VAGABOND

The Design and Analysis of a

Temporal Object Database Management System

Kjetil Nørvåg

Department of Computer and Information Science
Norwegian University of Science and Technology

2000

Abstract

Storage costs are rapidly decreasing, making it feasible to store larger amounts of data in databases. However, the increase in disk performance is much lower than the increase in memory and CPU performance, and we have an increasing secondary storage access bottleneck. Even though this is not a new situation, the advent of very large main memory has made new storage approaches possible.

In most current database systems, data is updated in-place. To support recovery and increase performance, write-ahead logging is used. This logging defers the in-place updates. However, sooner or later, the updates have to be applied to the database. This often results in non-sequential writing of lots of pages, creating a write bottleneck. To avoid this, another approach is to eliminate the database completely, and use a *log-only* approach, similar to the approach used in *log structured file systems*. The log is written contiguously to the disk, in a no-overwrite way, in large blocks.

This thesis presents the architecture and design of Vagabond, a temporal object database management system (ODBMS) based on the log-only principle. Solutions to problems regarding temporal data management, fast recovery, efficient management of large objects, dynamic reclustering, and dynamic tuning of system parameters are provided. This includes a new index structure for indexing temporal objects, persistent caching of index entries to solve the object indexing bottleneck, algorithms for transaction management, and declustering strategies to be used in a parallel temporal ODBMS.

In order to compare the log-only approach with the traditional in-place update approach, analytical cost models are used to study the performance of the approaches. The analysis shows that with the workloads we expect to be typical for future ODBMSs, the log-only approach is highly competitive with the traditional in-place update approach.

Many of the ideas presented in this thesis are also useful outside the log-only context. In papers included as appendixes, we show how the ideas can be applied to temporal ODBMSs based on traditional in-place updating techniques.

Preface

This is a doctoral thesis submitted to the Norwegian University of Science and Technology for the doctoral degree “doktor ingeniør”. This work has been carried out at the Database Systems Group, Department of Computer and Information Science, at the Norwegian University of Science and Technology, under the supervision of Prof. Kjell Bratbergsengen. The doctoral study was funded by the Norwegian Research Council (NFR).

Acknowledgments

First of all I want to thank my advisor Kjell Bratbergsengen for guidance and many good ideas throughout the work towards my doctoral degree.

During the years I have worked on this thesis many people have helped me reach my goal. In particular, I would like to thank Olav Sandstå for insightful discussions, lots of valuable feedback, and for taking the time to proofread this thesis as well as my papers. I also want to thank the other members of the Database Systems group for providing a good environment for doctoral students. During the last two years I have been employed as a lecturer in the Group for Computer Architecture and Design, and I want to thank Lasse Natvig and Pauline Haddow for providing me the opportunity to finish this thesis. I am also grateful to the department’s always friendly and helpful administrative staff.

I would also like to thank Prof. Malcolm Atkinson for many valuable comments on the thesis.

Finally, I thank my family, who have always encouraged me. This support has been of great value.

Contents

List of Tables

List of Figures

Part I

Overview

Chapter 1

Introduction

The recent years have brought computers into almost every office, and this availability of powerful computers, connected in global networks, has made it possible to utilize powerful data management systems in new application areas. The increasing performance and storage capacity, combined with a decreasing price, have made it possible to realize applications that were previously too heavy for medium- and low-cost computers. However, high performance and storage capacity is not enough. We need support software, including database management systems, operating systems, and compilers, that are able to benefit from the advances in hardware. This often means rethinking previous solutions, similar to what was done in the hardware world with the introduction of the RISC concept.

In this thesis, we concentrate on database management systems (DBMS), quite likely to be the bottleneck in many future systems. The first step in the process of rethinking old solutions has already been done, with the advent of object database management system (ODBMS).[1] While relational database management systems (RDBMSs) have good performance for many of the traditional application areas, new applications demand more than traditional RDBMSs can deliver. The increased modeling power and removal of the language impedance mismatch in ODBMSs, have made integration between application programs easier, and in many cases helped to increase the performance of the applications.

Traditionally, data (objects/tuples) have lived in an artificial, modeled world, after being inserted into the database. This creates a mismatch in many ways similar to the language impedance mismatch in RDBMSs. What we would like, is DBMSs supporting a world more similar to our own, which includes *time and space*. This is not at all a new observation, especially the aspect of temporal database management has been an active research area for many years. However, current database architectures, which are adequate for yesterday's applications, might have problems coping with tomorrow's application. In this thesis, we will reconsider some of what is "established truth", and propose a new architecture, the *Vagabond Temporal Object Database Management System*, which should be more suitable for tomorrows applications.

Before we finish this section, it is very important to emphasize that some of the ideas in this thesis are not new. However, many of the ideas did not have a supporting framework when they were proposed. Hence, many of the ideas are now forgotten. One notable exception, is some of the ideas from the POSTGRES system. POSTGRES included many novel ideas, which, because they where incorporated into a system, managed to survive. Unfortunately, POSTGRES was in many ways too early, and even though many of the elements of POSTGRES survived into current object-relational systems, some of the ideas we will concentrate on in this thesis, like the no-overwrite strategy, and

[1] The term *object-oriented database management system* (OODBMS) was previously used, but now the more precise term *object database management system (ODBMS)* has gained acceptance.

keeping previous versions, have later had little attention in DBMS research.

In the rest of this chapter, we will motivate the work that will be presented in the rest of this thesis. In Section 1.1 we describe some application areas that have only limited support in existing database system. Based on this discussion, we summarize some of the problems and shortcomings of current systems in Section 1.2, and outline assumptions and features that motivated the design of the Vagabond system, which will be described in detail throughout the rest of the thesis. In Section 1.3, we outline the structure of the rest of the thesis.

1.1 Application Areas

We can categorize application areas into *existing* application areas, and *emerging* application areas. Existing application areas include the traditional database areas, for example transaction processing applications, well suited for RDBMSs. They also include application areas where application specific DBMSs or file systems have been used earlier, because existing general purpose DBMSs can not handle the performance constraints. Emerging application areas includes both new application areas, that are emerging as a response to the increased computer performance in general, and application areas that are a response to other technologies, for example the World Wide Web.

We will in this section first describe some examples of existing applications where DBMSs until recently have been a potential performance bottleneck:

- Geographical information systems.
- Scientific and statistical databases.
- Multimedia systems.
- PACS (picture archiving and communications systems).

Next, we will describe some applications where increased database support will be needed in order to deliver the desired performance:

- Temporal DBMSs.
- Semistructured data management/XML.

Geographical Information Systems. A geographical information system (GIS) is a system for management of geographical data, i.e., *data which describes phenomena directly or indirectly associated with a location (and possibly time and orientation as well) relative to the surface of the Earth* [34].

Earlier, GIS employed the DBMS (usually a RDBMS) to manage the fact data[2] only, but used proprietary file management systems to take care of geometrical and topological data. The main reason for this, was that most RDBMSs did not support sequences (“ordered sets”), and retrieving polygons from relations was (and still is) prohibitively expensive. This is unfortunate, because the file management systems tend to be single-user, and there are no transactional access control as in DBMSs. Recently, GIS have been built by extending database systems with spatial data types. However, these ad-hoc solutions do not really address the main problem, the data model: concepts are simple, the data

[2]Fact data is data describing the objects, e.g., the name of a road, but not the “road object” itself.

type system is weak, data have to be normalized in first normal forms while hierarchical structures are needed, semantic links are lost and need to be rebuilt through semantic constraints, and the data access system is very expensive because of joins [51].

With its increased modeling power, ODBMSs are ideal for GIS applications. They support complex objects and relationships efficiently. However, some ODBMSs do not have sufficient support for large objects, and not all ODBMSs have a scalable architecture.

Scientific and Statistical Databases. Scientific and statistical databases (SSDBs), for example survey data and data from physical experiments, have many characteristics in common, which makes it practical to consider them together:

- The size of the databases are usually *very large.*
- The update frequency is often *very low*. The reason for this, is that the primary purpose of an SSDB is to collect data for future reference and analysis.
- Bulk loading is frequently used to insert data into the database.
- The read/write ratio can be low. Because of the size of the database, summary data is often used instead of the whole database in queries.
- Data in both scientific and in statistical databases are eventually statistically analyzed.
- Complex relationships exist between data. For example, experiments not only carry result data, but also configuration and environmental data.
- Multidimensional data is frequent.
- Data is often sparse, i.e., many attributes have a NULL value.

The size of SSDBs pose a problem for many DBMSs, and the complex relationships make a data model with high modeling power desirable. Traditional systems do not support multidimensional data well, and in the case of sparse data, efficient support for compression is necessary. This does not only include support for compression and decompression itself, but also efficient access and manipulation of compressed data.

In analysis, statistical operators are needed. These are not included in traditional systems. Another important feature in practical SSDBs, is bulk loading, which few systems handle well.

