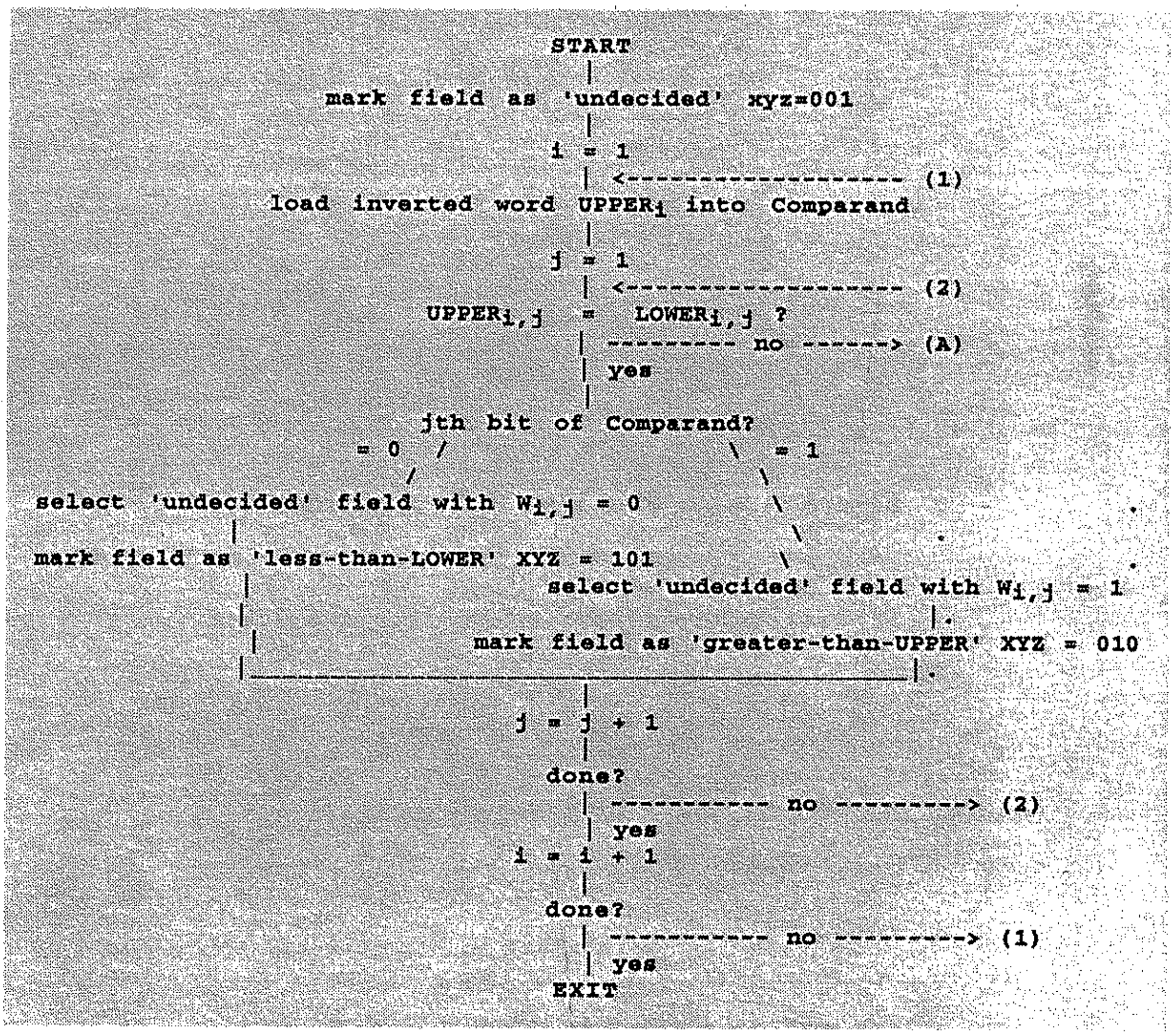
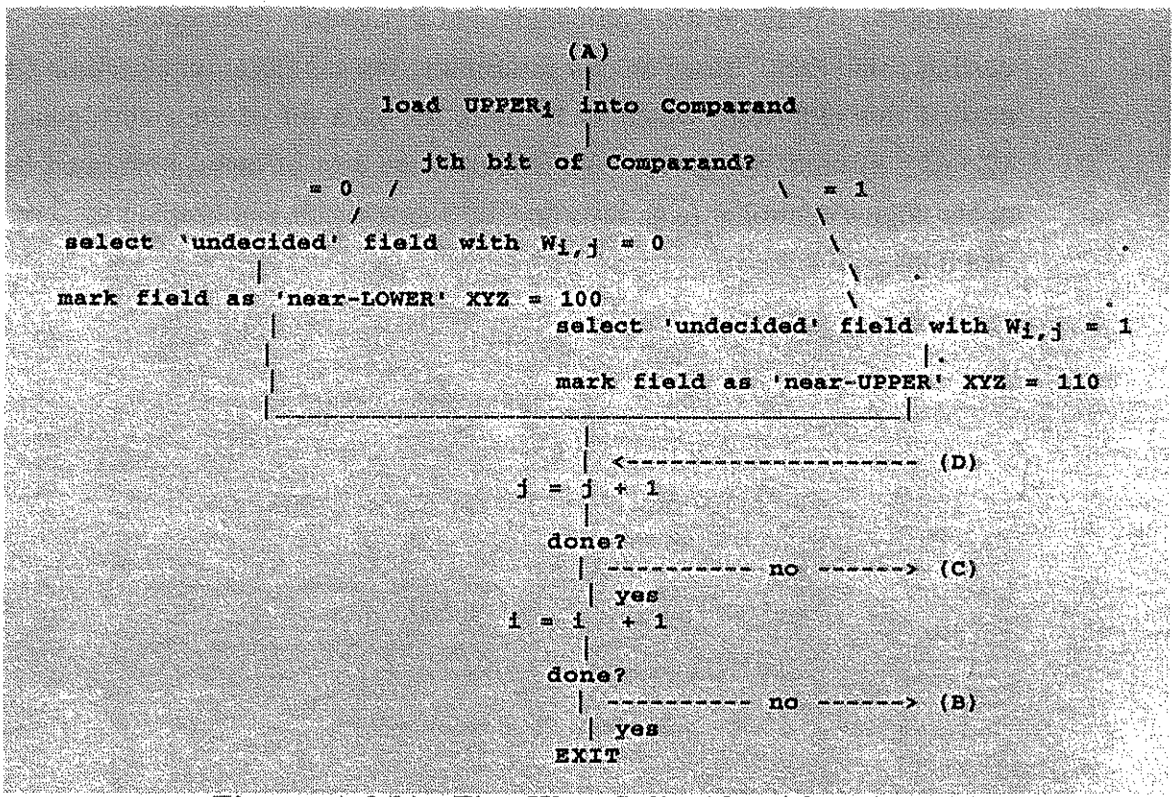