Database research and development is highly market driven, and until very recently, the active research in this area was very limited. This has changed dramatically the last few years, with the increased interest in data warehousing/OLAP, which has many similarities with SSDBs.

Multimedia Systems. Multimedia data management differs from traditional data management in several ways:

- Large objects, for example images and videos, are common. In general, there are sufficiently many large objects stored in such a database to make the total database size large as well.
- New and complex data types.
- New types of queries. One example is query by contents, another is queries on image characteristics (for example on image histograms).

- Isochronous retrieval: In the case of dynamic data, like video, data is to be delivered in pieces of the object at regular time intervals, and not the whole object at once. The scheduling of this data delivery is complicated, as is witnessed from dedicated video servers, which do not have to care about the other database aspects.

Even though some vendors define all databases capable of storing large objects as multimedia DBMS, the fact is that current DBMSs have only limited support for multimedia data, and this is particularly true for isochronous delivery of the data stored in the database.

Picture Archiving and Communications Systems. Picture archiving and communications systems (PACS) will be an important part of tomorrow's health care. In the future different kinds of data will be stored in such systems, but currently most systems concentrate on storage of pictures, for example X-ray pictures. The pictures stored in these systems are required to have a high resolution, and with the number of pictures to be stored in such a system the database size will be very large. The historical data in a PACS system will be very infrequently accessed, and can be stored in tertiary storage.

Temporal Database Management Systems. A temporal DBMS is a DBMS that supports some aspects of time. Informally, this means that an object (or a tuple) is associated with time, and that the object can exist in several versions, each version being valid in a certain time interval. An example is the salary of a person. If the salary is represented as a temporal object, a new object version is created every time the salary is changed. In a temporal DBMS, this versioning, related to time, is supported and maintained by the system, which also provides support for querying the temporal data.

The temporal aspect exists in most real life databases, where some or all of the data is associated with some aspect of time. Examples include:

- Accounting: What bills were sent out and when, and what payments were received and when.
- GIS: The geography, such as rivers, and the existence, shape and size of objects such as houses and roads, change over time.
- Stock marked data.
- Patient records.
- Personnel information, including salary histories.
- Airline reservation systems.
- In scientific DBMSs, timestamping of data is important, for example for data from an experiment that is repeated several times.

We will give a more detailed introduction to temporal DBMSs in Chapter 4.

Semistructured Data Management/XML. A very active research area at the moment, is *semistructured data management.* Semistructured data is data where the information that is normally associated with a schema, is contained *within the data*. In some forms of semistructured data there are no separate schemas, in others it exists, but only places loose constraints of the data [35].

The main reasons for the heavy interest in semistructured data are its application in data exchange/data integration, and the large amount of semistructured data available on the Web. Research

in semistructured data management has recently been concentrated on the theory of semistructured data; models, query languages, but less on physical management. Several approaches have been taken in incorporating the ideas on top of traditional ODBMSs, for example on O_2 [2], but only a few systems have been specially designed for semistructured data. One example of such a system is Lore [138].

ODBMSs are very appropriate for semistructured data management, as their underlying model has many similarities with most semistructured data models, for example the Object Exchange Model (OEM) [3]. However, some features not provided by most existing ODBMSs are desired. One example is query languages suitable for semistructured data, and appropriate optimization techniques. If these DBMSs should deliver reasonable performance, new indexing techniques are also needed, including full text indexes.

As pointed out by Abiteboul [1], we are often more concerned by querying the recent changes in some data source than in examining the entire source. Support for temporal data in the storage layer would facilitate this, and this can also be useful in distributed DBMSs where data is exchanged in bulk at regular intervals.

1.2 The Need for a New Architecture

Based on the discussion in the previous section, we have identified some features that should be supported by future database systems:

- Temporal data and operations on these.
- Large objects and flexible partitioning of large objects.
- Isochronous delivery of data.
- Queries on large data sets.
- In applications with low read/write ratio, it should be possible to use this characteristic to increase performance.
- Full text indexing.
- Multidimensional data.
- Efficient storage of sparse data (for example by the use of data compression).
- Dynamic clustering and dynamic tuning of system parameters.

Until now, no single existing system has supported all these features. Ad-hoc solutions exist for some of the features, but these are often not scalable, or will not work well together with support for the other features. We believe that future systems should efficiently support these features, in *one integrated system*. In this thesis, we show how this can be done, through the design of the temporal ODBMS Vagabond.[3] Vagabond is designed to support the listed features, with a philosophy based on the following assumptions:

[3]From *Webster's Encyclopedic Unabridged Dictionary*: Vagabond: "a person, usually without a permanent home, who wanders from place to place; nomad". Quite similar to our objects!

1. Although many of the current problems might be handled by future main-memory database management systems (MMDBMSs), there are many problems (and more will appear, as the computers become powerful enough to solve them) that require the management of larger amounts of data than can be handled by a MMDBMS alone. However, the increasing amounts of main memory should be utilized as far as possible in order to reduce time consuming secondary storage accesses.

2. *The main bottleneck* in a DBMS for large databases is still secondary storage access. In a DBMS, most accesses to data are read operations. Consequently, database systems have been *read-optimized*. However, as main-memory capacity increases, we expect that the amount of disk-write operations relative to disk-read operations will increase (most read operations can be satisfied from the main-memory buffer). This calls for a focus on *write-optimized* DBMSs.

3. To provide the necessary computing power *and* data bandwidth, a parallel architecture is necessary. A shared-everything approach is not scalable, so our primary interest is in ODBMSs based on shared-nothing multicomputers. With the advent of high performance computers, and high speed networks, we expect multicomputers based on commodity workstations/servers and networks to be most cost effective.

4. In most application areas, there is a need for increased data bandwidth, and not only increased transaction throughput (although these points are related). This is especially important for emerging application areas such as multimedia and supercomputing applications, which have earlier used file systems.

5. Even though set-based queries have been a neglected feature in most ODBMSs, we expect it to be as important in the future for ODBMSs as it has been previously for RDBMSs. The popularity of the hybrid object-relational systems justifies this assumption.

6. Distributed information systems are becoming increasingly common, and they should be supported in a way that facilitates both efficient support for distribution, *and* efficient execution of local queries and operations.

1.3 Outline of the Thesis

The thesis is logically divided into four parts. The first part, Chapter 2 to 5, is mainly an introduction to ODBMSs and ODBMS implementation issues, temporal DBMS, and the log-only approach.

- *Chapter 2* describes the most important features of ODBMSs, gives an overview of the ODMG standard, and outlines the history of ODBMSs.
- *Chapter 3* discusses design issues in ODBMSs.
- *Chapter 4* gives an introduction to temporal DBMSs in general.
- *Chapter 5* gives an introduction to log-only DBMS, and a short overview of previous systems based on the log-only approach.

In the second part, the architecture of the Vagabond log-only DBMS and the most important algorithms are described in detail.

- *Chapter 6* describes the architecture of the Vagabond temporal ODBMS.
- *Chapter 7* discusses two techniques for reducing the data transfer volume: signatures and object compression.
- *Chapter 8* studies the problems of indexing object identifiers (OIDs) in a temporal ODBMS, and proposes a new indexing structure suitable for this task.
- *Chapter 9* introduces a novel structure called the *Persistent Cache*, which reduces the OID indexing cost.
- *Chapter 10* gives a more detailed description of large objects and their use in Vagabond.
- *Chapter 11* discusses object declustering in parallel and distributed temporal ODBMSs.
- *Chapter 12* describes the most important operations in Vagabond.
- *Chapter 13* describes the most important physical data structures in Vagabond.

In the third part, log-only database systems are compared analytically with traditional in-place updating ODBMSs, and we conclude the thesis.

- *Chapter 14* contains analytical models of a log-only ODBMS and an in-place update ODBMS, and uses these models to compare the hypothetical performance of the two approaches.
- *Chapter 15* contains a qualitative analysis of the declustering strategies discussed in Chapter 11.
- *Chapter 16* concludes the thesis and outlines directions for further research.

The fourth part, Appendix A to F, is a compilation of papers that discuss issues not covered in detail by the main part of the thesis. The four last papers show how the results of the main part of the thesis are also applicable for temporal ODBMSs based on traditional techniques.

- The paper “Aggregate and Grouping Functions in Object-Oriented Databases”, presented at SCCC’96, is included in Appendix A.
- The paper “Improved and Optimized Partitioning Techniques in Database Query Processing”, presented at BNCOD’97, is included in Appendix B.
- The paper “An Analytical Study of Object Identifier Indexing”, presented at DEXA’98, is included in Appendix C.
- The paper “Optimizing OID Indexing Cost in Temporal Object-Oriented Database Systems”, presented at FODO’98, is included in Appendix D.
- The paper “The Persistent Cache: Improving OID Indexing in Temporal Object-Oriented Database Systems”, presented at VLDB’99, is included in Appendix E.
- The paper “Efficient Use of Signatures in Object-Oriented Database Systems”, presented at ADBIS’99, is included in Appendix F.

In Appendix G we present a validation of the index buffer model used in the papers in Appendix C, D, E, and F. In addition, a list of abbreviations used in this thesis is provided in Appendix H.

Chapter 2

Object Database Management Systems

Relational database management systems (RDBMS) have revolutionized database management during the last 20 years. Important reasons for the success are SQL, and the ability to efficiently perform queries over large amounts of data. However, RDBMSs are based on a simple data model. Even though this gives high performance for many typical data-retrieval applications, the result can often be very low performance in applications managing data with complex relationships. For example, until very recently it was impossible to design a GIS systems based on traditional RDBMS technology.[1] In addition, in many applications with high transactions rates, systems based on hierarchical and network data models have continued to be used.

For some application areas, a more complex data model and focus on data manipulation, rather than data retrieval, is desired. Typical examples of such systems have been GIS, CAD, software development systems and more recently also Web databases. Many applications also need to do complex operations on the data. In a typical RDBMS, this has to be done by accessing the database from the application program by using database commands embedded in some general programming language. This *language impedance mismatch* is costly and inefficient.

Object database management systems (ODBMS), previously called object-oriented database management systems, emerged as an answer to the shortcomings of previous models and systems. The rest of this chapter will give an introduction to ODBMS, and we will start by defining the term *object database management system (ODBMS)* in the next section. An overview of the world of ODBMS would not be complete without an overview of the contents of the ODMG standard, which is given in Section 2.2. To set our work into perspective, we briefly outline the history of ODBMSs in Section 2.3, from the first approaches in persistent programming languages, via storage managers, to today's commercially available ODBMSs. We also give a brief overview of object-relational database management systems (ORDBMS).

2.1 What is an Object Database System?

As is obvious from the ODBMS research prototypes and commercially available ODBMSs, the design space for an ODBMS is much larger than for RDBMSs. However, there are some features and characteristics shared by most of them, initially described in *The Object-Oriented Database System Manifesto* by Atkinson et al. [6]. We will now summarize the most important features, separated into language related features (the OO part), and the database features (the DB part).

[1] A notable exception is Techra [204], which over a decade ago included support for GIS data management, including sequences.

Language features:

- Complex objects. The ability to build complex objects from simpler ones by applying constructors to them. Complex object constructors include tuples, sets, bags, lists, and arrays.
- Object identity. All objects have a system managed identity that is independent of the value of the object. The identity is assigned by the system, can not be altered by the user, and remains the same even when the value of the object changes.
- Encapsulation. Encapsulation is used to distinguish between the specification and the implementation of an operation. No operations, outside those specified in the interface, can be performed. This restriction holds for both update and retrieval operations.
- Types or classes. Types or classes should be supported. The ODMG standard encapsulates both, and the language binding used decides to what extent these concepts are supported.
- Class or type hierarchies. Inheritance is a powerful modeling tool, because it gives a concise and precise description of the world, and it helps in factoring out shared specifications and implementations in applications.
- Overriding, overloading and late binding. This is the concept of having several implementations of an operation, for each of the types. Which implementation to use, is decided at run-time, *late binding*.
- Computational completeness. To avoid the language impedance mismatch, the data manipulation language should be computationally complete.
- Persistence. Persistence is the ability of the programmer to have her/his data survive the execution of a process, in order to eventually reuse the data in another process. Persistence should be orthogonal, i.e., each object, independent of its type, is allowed to become persistent as such (i.e., without explicit translation). It should also be implicit: the user should not have to explicitly move or copy data to make it persistent.

Database features:

- Secondary storage management. Database mechanisms as index management, data clustering, data buffering, access path selection and query optimization should be invisible to the user: they are simply performance features. There should be a clear independence between the logical and the physical level of the system.
- Concurrency and recovery. The system should offer the same level of service as traditional database systems, i.e., atomicity and controlled sharing when multiple users access and update data. The same applies to recovery, in case of hardware or software failures, the system should recover, i.e., bring itself back to a consistent state.
- Ad-hoc query facility. The system should provide functionality of an ad-hoc query language, though not necessarily as an own query language. This is probably the feature where current ODBMSs differ most. While some systems, like O_2, offer a SQL like language (OQL), with query optimization similar to RDBMSs, other systems only provide primitive scan operations.

The *Manifesto* also lists some additional features, the most important being support for *distribution*, *design transactions* ("long" transactions) and versioning. These features are not mandatory to make a database system an object-oriented system, but features that are desired in many of the typical ODBMS applications. Thus, these features are supported to some extent in most commercial ODBMSs.

2.2 The ODMG Standard

ODBMS is now a relative mature technology, and commercial ODBMSs have proved to be competitive with RDBMSs in many application areas, and superior in others. They have been able to deliver high performance and provide high availability. Still, they have not managed to seriously threaten the traditional RDBMSs.

There are several reason why the ODBMS market segment is still small, but one important factor has been lack of standardization. One important reason for the success of the RDBMSs, is the common data model and the common data specification and manipulation language. This was realized by the ODBMS vendors in the early 90's, and the *Object Database Management Group (ODMG)* was formed in 1991 to develop and promote standards for object storage. The participants of the ODMG includes representatives from all major ODBMS vendors. Recently, the focus of the ODMG have been broadened, and the name changed to the *Object Data Management Group*.

2.2.1 The Components of the ODMG Standard

The components of the ODMG standard [43] are built upon the ODMG object model, which is a superset of the OMG object model. The specification covers three areas:

- Object definition language (ODL).
- Object query language (OQL).
- Language bindings.

Object Definition Language. ODL is in fact a syntax of the object model, and is a superset of OMG's IDL. It can be used to define a database schema in a programming language independent manner in terms of object type, attributes, relationships and operations. The resulting schema can be moved from one database to another. The schema of an application can be translated to declarations in different programming languages. These schemas can be included in the application code.

Object Query Language. OQL is a declarative query language, and is a superset of the part of SQL that deals with database queries. It includes support for object sets and structures, and has object extensions to support object identity, complex objects, path expressions, operation invocation, and inheritance.

Language Bindings. ODBMSs are accessed through languages with support for persistent objects, usually extensions of existing general purpose programming languages. The ODMG language bindings define extensions to the languages to support and integrate OQL, navigation and transactions. Currently, language bindings for C++, Java and Smalltalk have been standardized.

2.2.2 The ODMG Standard in Practice

The ODBMS vendors have been slow at adopting the ODMG standard, and unfortunately, they currently seem even less eager to do so. Vendors can claim compliance with one or more components of the ODMG standard, i.e. one or several of the C++/Java/Smalltalk language bindings, and OQL. Even though many vendors claim that their systems are ODMG compliant, the lack of certification procedures is a problem. The only vendor close to supporting the whole standard was O_2, which is no surprise, as the standard itself, especially OQL, borrowed heavily from O_2. However, the O_2 is no longer on the market.

Currently, there seems to be little interest in continued work on the ODMG object model, OQL, and the language bindings. The model is only used in the ODMG specification itself, the ODBMS vendors prefer to use their proprietary C++ language bindings, and OQL has only limited support. Most of the interest at the moment is in the Java binding and object/relational mappings, and it is very likely that future work will be in these directions.

2.3 Object Database Systems

To set our work into perspective, we briefly summarize previous work on ODBMSs. We have summarized all implemented systems we are aware of in Table 2.1. This summary is provided for two reasons. First of all, we want to show that ODBMSs have been an active research area, and still is. Second, we provide the summary with references, to make it easier for others to probe earlier works, as we are not aware of any other published summary or survey trying to cover the implemented systems. In this summary, we classify the systems in three groups:

- Early approaches.
- ODBMSs and storage managers with language binding.
- ORDBMSs.

For the commercial systems, the publications cited do not necessarily represent descriptions of the current versions of the commercial systems. For information on current versions of the systems and their features, the reader is encouraged to visit the Web sites of the respective ODBMS companies.

2.3.1 Early Approaches to Persistent Programming Languages

Traditionally, users and applications have communicated with the database system through special data definition languages (DDL) and data manipulation languages (DML). Operations on tuples have been done with some predefined functions. If more advanced operations were desired, a general purpose language with embedded database language functions were used. With this approach, you get a language impedance mismatch.

To avoid the language impedance mismatch, new systems was developed. In these systems, there were no distinction between database (persistent) and no-database (transient) data, the same language is used for both. The first such systems were ASTRAL [32, 33] and PASCAL/R [181]. Later, other systems followed, for example PS-algol [49]. In the next phase of the evolution, persistent versions of Smalltalk and persistent C++ became popular, used in combination with the storage managers summarized in the next section.