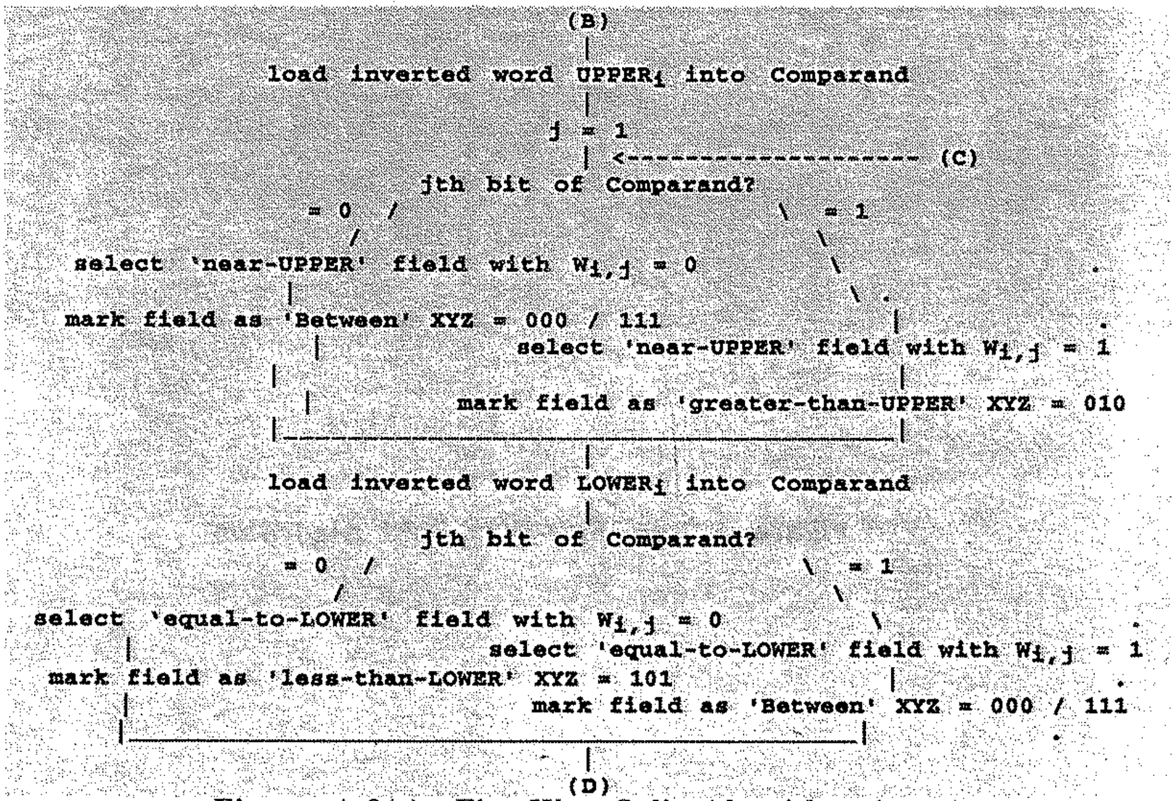

START
mark field as 'undecided' xyz=001
i = 1
<------------------ (1)
load inverted word UPPER$_i$ into Comparand
j = 1
<------------------ (2)
UPPER$_{i,j}$ = LOWER$_{i,j}$?
--------- no ------> (A)
yes
jth bit of Comparand?
= 0
= 1
select 'undecided' field with W$_{i,j}$ = 0
mark field as 'less-than-LOWER' XYZ = 101
select 'undecided' field with W$_{i,j}$ = 1
mark field as 'greater-than-UPPER' XYZ = 010
j = j + 1
done?
---------- no --------> (2)
yes
i = i + 1
done?
---------- no --------> (1)
yes
EXIT

Figure A.9(a): Five-Way Split Algorithm



```
                                (A)
                                 |
                  load UPPERi into Comparand
                                 |
                     jth bit of Comparand?
              = 0  /                              \  = 1
                  /                                \
 select 'undecided' field with Wi,j = 0             \
              |                                      \
mark field as 'near-LOWER' XYZ = 100                  \
               |                select 'undecided' field with Wi,j = 1
               |                                          |
               |                mark field as 'near-UPPER' XYZ = 110
               |__________________________________________|
                                 |
                                 | <------------------- (D)
                             j = j + 1
                                 |
                               done?
                                 | ---------- no ------> (C)
                                 | yes
                             i = i  + 1
                                 |
                               done?
                                 | ---------- no ------> (B)
                                 | yes
                                EXIT
```

Figure A.9(b): Five-Way Split Algorithm (cont.)



```
                                (B)
                                 |
            load inverted word UPPERi into Comparand
                                 |
                               j = 1
                                 | <------------------- (C)
                     jth bit of Comparand?
              = 0  /                              \  = 1
                  /                                \
 select 'near-UPPER' field with Wi,j = 0            \
              |                                      \
mark field as 'Between' XYZ = 000 / 111               |
                |             select 'near-UPPER' field with Wi,j = 1
              |                                       |
                |          mark field as 'greater-than-UPPER' XYZ = 010
              |_______________________________________|
                                 |
            load inverted word LOWERi into Comparand
                                 |
                     jth bit of Comparand?
              = 0  /                               \  = 1
                  /                                 \
select 'equal-to-LOWER' field with Wi,j = 0          \
      |                      select 'equal-to-LOWER' field with Wi,j = 1
mark field as 'less-than-LOWER' XYZ = 101              |
      |                        mark field as 'Between' XYZ = 000 / 111
      |________________________________________________|
                                 |
                                (D)
```

Figure A.9(c): Five-Way Split Algorithm (cont.)

Maximum-Minimum Algorithm

In this algorithm, we attempt to locate the object(s) with maximum or minimum value for a multi-word attribute. To pick the maximum attribute value, the algorithm compares each bit of the attribute, first, with value '1'. If any object qualifies, its result bit (x) is left set to '1', while disqualified objects have their x-bit reset to '0' and are discarded from later checks. If no object qualifies for a comparison operation, no changes are made to their x-bits. That, in effect, excludes the comparison bit from any selections. At the end of algorithm execution, the object(s) the maximum value will have its (their) objects marked with x-bit = 1.

Note: The 'none' and 'some' states shown in Figure A.10 are determined by the RESULT or NONE/SOME line.