Name	References	Name	References
AGNA	[145]	O_2	[7, 53, 165]
Amadeus	[80, 203]	Objectivity	[167]
BeSS	[17]	ObjectStore	[118, 171]
Bubba	[28]	OBST	[42]
Cricket	[186]	ODB-II	[168]
Dalí[2]	[99]	ODE	[71]
DASDBS	[180]	ONTOS	
Eiffel**	[137]	ORION	[112]
Encore	[89]	OSAM*.KBMS/P	[201]
EOS	[81]	PJama	[175]
EXODUS	[38]	Poet	
EyeDB		PPOST	[22]
GemStone	[37, 133]	PRIMA	[74, 75]
ITASCA	[97]	Ptool	[79]
Iris	[68, 215]	QuickStore	[214]
Jasmine	[95]	Shore	[39]
KIOSK	[146]	Texas	[189]
Lumberjack	[92]	Thor	[128, 130]
MATISSE	[134]	Tycoon	[135]
Mneme	[140, 142]	Versant	
Monet	[26, 27]	VODAK	[115]

Table 2.1: ODBMSs and storage managers with language binding.

2.3.2 ODBMSs and Storage Managers with Language Binding

After the first attempts with database programming languages, systems more closely resembling what we today call object database management systems entered the scene. In many of these systems, the focus was more on persistent programming than database management, and as a result, many of these system have only a very primitive query language, if any at all. However: all systems share one important goal, removing the language impedance mismatch.

The number of ODBMSs and storage managers with language binding is quite large, and the implemented systems we are aware of are summarized in Table 2.1. Most of the systems are only research prototypes, but some of them are commercialized ODBMSs: GemStone, Itasca, Jasmine, MATISSE, O_2, Objectivity, ObjectStore, ODB II, ONTOS, Poet, and Versant.

Several systems are marketed as ODBMSs, but are not included in Table 2.1. The reason for not including theses, is that they either lack some of the more important features expected from ODBMSs (they would more correctly be classified as object file managers or indexing tools), or that we have only limited information about the systems. The systems omitted from the summary include ActiveInfo, GOODS/POST++, Jeevan, Neoaccess (NeoLogic), ObjectFile (ObjectFile Ltd.), OOFILE (A.D. Software), Persist (Persist AG), PLOB! (Persistent Lisp OBjects, from University of Hamburg), Tenecit (Totally Objects), and TERSOL (TechKnowledge).

In addition, several systems use one of the systems in Table 2.1 as the storage manager in the system. This includes AllegroStore, which combines ObjectStore with CLOS (Common Lisp Object System), Multicomputer Texas [18] (described in Section 3.11.3), which uses Texas as the storage

Name	References
DB2	[41]
Illustra/Informix Universal Server	
Oracle 8	
POSTGRES	[195, 200, 197, 198]
Starburst	[85, 131]
UniSQL	

Table 2.2: Object Relational Database Systems.

manager in a parallel ODBMS, and Open ODBMS [20] and METU ODBMS [58], which both were developed on top of the Exodus storage manager.

2.3.3 Object-Relational Database Systems

ORDBMSs[3] carry on the relational paradigm. Data is still organized in relations, but the systems offer additional features, including support for more complex data types, and large objects. The most well-known of these are summarized in Table 2.2.

The history of ORDBMSs started with POSTGRES,[4] later commercialized into Illustra/Informix Universal Server. Currently, most major RDBMS vendors have extended their products to support object-relational features. Some ODBMSs, included in Table 2.1, have also been marketed as object-relational, or have features that make it possible to classify them as object-relational, for example MATISSE and ODB-II.

It is possible to implement an ODBMS on top of an ORDBMS backend, and vice versa. Examples are Paradise [55, 173], which uses Shore [39] as its underlying persistent object manager, and several commercial products that offer Java and C++ language bindings on top of ORDBMSs. Based on this observation, one might think that the division of ODBMS and ORDBMS is artificial, and that the ODBMS vs. ORDBMS discussion is more a debate on what interface to make available for users and programmers. It is important to note that this is not the case. While such approaches deliver the functionality, they are in general not efficient and scalable approaches.

2.4 Summary

We have in this chapter described the most important features of ODBMSs, given an overview of the ODMG standard, and provided an overview of previous and existing ODBMSs. Although we have tried to make the overview as complete as possible, we are fully aware that the list is not complete: many projects have been completed without any publication efforts, and new systems are developed and marketed as this thesis is written.

[2] Dalí has now been commercialized, and renamed *Datablitz*.

[3] Object relational database systems were previously called *extended relational database systems*.

[4] A "cleaned up version" of POSTGRES, *PostgreSQL*, is continuously under development by "the public domain community".

Chapter 3

Design Issues

The design of an ODBMS introduces new issues not found in RDBMSs. Each design issue may have alternative solutions, and few have definite answers. Many of them are also highly related, one of the alternatives for one issue can rule out alternatives for other issues. In this chapter, we discuss the most important issues, and provide a background for the description of Vagabond. This chapter also establishes the terminology which will be used in the rest of this thesis.

3.1 Object Identifiers

An object in an ODBMS is uniquely identified by an object identifier (OID). This OID is used as the "key" when retrieving the object from disk. OIDs can be *physical* or *logical*. If physical OIDs are used, the disk blocks where an object resides is given directly by the OID. If logical OIDs are used, it is necessary to use an OIDX index (OIDX) to map from a logical OID to a physical location. Most of the early ODBMSs and storage managers used physical OIDs because of its performance benefits, and many of the commercial ODBMSs still do. However, using physical OIDs have major drawbacks: relocation and migration of objects are more difficult, which in turn makes schema changes and reclustering more difficult. In a system that manages data which is expected to be stored for a long time (which is the case for most databases!), with possible changing applications and access patterns, logical OIDs should be used to avoid performance degradation later.

3.1.1 Physical OID

The OID is usually organized as a data structure, designed to help the ODBMS achieve good performance. For example, consider the 64-bit OID used by the Objectivity/DB (illustrated in Figure 3.1):

1. A logical (federated) database can be composed of several physical databases, and the first field in the OID identifies the physical database. A physical database is mapped to a file on a server, so this field identifies the server and file where the object is stored.

2. A physical database is composed of a number of containers. The container field identifies the actual container.

3. The page field identifies the page where the object is stored.

4. The slot field identifies the slot of an object on a page.

Database	Container[1]	Page	Slot
16 bits	16 bits	16 bits	16 bits

Figure 3.1: OID in Objectivity/DB [167].

An OID organized as a data structure like the one described above helps in providing efficient access to objects, but at the same time it also imposes a strict limit on the number of databases and containers in the system. Although 2^i objects can be created during the lifetime of a database if an OID size of i bits is used, the number is much smaller in practice. In real world applications, it is impossible to exploit all these fields, and it is obvious that careful design is needed to avoid problems with the maximum numbers of containers in the system. At the same time, it is also possible to get problems because of the limited number of objects that is possible to store in one container. These problems can be eliminated by increasing the OID size, but that reduces the storage efficiency.

3.1.2 Logical OID

Logical OIDs are more flexible. Objects can be relocated, and in theory, it should be possible to exploit the whole range of possible OIDs, given a certain OID size. However, in practice, the structure of logical OIDs is often similar to physical OIDs, for example the OID structure used in Versant [46]. If such a structure was not used, OIDs from the same collection and from the same database would be distributed over the range of allocated OIDs, making the OIDX very unclustered.

The number of OIDs can be very large, and if logical OIDs are used, a fast and efficient index structure is necessary. The OIDX is typically realized as a hash file or as a tree structure [62]. Most common is the use of B-trees, but other specialized structures have been proposed: One example is the *hcC-tree* [193], another example is *direct mapping* [62], where OIDs contain the physical address of the mapping information, and the mapping information is kept in a structure organized as an extensible array.

3.1.3 Combination of Physical and Logical OID

To improve performance, it is possible to use a combination of physical and logical OIDs, as is done in Shore [136]. In Shore, physical OIDs are used at the storage manager level. However, the *Value Added Server*, which is the interface to the users/application programs, can support logical OIDs by maintaining an OIDX.

3.2 Object Storage Structure

The way an object is stored, determines the update and query costs. In general, we have two primary strategies:[2]

- Direct storage model.
- Decomposed storage models.

[1] One bit is for internal use, so that only 15 bits are used for the container number.

[2] Note that authors of the papers discussing these models, do not always use a terminology consistent with previous definitions.

3.2.1 Direct Storage Model

In the direct storage model, there is no fragmentation of an object. The object, with all its attributes, will be stored as one contiguous sequence of bytes, similar to the in-memory version of an object in most programming languages. Large objects can also be stored as a contiguous sequence of bytes, but they are normally implemented with some access method to improve access efficiency.