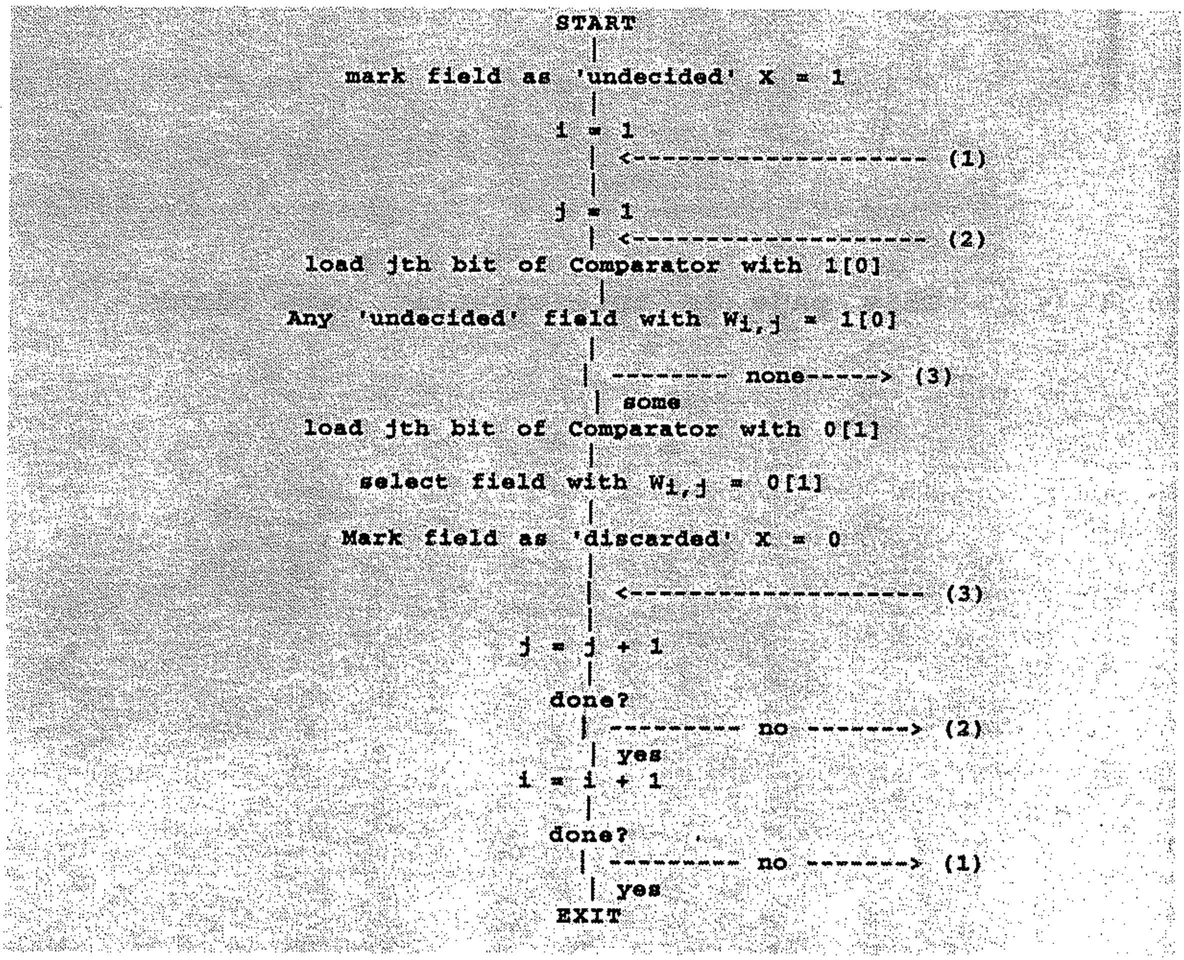

START
mark field as 'undecided' X = 1
i = 1
<------------------- (1)
j = 1
<------------------- (2)
load jth bit of Comparator with 1[0]
Any 'undecided' field with $W_{i,j}$ = 1[0]
-------- none-----> (3)
some
load jth bit of Comparator with 0[1]
select field with $W_{i,j}$ = 0[1]
Mark field as 'discarded' X = 0
<------------------- (3)
j = j + 1
done?
--------- no -------> (2)
yes
i = i + 1
done?
--------- no -------> (1)
yes
EXIT

Figure A.10: Maximum-Minimum Algorithm

The values between square brackets [] indicate the alternate Comparator setting for the Minimum algorithm. The value outside the brackets indicate the case of Maximum algorithm.

This page is intentionally left blank

Bibliography

[Ahad 88] Ahad, R., "*The Object Shell: An Extensible System to Define an Object-Oriented View of an Existing Database* ", Advances in Object-Oriented Database Systems, 2nd International Workshop on Object-Oriented Database Systems, Springer-Verlag, September 1988, pp. 174-192.

[Atkinson 78] Atkinson, M., "*Programming Languages and Databases* ", Technical Report CSR-26-78, University of Edinburgh, August 1978.

[Atkinson 89] Atkinson, M., *et al.*, "*The Object-Oriented Database System Manifesto* ", Report Technique *Altäir* 30-89, 21 aoüt 1989.

[Bancilhon 92] Bancilhon, F., "*Understanding Object-Oriented Database Systems* ", EDBT'92 Advances in Database Technology, Vienna, Austria, Springer-Verlag, March 1992, pp. 1-9.

[Banerjee 87] Banerjee, J., *et al.*, "*Semantics and Implementation of Schema Evolution in OODBS* ", Proceedings of ACM Special Interest Group on Management of data, San Francisco, 1987, pp. 311-322.

[Batcher 74] Batcher, K., "*STARAN Parallel Processor System Hardware* ", Proceedings of National Computer Conference, 1974, pp. 405-410.

[Berg 72] Berg *et al.*, "*PEPE - An Overview of Architecture, Operation, and Implementation* ", Proceedings of National Electronics Conference, 1972, pp. 312-317.

[Bergh 90] Bergh, H., *et al.*, "*A Fault-Tolerant Associative Memory With High-Speed Operation* ", IEEE Journal of Solid-State Circuits, Vol.25, No.5, August 1990, pp. 912-919.

[Bershad 88] Bershad, B., *et al.*, "*PRESTO : A System for Object-Oriented Parallel Programming* ", Technical Report 87-09-01, Department of

Computer Science, University of Washington, September 1987, Revised January 1988.

[Bertino 91] Bertino, E., Martino, L., "*Object-Oriented Database Management Systems: Concepts and Issues* ", *Computer* , IEEE Computer Society, Vol.24, No.4, April 1991, pp. 37-47.

[Bertino 92] Bertino, E., "*A View Mechanism for Object-Oriented Databases* ", Advances in Database Technology - EDBT'92 International Conference on Extending Database Technology, Vienna, Austria, Springer-Verlag, March 1992, pp. 136-151.

[Bertino 93a] Bertino, E., Martino, L., "*Object-Oriented Database Systems: Concepts and Architectures* ", Addison-Wesley, 1993.

[Bertino 93b] Bertino, E., Guglielmina, L., "*Path-index: An Approach to the efficient execution of object-oriented queries* ", Data and Knowledge Engineering, North Holland, 10 (1993), pp. 1-27.