3.2.2 Decomposed Storage Models

In the decomposed storage models [50], complex objects are decomposed so that each tuple in a database file (set of pages), only contains one of the attributes, together with a surrogate (OID in the case of an ODBMS). In one particular model, the binary storage model, attributes are stored as `(OID,value)` tuples. Providing that the tuples from objects in a certain collection are clustered together, this keeps the number of disk-reads to a minimum. This is very beneficial in applications where set queries are frequent, but if the objects are used by application programs in a persistent programming language, the objects have to be reconstructed before delivery. To reconstruct a set of objects, join operations are needed. Several studies have been done to study these tradeoffs [11, 205].

3.3 Object Clustering

In general, an object page contains more than one object. The performance of an ODBMS depends heavily on the number of object pages it has to read and write. In order to keep this number as low as possible, we try to store objects that are expected to be accessed together, on the same page. This process is called *object clustering*, and is done by using one or more of the following strategies:

- Clustering hints.
- Cluster trees.
- Dynamic clustering.

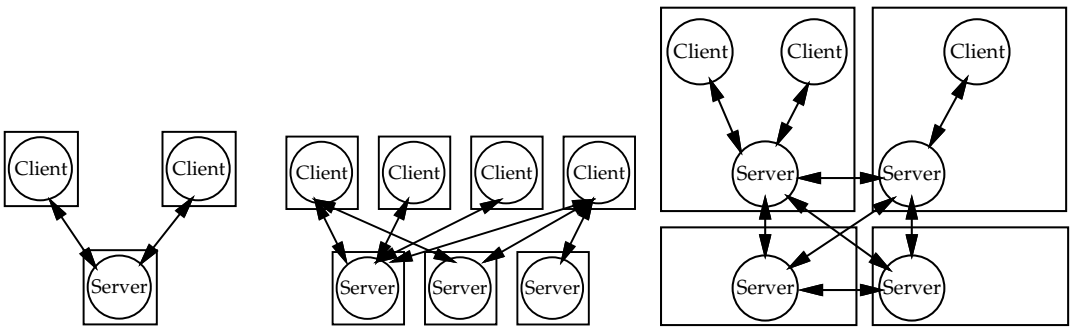
In most systems, objects that have been made persistent by a given clustering strategy remain where they are, even if the clustering policy changes (modification of the cluster tree in the case of a cluster trees strategy, or changing access pattern in the case of dynamic clustering).

3.3.1 Clustering Hints

When using clustering hints, the *application programmer* has to specify an existing object which the new created object should be stored close to (if possible). The performance of this approach is heavily dependent of an application programmer's predictions of future access patterns, and is likely to break down in more complex multiuser systems. Systems where this strategy is supported, includes ObjectStore, Objectivity and O_2.

3.3.2 Cluster Trees

Cluster trees is a more general approach to obtain good clustering. In this case, the *database administrator* specifies rules for object clustering. Typical examples of clustering strategies are to store together objects and related subobjects that are expected to be accessed together later, and members of a set that are later going to be accessed in scan operations. This strategy is supported by O_2 [165].



(a) Single server/multiple client.

(b) Multiple server/multiple client.

(c) Multiple server/multiple client (peer-to-peer).

Figure 3.2: Client/server architectures.

3.3.3 Dynamic Clustering

If dynamic clustering is supported, the *ODBMS* takes all responsibility for the clustering, and uses sampling of previous access patterns to decide where to store the objects. Algorithms and strategies for dynamic clustering strategies include the Cactis algorithms [59] and stochastic clustering [207]. A combination of cluster tree and dynamic clustering is also possible, as described by Benzaken et al. [10]. We do not know of any commercial ODBMS that supports a dynamic clustering strategy.

The performance of some object clustering techniques, under different workloads, have been studied by Tsangaris and Naughton [208]. Of those studied, stochastic clustering [207] had the best average performance.

3.3.4 Reclustering

Although not yet supported by any of the commercial systems, adaptive on-line reclustering is possible. One approach is described by McIver and King [139], and the cost of monitoring and reorganization has been studied by Gerlhof et al. [73].

3.4 Client/Server Architectures

The architecture of an ODBMS is usually a client/server variant. A client requests data, performs some operations on the data, and sends updated data back to the server. The exact division of the work between a client and its server, for example which of them is indexing data, varies between systems.

Several client/server architectures are possible. Figure 3.2 illustrates the most typical client/server architectures, with processes drawn as circles, computer nodes as squares, and communication channels illustrated with arrows:

- Figure 3.2a is a *single server/multiple client* architecture. This is the traditional and least complex architecture, supported by most systems. The clients can run on the same node as the server if desired.

- Figure 3.2b is a *multiple client/multiple server* architecture. The client itself has the responsibility of connecting to the servers containing the data, and to manage distributed commit (2-phase commit). This architecture is also supported by most commercial systems. A client can also in this case run on the same node as one of the servers or as other clients.

- Figure 3.2c is another variant of a *multiple client/multiple server* architecture, similar to the architecture of Shore ODBMS as presented in [39].[3] In this case, a client connects to *only one* server, running on the same node. The server has the responsibility of fetching remote objects/pages. A possible variant of this option, is clients residing on other nodes than the servers they are connected to. The main point is that one client only connects to *one* server, and this server communicates with the other servers as needed on behalf of the client.

3.5 Method Execution

The client and the server are in general different processes, usually executing on different nodes. When an object method is to be executed, this can be done either by the *client*, by the *server*, or by *a separate process on behalf of the client* on the server node:

1. Client node/client process: The method executes in the client's address space.

2. Server node/client process: A process is executing on behalf of the client on the server node.

3. Server node/server process. The method executes in the server's address space.

The problem with the first approach, is that all data have to be sent from the server to the client, something that easily makes the network a bottleneck. The two other approaches are (partial) solutions to this problem, but at the same time they create some new problems, which we will discuss in the following sections. Not all executed methods need to be executed in the same way. In systems that support more than one of the options above, it is possible to choose one of the options, as a way of tuning the performance.

3.5.1 Client Node/Client Process

Executing methods at the client, on the client node, is the most common in ODBMSs, and is supported by all commercial systems [8].

In applications where good page clustering has been achieved, and with only moderate data volumes, this approach works well. However, in the case of queries involving filtering operations (for example attribute selection), this approach wastes valuable network bandwidth. If filtering could be done on the server, less of the data actually has to be transported.

3.5.2 Server Node/Client Process

It is possible to run the whole client at the server node, but this can make the server node overloaded, and we do not benefit from the processing power of the client node. To solve this problem, and still avoid the drawback of the client node/client process, it is possible to execute some of the methods (or some of the query) on the server node (but note that they run as separate processes, i.e., not in the same address space as the server process itself). This approach is supported by ITASCA [8, 97].

[3] This architecture was to our knowledge never implemented in Shore.

If the client and the server run on the same node, we have more options on how to do interprocess communication. We can use message passing, as is done in general client/server communication. One step further, is to make the server buffer itself available to the client. In this case, *all clients access the same buffer*, and the design of such a buffer has to be done very carefully:

- If using a shared read/write buffer, clients as well as the server itself can write to the buffer. A shared buffer eliminates the need for a separate client buffer. A shared read/write buffer gives good performance, and locking can be done efficiently, but it gives integrity and security problems. A method can damage contents in the buffer in the case of failure, and there is no check of access permissions. We consider this approach too vulnerable without a safe interface language.

- If using a read-only buffer, clients can read from the buffer, but not write. The clients need separate buffers for modified objects/pages, or alternatively send modified objects/pages back to the server immediately (which usually will prove to be inefficient). From a performance point of view, a read-only buffer will in many cases be sufficient, because many of the typical heavy queries are read-only queries. Security is still a problem, although encryption of shared memory is possible. However, the cost of encryption would probably be unacceptable. Integrity can also be a problem if a client read data that is being updated by another client without adhering to the locking protocol.

3.5.3 Server Node/Server Process

If the server knows the contents and structure of the objects stored on the pages, it is possible to execute methods inside the server process. Systems that support this approach include ITASCA, MATISSE, POET, Objectivity, and Versant [8].[4]

Methods written in C and C++, which are popular programming languages for ODBMS applications, can literally do whatever they want, causing data integrity problems as well as damaging the server process itself. This means that special care has to be taken if general methods should be allowed to be executed by the server process. Several solutions to this problem exist:

1. Use a type-safe language as data manipulation language. This makes it easier to guarantee that the methods executed in the language can not modify privileged data in the DBMS. This approach has been used by Liskov et al. in Thor [128, 129, 130]. Variants of this approach is to use safe “data access languages”. language

2. Software-based fault isolation. In this approach, code and data are loaded into their own fault domain, a logically separate portion of the server address space, and the object code of the method to be executed is modified to prevent it from writing or jumping to an address outside its fault domain [211].

3. Interpreting the code. As Java has gained popularity, this option has become more commercially popular. Most commercial ODBMSs already support a Java binding, but in most cases, client methods are still only executed by the client. Interpreting the code is also done in Jasmine [95], where a reduced functionality C interpreter is used.

[4]MATISSE and Objectivity only support SQL queries at the server, while Versant can only execute registered events (change notifications/triggers).

4. Trusted methods. In an ODBMS, user supplied methods can be declared as trusted by the database administrator. They can then be executed in the server's address space, while untrusted methods are still executed in a separate address space. A variant of this approach is the possibility of user written "subservers" compiled into the DBMS. This is similar to the *Value Added Server* concept in Shore [39], and *DataBlades/Cartridges* in commercial ORDBMSs.

In the case of applications based on C and C++ language bindings, the trusted method approach and the software-based fault isolation are the the only realistic alternatives. However, the increasing popularity of Java makes C/C++ less popular as ODBMS application languages, and it is likely that the type-safe language alternative (using Java) will be the most popular in the future.

3.6 Data Granularity

The issue of data granularity arises in several contexts in ODBMSs. From an application program or query language point of view, data is usually accessed at object granularity. However, at the data storage level, most ODBMSs handle data at page granularity, which means that fixed size pages are read from and written to data volumes. We will now study the data granularity issues in ODBMSs, in three different contexts:

- Client-server data transfer.
- Buffer management.
- Concurrency control.

An issue not discussed here, is *page size*. The aspects of page size have been mostly ignored in ODBMS related research publications. Obviously, page size can affect performance, and among commercial ODBMSS, we see that the page sizes differ. For example, Objectivity can use different page sizes, up to 64 KB, while Versant has a fixed page size of 16 KB.

3.6.1 Data Transfer Granularity

Most ODBMSs are variants of data shipping object or page servers. Object servers have objects as the unit of transfer between the server and client, while page servers have pages as the unit of transfer.

The advantages of an object server are:[5]

- If the objects on the object pages are not well clustered, shipping the whole page is a waste of communication bandwidth.
- Understanding the concept of an object makes it possible for the server to apply methods on the object. This is very important in order to be able to do filtering operations in object-relational queries.
- Fine-grained (object level) concurrency control is easy to implement.

The advantages of a page server are:

[5] Parts of this summary are based on the descriptions by DeWitt et al. in [54].

- If objects on the object pages are well clustered, shipping the whole page can save many object requests and communication overhead for each object. This is also an issue even in the case where the client runs on the same node as the server. If requesting only one object at a time, two process context shifts are needed for each requested object (between client and server processes). This is obviously a too much, even on a relatively fast node this would limit the number of object requests per second to a number in the order of 50000.

- Fixed size pages are easier to manage than variable size objects, and space allocation for pages is easy on disk as well as in main memory.

Most commercial ODBMSs are page based. One exception is Versant, which is an object server, but with some features to avoid the performance problems related to single object accesses as described above:

1. *Get closure*, to retrieve references to all possible objects that can be navigated to, starting from a group of objects.

2. *Group read*, to retrieve a specified group of objects, for example based on the result from a *get closure* operation.

The page server architecture has, since the study of performance of alternative architectures by DeWitt et.al. [54], been considered as superior to object servers. However, that study was done under the assumption that each access to an object not resident in the client cache needed one remote procedure call, although it is noted that it would be possible for the server to simulate a clustering mechanism by figuring out what related objects might be needed. The study also appears to be misinterpreted (on purpose?) by many of the commercial companies. The paper's conclusion is actually that there is no clear winner in this study. An even more important factor, *not* considered in the paper, is that different applications often have different access patterns to the database. This means that it can be impossible to get a good clustering. Recent evaluations of real world applications, for example by Hohenstein et al. [88], support the view that object servers in many cases will perform as well as, and in many cases much better than, a page server. This is also verified by Kempe et al. [105]. One drawback of page servers that should be taken more seriously is the security and integrity risks of clients operating on pages.

In our opinion, the most important argument in favor of the object server architecture is the possibility to do some of the work at the server side. This is especially important for complex set operations, where filtering operations can significantly reduce the amount of data transfer. This has been a neglected issue in ODBMS, but we expect set operations to be given more attention in the future. This issue is also discussed in more detail in Appendix A.

3.6.2 Buffer Granularity

In the previous section we discussed the data shipping granularity. A related issue is the buffer granularity. We have the following alternatives:

- Page buffer.

- Object buffer.

- Dual buffer.

Note that buffering at the server and the clients may be handled differently. For example, the server can use a page buffer, while the clients use object buffers. However, if data transfer granularity is pages, it does not make sense to have an object buffer at the server side.

It is also important to note that multiple copies of data might reside in different client caches. Replica management is necessary to ensure cache consistency. Cache consistency is usually achieved by using pessimistic locking-based cache consistency protocols [40].

Page Buffer

Most ODBMSs use a page buffer. If objects on object pages are well clustered, a page buffer makes good use of the buffer memory. Fixed size pages are also easy to manage. Space allocation is easy, and we have no memory fragmentation problem.

Object Buffer

With an object buffer, objects are stored as independent objects in main memory, and not in the pages. This approach is beneficial if objects on the object pages are not well clustered. In that case, storing the whole page in main memory is a waste of space, because many objects in memory are not really needed there. The result will be a lower buffer hit rate than necessary when accessing objects. Another advantage is that it is possible to store objects larger than one page as one contiguous object, which is beneficial if server side execution of methods is possible.

Disadvantages of using an object buffer is that the per object overhead in an object buffer can be quite high, and we must expect some degree of memory fragmentation as well. Updates are also more complicated if we employ in-place updates. In that case, when a dirty object is to be written back to disk, it is necessary to first do an installation read of the page where the object should be stored.

Dual Buffer

A third alternative is a combination of page and object buffers. In this case, we try to keep well clustered pages in a page buffer, and objects from less clustered pages in an object buffer. This approach is used in several commercial systems, including Itasca, Ontos and Versant [52].

A thorough study of client side dual buffering by Kemper and Kossmann [108] showed that dual buffering can give a substantially higher buffer performance than a page buffer. However, the use of a dual buffer introduces several new options that makes tuning more complicated, for example when to copy an object from the page buffer to the object buffer, and when to copy a dirty object in the object buffer back to its home page. This makes it less clear how well dual buffering would perform in systems with complex workloads. Also, the study showed that dual buffering was mainly beneficial with read queries, with update queries the gain was less or negative.

3.6.3 Concurrency Control Granularity

Concurrency control can also be done at different granularities. This is usually adaptive, and can be fine grained, e.g., object, or coarse grained, e.g., page or file granularity.

3.7 Buffer Management

Keeping the most frequently used data in main-memory buffers reduces the number of disk accesses. Efficient buffer management is crucial to achieve good performance, and in this section we will discuss

buffer allocation and replacement.

3.7.1 Buffer Allocation Algorithms

With fixed size granules, for example pages, buffer allocation and deallocation is straightforward, and we have no memory fragmentation.

With variable sized granules, we will in practice have some degree of memory fragmentation. The amount of fragmentation is dependent of the amount of CPU we are willing to use to reduce the fragmentation. Using buddy allocation, which has a low CPU cost, gives a memory utilization of approximately 80%. However, it has been shown that it is easy to get a memory utilization above 90% by only a marginal increase in the CPU cost [70, 104].

3.7.2 Buffer Replacement Algorithms

The main-memory buffers can usually only keep a selected subset of the contents that are stored on secondary and tertiary storage. When an item is brought into main memory, another item has to be removed from the buffer to make space for the new item.

Buffer replacement is often LRU based. In the case of a page buffer, the pages are usually linked in an LRU chain. The overhead of an LRU chain is acceptable when the size of the pages is much larger than the extra data structures needed for the LRU chain. With a finer granularity, there is a larger number of granules, and a higher number of accesses to each of them. In this case, the traditional LRU chain can be a bottleneck:

- The memory overhead may be too high. For the LRU chain, two pointers are needed for each item.
- The CPU overhead can be too high, because we have to update the chain on every access.
- When the buffer is shared between several threads or processes, the pointers need to be protected by semaphores, and the head of the chain will often become a semaphore bottleneck.

Good approximation to LRU, useful for finer granules, are the *clock* and *enhanced clock* algorithms [61], also called second-chance algorithms. With the clock algorithm, only one overhead bit is needed for each granule, an *access bit*. For the enhanced clock algorithm, two bits are used, an *access* and a *dirty* bit.

When using the clock algorithm, the access bit is set each time an item is accessed. The buffer is treated like a circular queue. We have a *clock arm* (a pointer) that points to an item. When we need a candidate to discard during replacement, we move the clock arm clockwise until we find an entry where the access bit is not set. When we move the clock arm over items with the access bit set, we reset the access bits while we move the arm. In this way, an item will be discarded the next time the clock arm points at it, if it has not been accessed in the meantime.

With the enhanced clock algorithm, we also consider the dirty bit when deciding which item to discard. In general, it is cheaper to discard an item that is both clean and has not been accessed for a while, because it does not have to be written back before it is removed.