[Bjornerstedt 88] Bjornerstedt, A., Britts, S., "*AVANCE: An Object Management System* ", Proceedings of OOPSLA'88, 1988, pp. 206-221.

[Bjornerstedt 89] Bjornerstedt, A., Hulten, C., "*Version Control in an Object-Oriented Architecture* ", Object-Oriented Concepts, Applications and Databases, Edited by W. Kim and F. Lochovsky, Addison-Wesley, 1989, pp. 451-486.

[Blasgen 77] Blasgen, H., Eswaran, K. P., "*Storage and Access in Relational Data Bases* ", IBM Systems Journal 16, 4 (1977), pp. 363-378.

[Booch 91] Booch, G., "*Object Oriented Design with Applications* ", Benjamin/Cummings Publishing Company, 1991.

[Boral 81] Boral, H., DeWitt, D., "*Processor Allocation Strategies for Multiprocessor Database Machines* ", ACM Transaction on Data Base Systems", Vol.6, No.2, June 1981, pp. 227-254.

[Bretl 89] Bretl, R., *et al.*, "*The GemStone Data Management System* ", in *Object-Oriented Concepts, Databases and Applications*, edited by

W. Kim and F. Lochovsky, Reading, MA, Addison-Wesley, 1989, pp. 283-308.

[Brown 91] Brown, A., "*Object-Oriented databases-Applications in Software Engineering* ", McGraw-Hill, 1991.

[Budd 91] Budd, T., "*An Introduction to Object-Oriented Programming* ", Addison-Wesley, 1991.

[Butterworth 91a] Butterworth, P., "*ODBMSs as Database Managers* ", JOOP, February 1991, pp. 44-46.

[Butterworth 91b] Butterworth, P., "*ODBMSs as Database Managers* ", JOOP, May 1991, pp. 47-50.

[Byeon 93] Byeon, K., McLeod, D. , "*Towards the Unification of Views and Versions for Object Databases* ", Proceedings of the First JSSST International Symposium on Object,Technologies for Advanced Software, Springer-Verlag, November 4-6, 1993, pp. 220-236.

[Byte 89] "*Equus : A Parallel Operating System* ", Byte, September 1989, pp. 80LS-3:8.

[Carey 86] Carey, M., *et al.*, "*Object and File Management in the EXODUS Extensible Database System* ", Proceedings of the 12th International Conference on Very Large Databases VLDB'86, Kyoto, Japan, August 1986.

[Carey 93] Carey, M., *et al.*, "*The OO7 Benchmark* ", Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, DC, May 26-28, 1993, pp. 12-21.

[Casais 91] Casais, E., "*Managing Evolution in Object Oriented Environments: An Algorithmic Approach* ", Ph.D. Dissertation, Université de Genève, 1991.

[Cattell 88] Cattell, R., "*Object-Oriented Database Management Systems Performance Measurement* ", EDBT'88 Advances in Database Technology, Italy, Springer-Verlag, March 1988, pp. 364-367.

[Cattell 92] Cattell, R., Skeen, J., "*Object Operations Benchmark* ", ACM Transaction on Data Base Systems, Vol.17, No.1, March 1992, pp. 1-31.

[Cesarini 87] Cesarini, F., *et al.*, "*A procedural Strategy for Database Machine Analysis* ", in Database Machine Performance: Modelling Methodologies and Evaluation Strategies, edited by: F. Cesarini and S. Salza, Springer-Verlag, 1987, pp. 95-128.

[Chalmers 89] Chalmers, M., "*An Object-Oriented Style for The Computer Surface* ", Proceedings of OUG-11 Occam User Group 11th Technical Meeting, Edinburgh, Scotland, September 1989.

[Choi 89] Choi, C., *et al.*, "*Generic Associative Memory for Information Retrieval* ", Proceedings of the Sixth International Workshop on Database Machines, IWDM'89, France, June 1989, pp. 202-214.

[Chorafas 93] Chorafas, D., Steinmann, H., "*Object-Oriented Databases* ", Prentice Hall, 1993.

[Chou 88] Chou, H., Kim, W., "*Versions and Change Notification in an Object-Oriented Database System* ", Proceedings of the 25th Design Automation Conference, June 1988.

[Clamen 92] Clamen, S., "*Class Evolution and Instance Adaptation* ", Technical Report CMU-CS-92-133, Carnegie Mellon University School of Computer Science, Pittsburgh, PA, June 1992.

[Clamen 94] Clamen, S., "*Schema Evolution and Integration* ", Distributed and Parallel Databases, Kluwer Academic Publishers, Boston, 2 (1994), pp. 101-126.

[Dasgupta 89] Dasgupta, S., "*Computer Architecture - A Modern Synthesis* ", Volume 2: Advanced Topics, John Wiley & Sons, 1989.

[De Groat 92] De Groat, D., *et al.* , "*Issues in Parallelizing OODB Systems* ", in Parallel Processing and Data Management, edited by P. Valduriez, Chapman & Hall, 1992, pp. 195-206.

[DeCegama 89] DeCegama, A., "*Parallel Processing Architecture and VLSI Hardware* ", Prentice-Hall, Inc., 1989.