The advantages of using a clock algorithm are:

1. Lower cost when accessing an item, only the access bit has to be updated.

2. Less synchronization overhead is necessary. For example, no locks need to be acquired when an entry is accessed. That would be necessary when moving entries after an access in an LRU list.

3. If the access bits for the entries are stored in a packed format, i.e., access bits for several entries are stored in one machine word, the space overhead is reduced considerably. In this case, locking the word where an actual access bit resides, is necessary to get a serialized behavior,[6] in order to avoid loosing a *set bit* operation if two threads try to update different bits in a word by doing a *read word, set bit, write word* sequence. However, loosing an occasional access bit update should not seriously affect the buffer hit performance, so in practice, locking is not necessary!

3.8 Indexing

Indexing is a well-known technique used to reduce the query costs in DBMSs. In RDBMSs, only primitive attributes are indexed, but the increased expressiveness of the ODBMS data model makes new indexing techniques possible as well. There are also some aspects that is different in RDBMS and ODBMS indexing, and should be kept in mind:

- In RDBMSs indexing is not an integral feature, although very frequently employed. In an ODBMS, on the other hand, indexing is always employed. As discussed in Section 3.1, every object has an unique OID which can be used as a handle to retrieve the object.[7]

- In RDBMSs the relation as an extent, i.e., all members of the relation, is always maintained. In the case of object classes in ODBMSs, this is optional. In some systems, the extent is always implicitly maintained, while in other systems, this has to be done explicitly, with additional cost as a result.

- Although indexing primitive attributes in ODBMSs can be similar to indexing attributes in RDBMSs, the indexing is more complex due to existence of class hierarchies [112].

In the rest of this section we give a brief overview of path indexing and function materialization, which are not issues in RDBMSs, but can be important in order to achieve good query performance in ODBMSs.

3.8.1 Path Indexes

Most ODBMS query languages allow queries on path expressions (usually expressed by the *dot* notation). Several techniques for indexes supporting path expressions have been proposed. These include different *path indexes* [13, 14] as well as *access support relations* [109].

Path expressions is actually a kind of implicit join. If no path index exists, it can be cheaper to use explicit join techniques (pointer-based joins) in set queries, instead of doing pointer traversals [188].

Related to path indexing, is *field replication* [187], where the field (attribute) at the end of a path expression is replicated, and stored inside the first object in the path.

[6] If a *set bit* operation exists, this is not necessary. However, single bit operations is not always available, usually they are provided only by CISC processors, for example the Intel x86 family.

[7] In the case of physical OIDs, the indexing is implicit.

3.8.2 Function Materialization

Predicates in ODBMS queries can involve methods as well as attributes. It is possible to use precomputed values for methods to increase query performance. This technique is called *function materialization* [106].

3.9 Swizzling

Pointer swizzling is the process of converting pointers in an object from disk format (physical or logical OIDs), to memory addresses, so that subsequent object navigation operations do not have to go through an index or "resident object table" in order to find the actual object. Although swizzling has not previously been considered as a server issue, it is likely that it can increase the performance if methods can be executed by the server.

The possible gain from swizzling does not come for free. If an object has been modified, all swizzled pointers to this object have to be changed back to disk format before the object is removed from the buffer. Thus, swizzling is only beneficial if the swizzled object is referenced several times, and the update rate is sufficiently low. Several swizzling strategies exists [141, 213]. Which strategy to use, and whether to swizzle at all, depends heavily on the access pattern, and adaptable swizzling strategies might be a good alternative [107].

3.10 Query Processing

Query processing in an ODBMS can be done in much the same way as it is done in a RDBMS [110, 219]. The user[8] submits a query to the system, usually in some declarative language. This query is optimized, normalized, and transformed to some object algebra expression. After type checking, algebra optimization is performed, and an execution plan from this optimized algebra expression is generated and executed. Similar to a RDBMS, the difference in execution time between a query with good optimization and a query with bad optimization, can be several orders of magnitude.

Even though the basic techniques are the same as in RDBMSs, ODBMS query processing has many aspects which makes it more complex than query processing in RDBMSs. The most important differences are [111, 219]:

- ODBMSs have a much richer type system than RDBMSs, which only have the single aggregate type *relation*. In ODBMSs, queries can be performed on various kinds of collections, where members can be of different types.
- Encapsulation and methods: how much should the system know about the implementation of a method, and should it be able to break encapsulation?
- An object may reference other objects, and accessing these objects involves path expressions/implicit joins.
- In ODBMSs, indexing can also be done on access paths, not only on primitive attributes, as in RDBMSs. Class hierarchies also complicate the use of indexing.

[8] User in this context can be either a person giving a command to the DBMS, or an application program sending a request to the DBMS.

- Inheritance can make it difficult to determine the access scope of a query. This makes efficient object access more complicated.

- Cyclic queries need special attention.

These differences, the availability of different kind of indexes, and the choice between forward and reverse traversal (whether to start on the target class/root of query graph, or at any intermediary) increase the number of possible query plans. This makes the process of evaluating query plans more costly and difficult in an ODBMS compared to a RDBMS.

3.11 Parallel ODBMSs

The performance of a DBMS can be increased by increasing the available hardware resources. This means more powerful hardware, or duplication of resources. Employing more powerful hardware is one solution that has been considered "easy", as it has no consequences for the implementation of the ODBMS itself. However, this strategy is only cost effective up to a certain point. After that, duplication of resources, i.e., a larger number of CPUs and disks, is needed. It should also be noted that this strategy is more difficult than it looks, because the CPU speed, memory- and disk bandwidth have to be kept in balance.

If using more than one CPU and more than one disk, work has to be distributed over the CPUs and the disks in a way that make all of them busy most of the time, and avoids any single bottleneck. Even though parallelization of "simple" set queries are well understood from the work on parallel query processing in RDBMSs, parallel query processing in ODBMSs is less mature. There are several reasons for this, but the most important is that ODBMS query processing can be very complex in itself. The fact that the architectures of most systems are based on data shipping, makes filtering on the servers difficult, and it is difficult to keep the data transfer volume at a moderate level.

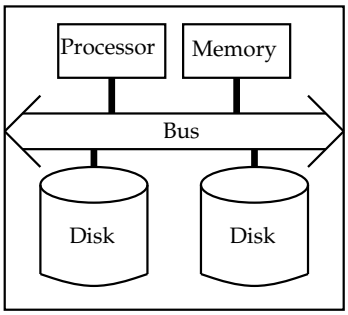
In this section, we discuss parallelization in ODBMSs. We start with a presentation of alternative parallel architectures, and then give an overview over issues in parallel query processing. To set the work presented later in this thesis in context, we also summarize work on previous parallel ODBMSs.

3.11.1 Alternative Parallel Architectures

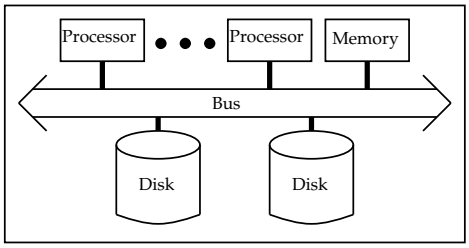
Even on single processor computers, as illustrated in Figure 3.3a, multiple disks are common, and if used to host a DBMS, are necessary in order to provide redundancy in the case of media failure. A larger number of disks can also be used to improve the data transfer bandwidth and transaction throughput, for example by using RAID technology. The advantage of this approach, is that it is relatively easy to utilize the disks.

Current servers are often symmetric multiprocessors (SMP), with a number of disks attached. In an SMP, all processors have equal access to memory and disks. This is called a *shared everything* configuration (see Figure 3.3b). The limiting resource in an SMP node is the bus, which soon gets saturated as more processors are added. The advantage with this approach, is that it is relatively easy to utilize the CPUs if the DBMS is implemented as a multithreaded or multiprocess server.

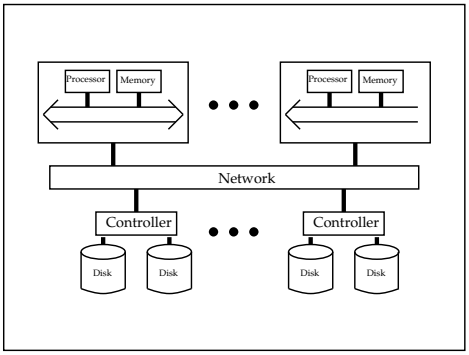
A further improvement is *shared disk*, where processors have equal access to the disk system, but not on the same bus (see Figure 3.3c). In a shared disk configuration, issues such as fragmentation and clustering are easier than for a shared nothing approach. Shared disk is the traditional mainframe approach, and has not been very common in the case of systems made from off-the-shelf hardware.