[Duhl 88] Duhl, J., Damon, C., "*A Performance Comparison of Object and Relational Databases Using the Sun Benchmark* ", OOPSLA'88, 1988, pp. 153-163.

[El-Sharkawi 90] El-Sharkawi, M., *et al.*, "*Object Migration Mechanisms to Support Updates in Object-Oriented Databases* ", PARBASE-90 International Conference on Databases, Parallel Architectures and their Applications, IEEE Computer Society Press, 1990, pp. 378-387.

[Equus] "*Using Equus: A Guide to the EQUUS Parallel Processing System V.3.1* ", Zebra Parallel Ltd, UK.

[Ertem 89] Ertem, M., "*Multiple Operation Memory Structures* ", 22nd Annual International Workshop on Microprogramming and Microarchitecture, Dublin, Ireland, August 14-16, 1989, pp. 181-185.

[Fathy 91] Fathy, S., "*Exploring Parallelism with an OODB* ", Ph.D. Dissertation, University of Kent, 1991.

[Fishman 89] Fishman, D., "*An Overview of the Iris DBMS* ", in Object-oriented Concepts, Databases, and Applications, W. Kim and F. Lochovsky (Editors), Addison-Wesley, Reading, Mass, 1989, pp. 219-250.

[Foster 76] Foster, C., "*Content Addressable Parallel Processors* ", Van Nostrand Reinhold Company, 1976.

[Gardarin 81] Gardarin, G., "*An Introduction to SABRE: A Multiprocessor Database Machine* ", Proceedings of the 6th Workshop on Computer Architecture for Nonnumeric Processing, Hyeres, France, June 1981.

[Gorlen 91] Gorlen, K., *et al.*, "*Data Abstraction and Object-Oriented Programming in C++* ", John Wiley & Sons, 1991.

[Graefe 88] Graefe, G., Maier, D., "*Query Optimization in Object-Oriented Database Systems* ", EDBT'88 Advances in Database Technology, Italy, Springer-Verlag, March 1988, pp. 358-363.

[Gray 92] Gray, P., *et al.*, "*Object-Oriented Databases, A Semantic Data Model Approach* ", Prentice Hall, 1992.

[Härder 92] Härder , T., *et al.*, "*Query Processing for Complex Objects* ", Data and Knowledge Engineering, North-Holland, 7 (1992) 181.

[Harland 86] Harland, D., "*REKURSIV : An Architecture for Artificial Intelligence* ", Proceedings "AI Europe", Wiesbaden, Gunn. Pringle & Beloff, September 1986.

[Harland 88] Harland, D., "*REKURSIV : Object-Oriented Computer Architecture* ", Ellis Horwood Ltd, 1988.

[Harp 89] Harp, G., "*Transputer Applications* ", Pitman Publishing, 1989.

[Higuchi 91] Higuchi, T., *et al.*, "*IXM2: A Parallel Associative Processor* ", The 18th Annual International symposium on computer Architecture, Toronto, Canada, May 27-30, 1991, pp. 22-31.

[Hwang 85] Hwang, J., Humphrey, F., "*An Associative Memory That Breaks The Hardware Barrier* ", Electronics, December 16, 1985, pp. 39-41.

[Hyatt 93] Hyatt, C., "*A high Performance Object-Oriented Memory* ", Computer Architecture News, ACM Publication, Vol.21, No. 4, September 1993, pp. 11-19.

[IBM 87] IBM Manual, "*SQL/Data System Diagnosis Reference for VSE* " , Version 2 Release 1, LH09-8041-00 (IBM Canada, 1987).

[INMOS 88] INMOS Limited, "*Occam 2 Reference Manual*" , Prentice Hall, 1988.

[ISO 93] ISO/IEC 10728:1993 International Standard, "*Information Technology-Information Resource Dictionary System (IRDS) Service Interface* ".

[Katz 90] Katz, R., "*Towards a Unified Framework for Version Modelling in Engineering Databases* ", ACM Computing Surveys, 22(4), 1990, pp. 375-408.

[Kemper 88] Kemper, A., Wallrath, M., "*A Uniform Concept for Storing and Manipulating Engineering Objects* ", Advances in Object-Oriented Database Systems, 2nd International Workshop on Object-Oriented Database Systems, Springer-Verlag, Sept. 1988, pp. 292-316.

[Kesim 93] Kesim, F., "*Temporal Objects in Deductive Databases* ", Ph.D. Dissertation, Imperial College, University of London, February 1993.

[Khoshafian 90a] Khoshafian, S., *et al.*, "*Storage Management for Persistent Complex Objects* ", Information Systems, 15(3), 1990.

[Khoshafian 90b] Khoshafian, S., Abnous, R., "*Object Orientation: Concepts, Languages, Databases, User Interfaces* ", John Wiley & Son Inc., 1990.

[Khoshafian 93] Khoshafian, S., "*Object-Oriented Databases* ", John Wiley & Son Inc., 1993.

[Kim 88a] Kim, W., "*Integrating An Object-Oriented Programming System With a Database System* ", OOPSLA'88, 1988, pp. 142-152.

[Kim 88b] Kim, W., Chou, H., "*Versions of Schema for Object-Oriented Databases* ", Proceedings of the 14th VLDB Conference, Los Angeles, California, 1988, pp. 148-159.

[Kim 89] Kim, W., "*A Model of Queries for Object-Oriented Databases* ", Proceedings of the 15th VLDB Conference, August 1989.

[Kim 90a] Kim, K., "*Parallelism in Object-Oriented Query Processing*", Proceedings of the 6th International Conference on Data Engineering, 1990, pp. 209-217.

[Kim 90b] Kim, W., "*Introduction to Object-Oriented Databases*", The MIT Press, 1990.

[Kim 90c] Kim, W., *et al., "Architecture of the ORION Next-Generation Database System",* IEEE Transactions on Knowledge and Data Engineering", SIGPLAN Notices, 25(10), 1990, pp. 109-124.

[Kindberg 90] Kindberg, T., "*A Reconfigurable Distributed Operating System* ", Ph.D. Dissertation, University of Westminster, 1990.

[Klahold 86] Klahold , P., *et al.*, "*A General Model for Version Management in Databases* ", Proceedings of the 12th International Conference on Very Large Data Bases VLDB, Kyoto, August 1986, pp. 319-327.

[Knuth 68] Knuth, D., "*The Art of Computer Programming* ", Vol.1, Addison-Wesley, 1968.

[Lam 89] Lam, H., *et al.*, "*An Object Flow Computer for Database Applications* ", IWDM'89: 6th International Workshop on Database Machines, France, June 1989, pp. 1-7.

[Lavington 87] Lavington, S., *et al.*, "*Hardware memory Management for Large Knowledge Bases* ", PARLE "Parallel Architecture and Languages Europe", Vol.1, Springer-Verlag, Eindhoven, The Netherlands, June 1987, pp. 226-241.

[Lavington 88] Lavington, S., *et al.*, "*Technical Overview of the Intelligent File Store* ", Knowledge-Based Systems, Vol.1, No.3, Butterworths, June 1988.

[Lavington 91] Lavington, S., *et al.*, "*A Modularly Extensible Scheme for Exploiting Data Parallelism* ", 3rd International Conference on Transputer Applications, Glasgow, August 1991.

[Lavington 92] Lavington, S., *et al.*, "*Parallel Architecture For Smart Information Systems* ", UNICOM Seminars on Commercial Parallel Processing, London, February 1992.

[Lee 89] Lee, C., "*An Object Flow Computer for Object-Oriented Database Applications* ", Ph.D. Dissertation, Department of Electrical Engineering, University of Florida, 1989.

[Leilich 78] Leilich, H., *et al.*, "*A Search Processor for Database Management Systems* ", Proceedings of the 4th International Conference on Very Large Databases, West Berlin, 1978.

[Lerner 90] Lerner, B., Habermann, A., "*Beyond Schema Evolution to Database Reorganization* ", ECOOP/OOPSLA'90 Proceedings, 1990, pp. 67-76.

[Li 94] Li, Q., Dong, G., "*A Framework for Object Migration in Object-Oriented Databases* ", Data & Knowledge Engineering, 13 (1994), pp. 221-242.

[Lyngbaek 86] Lyngbaek, P., Kent, W., "*A Data Modelling Methodology for the Design and Implementation of Information Systems* ", Proceedings of the International Workshop on Object-oriented Database Systems, IEEE Computer Society Press, New York, 1986.

[MacDougall 87] MacDougall, M., "*Simulating Computer Systems: Techniques and Tools* ", MIT Press Series in Computer Systems, Herb Schwetman (Editor), 1987.

[Meyer 88] Meyer, B., "*Object-Oriented Software Construction* ", Prentice-Hall, 1988.

[Monk 92] Monk, S., Sommerville, I., "*A Model for Versioning of Classes in Object-Oriented Database* ", Proceedings of BNCOD 10, Aberdeen, Scotland, 1992, pp. 42-58.

[Monk 93] Monk, S., Sommerville, I., "*Schema Evolution in OODBs Using Class-versioning* ", SIGMOD RECORD, Vol.22, No.3, September 1993, pp. 16-22.

[Ng 88] Ng, Y., Barros, S., "*Active memory for Text Information Retrieval* ", ACM SIGIR'88: 11th International Conference on Research and Development in Information Retrieval, Grenoble, France, June 1988, pp. 613-627.

[Nguyen 89] Nguyen, G., Rieu, D., "*Schema Evolution in Object-Oriented Database Systems* ", Data and Knowledge Engineering, Vol.4, No.1, July 1989, pp. 43-67.

[Odijk 87] Odijk, E., "*The DOOM system and its applications: A survey of Esprit 415 subproject* ", Philips Research Laboratories, PARLE: Parallel Architecture and Languages Europe, Vol.1, Springer-Verlag, Eindhoven, The Netherlands, June 1987, pp. 461-479.

[ODMD-93] "*The Object Database Standard: ODMG-93* ", Release 1.1, Edited by Cattel R., Morgan Kaufmann Publishers, California, 1993.

[Ogura 89] Ogura, T., *et al.*, "*A 20-kbit Associative Memory LSI for artificial Intelligent Machines* ", Journal of Solid-State Circuits, Vol.24, No.4, August 1989, pp. 1014-1020.

[Parhami 73] Parhami, B., "*Associative memories and Processors: An Overview and Selected Bibliography* ", Proceedings of the IEEE, Vol.61, No.6, June 1973, pp. 722-730.

[Parhami 90] Parhami, B., "*Associative Memory Designs for VLSI Implementation* ", PARBASE-90: International Conference on Databases, Parallel Architectures, and their Applications, IEEE Computer Society Press, 1990, pp. 359-366.

[Pears 92] Pears, R., Gray, W., "*Response Time Minimisation in a Parallel Database System* ", Proceedings of BNCOD 9, Wolverhampton, UK., 1991, pp.148-167.

[Penney 87] Penney, D., Stein, J., "*Class Modification in the GemStone Object-oriented DBMS* ", Proceedings of OOPSLA'87, 1987, pp. 111-117.

[Pountain 88] Pountain, D., "*REKURSIV : An Object-Oriented CPU* ", BYTE, November 1988, pp. 341-349.

[Riegel 88] Riegel, S., *et al.*, "*Integration of Database Management with an Object-Oriented Programming Language* ", Advances in

Object-Oriented Database Systems, 2nd International Workshop on Object-Oriented Database Systems, Springer-Verlag, September 1988, pp. 317-322.

[Schildt 91] Schildt, H., "*C++- The Complete References* ", McGraw-Hill, 1991.

[Scholl 90] Scholl, M., *et al.*, "*Views in Object-Oriented Databases* ", Proceedings of Second Workshop on Foundations of Models and Languages for Data and Objects, Aigen, Austria, September 1990.

[Shapiro 89] Shapiro, M., *et al.*, "*Persistence and Migration for C++ Objects* ", Proceedings of ECOOP'89 The Third Conference on Object-Oriented Programming, Cambridge, July 1989, pp. 191-206.

[Shin 92] Shin, Y., *et al.*, "*A Special-Purpose Content Addressable Memory Chip for Real-Time Image Processing* ", IEEE Journal of Solid-State Circuits, Vol.27, No.5, May 1992, pp. 737-744.

[Sjøberg92] Sjøberg, D., "*Measuring Schema Evolution* ", Technical Report FIDE/92/36, Department of Computing Science, University of Glasgow, Glasgow, G12 8QQ, 1992.

[Skarra 86] Skarra, A., Zdonik, S., "*The Management of Changing Types in an Object-Oriented Database* ", OOPSLA'86 proceedings, 1986, pp. 483-495.

[Skarra 87] Skarra, A., Zdonik, S., "*Type Evolution in an Object-Oriented Database* ", Research Directions in Object-Oriented Programming, MIT Press Series in Computer Systems, MIT Press, Cambridge, MA, 1987, pp. 393-415.

[Su 89] Su, S., *et al.*, "*An Object-Oriented semantic association Model (OSAM*)* " Chapter 17 in AI:Manufacturing Theory and Practice, Industrial Engineering and Management Press, Norcross, GA, 1989, pp. 463-494.

[Tamura 90] Tamura, L., *et al.*, "*A 4-ns BiCMOS Translation-Lookaside Buffer* ", IEEE Journal of Solid-State Circuits, Vol.25, No.5, October 1990, pp. 1093-1101.

[Thakore 94] Thakore, A., Su, S., "*Performance Analysis of Parallel Object-Oriented Query Processing Algorithms* ", Distributed and Parallel Databases 2 (1994), pp. 59-100.

[Ungar 87] Ungar, D., "*The Design and Evaluation of a High Performance Smalltalk System* ", An ACM Distinguished Dissertation 1986, MIT Press, 1987.

[Urban 87] Urban, S., "*Constraint Analysis for the Design of Semantic Database Update Operations* ", University of Southern Louisiana, Ph.D. Dissertation, 1987.

[Valduriez 84] Valduriez, P., Gardarin, G., "*Join and Semijoin Algorithms for a Multiprocessor Database Machine* ", ACM Transaction on Data Base Systems, Vol.9, No.1, March 1984, pp. 133-161.

[Valduriez 86] Valduriez, P., *et al.*, "*Implementation Techniques of Complex Objects* ", Proceedings of the twelfth International Conference on Very Large Data Bases VLDB, Kyoto, August 1986, pp. 101-110.

[Valduriez 87] Valduriez, P., "*Join Indices* ", ACM Transaction on Data Base Systems, Vol.12, No.2, June 1987, pp. 218-246.

[Valduriez 92] Valduriez, P. (Editor),"*Parallel Processing and Data Management* ", Chapman & Hall, 1992.

[Vossen 90] Vossen, G., "Data Models, Database Languages and Database Management Systems", Addison-Wesley, 1990, chapter 16, pp. 307-363.

[Willshire 90] Willshire, M., Kim, H., "*Properties of Physical Storage Models for Object-Oriented Databases* ", International Conference on Databases, Parallel Architectures and their Applications PARBASE-90, IEEE Computer Society Press, 1990, pp. 94-99.

[Zdonic 88] Zdonic, S., "*Data Abstraction and Query Optimization* ", Advances in Object-Oriented Database Systems, 2nd International Workshop on Object-Oriented Database Systems, Springer-Verlag, September 1988, pp. 368-373.