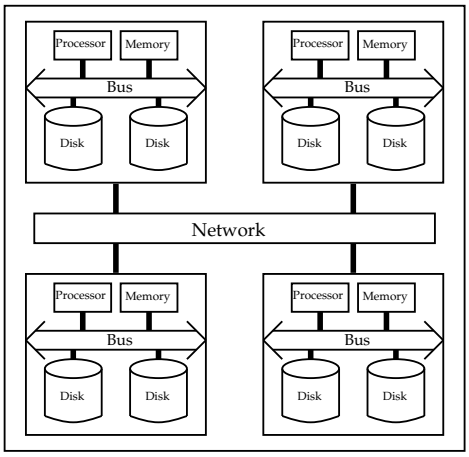
(a) Single processor computer.



(b) Shared everything configuration.



(c) Shared disk configuration.



(d) Shared nothing configuration.

Figure 3.3: Alternative parallel architectures.

However, with the increasing popularity of storage area networks, using Fibre Channel, we expect it to be more common in future systems.

The most scalable approach is *shared nothing* (see Figure 3.3d). In this case, we have a number of nodes which each has local memory and a number of disks. The nodes communicate through some kind of interconnection network, using message passing. A shared nothing computer is also commonly called a *multicomputer*, but nowadays, a cluster of workstations connected to high bandwidth network is also suitable as a platform for a parallel database server.

3.11.2 Parallel Query Processing

We will in this section concentrate on issues related to shared nothing servers. However, it should be noted that some of the work done in the context of multiprocessors is also relevant, including the research on parallel query evaluation done by Härder et al. [90], and on optimizing and parallelizing ODBMS programming languages by Lieuwen et al. [127].

Data Distribution

Optimal allocation and fragmentation is very important, but complex objects, object classes and inheritance increase the size of the solution space for the data distribution problem. These issues have been studied by a number of researchers, including Gruber and Valduriez [82], Karlapalem et al. [103], and Ghandeharizadeh et al. [76]. Ghandeharizadeh et al. also show how replication can be efficiently employed to increase performance. It has also been shown that in a parallel system where it is possible to store most of the working set in main memory, utilizing the aggregate memory of all the nodes can significantly improve the performance [209, 210]. A more detailed discussion if the data distribution problem is given in Chapter 11.

Query Processing

Optimal data distribution is in general heavily linked to queries on the data. This has been studied in detail by Kim [111] and Chen and Su [48]. In many cases, redistribution of data can be efficient [121]. Parallel join algorithms for set-valued attributes is described by Lieuwen et al. [126].

3.11.3 A Brief Overview of Parallel ODBMSs

We will now give a short description of previous parallel ODBMSs. With one exception (Objectivity), all the described systems are research prototypes. Several of the prototypes (ADAMS, AGNA, and PPOST) are systems built as a part of a PhD work, and seems to have been abandoned after the PhD work was finished. One of the systems, Shore, has never been implemented as intended (multi-server version), while the work on Multicomputer Texas seems to have been restricted to the cited paper only. All in all, these systems illustrate well how immature the area of parallel ODBMSs is. Although the complexity of the area is one reason why so little research has been done, it is likely that the advent of ORDBMSs, which had a negative impact on the amount of ODBMS research in general, also has been important. However, the results from some of these projects have been convincing enough to make us believe that this is an area that deserves more active research.

In addition to the systems described here, some of the commercial systems also provide some support for multiple servers. However, the application programmer has the responsibility for the distribution of data, and the support for distribution mostly means “the system supports 2-phase commit”.

ADAMS

ADAMS is a parallel "data management system", running on a network of workstations [86, 174]. It has many ODBMS features, but lacks concurrency control and recovery, which are important features of a database system. Its main application area is SSDBs, an area where these features often are of minor importance.

ADAMS employs the decomposed storage model for object storage, and declusters objects by the OID. The system processes set operations by streaming of data like most parallel RDBMSs, and has shown good performance and scalability.

AGNA

AGNA [145], a persistent programming system, is based on a LISP like environment. The system is designed to run on a shared nothing multicomputer. Objects are referenced by their heap address. The heap is global, and distributed over the nodes in the system.

Bubba

Bubba is a highly parallel DBMS [28]. Its application area is data-intensive applications. Data is horizontally partitioned (which favors objects with few references to other objects), and performance depends on executing operations at the node where the object resides. This is supported by the use of automatic parallelization by the Bubba compiler and an analytical model for data placement.

Eos

Eos, which is short for *Environment for building Object-based Systems*, is a distributed single-level store [81]. Distribution of data over the nodes is supported by facilities in the Mach operating system. Eos is supposed to be scalable, but there are no data on performance that can support this claim.

Multicomputer Texas

The Multicomputer Texas [18] is a parallel object store based on the Texas object store [189]. Multicomputer Texas has been implemented on a Fujitsu AP1000 multicomputer and a network of workstations. A modified Texas object store is run on each node, providing a global persistent address space. In this way, we can see it as a distributed shared memory implementation. No support for parallel query processing or efficient declustering of data is provided, and performance is highly dependent of the locality of data to be accessed at the nodes. In this respect, we feel that the practical value of the implementation is limited.

Objectivity

Objectivity [167] is the only commercial ODBMSs that is able to use parallelism to significantly increase performance. Objectivity is a page server ODBMS, employing NFS (Network File System). The servers run on ordinary network connected workstations, and the distribution of data can be used to increase performance as well as availability (by replication). Objectivity has been chosen as the ODBMS to be used in the CERN RD45 project, where experiments will generate an amount of 1 PB of data a year, and up to 1.6 GB/second data rate [44, 45, 46, 47].

OSAM*.KBMS/P

OSAM*.KBMS/P is a parallel, active, object-oriented knowledge base server [201]. The server runs on a shared nothing computer (nCUBE2), and the clients on workstations connected to the nCUBE via Ethernet. The knowledge base is partitioned class-wise, i.e., all members of a class is stored on the same node. A global transaction server is used to supervise executions.

PPOST

PPOST [21, 22, 23] is a parallel, main-memory object store, implemented on a cluster of workstations. Because transactions are committed to disk sequentially, the architecture is only suitable for application areas with a small number of concurrent transactions, and where transactions are short in time, but with high data bandwidth.

Shore

Shore [39], Scalable Heterogeneous Object REpository, is a persistent object system with many novel features. The most interesting in the context of this section, is the introduction of a symmetric peer-to-peer server architecture. All application programs in the system are connected to *one server*, running on the same node. This server is the gateway to the DBMS. Unfortunately, multi-server Shore has never been implemented, although some research have been done on parallel set processing by the use of ParSets [57], and global memory management [209, 210].

The storage manager of Shore has later been used in Paradise [55, 173], a parallel DBMS for GIS applications. Paradise is described by the authors as object-relational, rather than object based (or object-oriented).

3.12 Summary

The design of an ODBMS introduces new issues not found in RDBMSs, and we have in this chapter discussed design issues in ODBMSs, with an emphasis on issues that are specific for ODBMSs. The discussions in this chapter will be used as a background for the description of Vagabond, as well as establishing the terminology which will be used in the rest of this thesis.

Chapter 4

Temporal Database Systems

The rest of this thesis will concentrate on the design of a temporal ODBMS. It is therefore appropriate to start with an introduction to temporal DBMS in general, the terminology, and related work in the area. This chapter is not intended as a complete study on temporal databases, its purpose is only to give the reader the necessary background to comprehend the rest of this thesis.

4.1 What is a Temporal DBMS?

A temporal DBMS is a DBMS that supports some aspect of time. Informally, this means that data is associated with time, and that a tuple (temporal RDBMS) or object (temporal ODBMS) can exist in several versions, each version being valid in a certain time interval. An example is the salary of a person. Each time the salary is changed, a new version of the person's salary tuple/object is created. In a temporal DBMS, this versioning, related to time, is supported and maintained by the system, which also provides support for querying the data.

Even before people started to think about temporal DBMS as an area of its own, time has been related to data in a database, for example by the use of an attribute containing a time value. Typical examples are attributes such as "birth day" and "hiring date." However, there has not been support for temporal aspects in the query languages, and queries and management have been done in various ad-hoc ways. In our terminology, we call this uninterpreted attribute domain of date and time *user-defined time*. A temporal DBMS is now defined as a DBMS that supports some aspect of time, *not counting user-defined time*. A traditional, non-temporal DBMS is called a snapshot database system. Not all data stored in a temporal DBMS needs to be temporal and the data that is not temporal is called *snapshot data*.

Even though temporal databases have a long history, it is only very recently that research in this area really has taken off, and more importantly, the industry has begun to signal interest in the work. Two projects have in particular contributed to the current interest and results in the area: the *consensus glossary of temporal database concepts* [101], and the *temporal structured query language* (TSQL2) specification [191]. The consensus glossary is recommended by a significant part of the temporal database community, and the definitions and terminology in this chapter are based on that glossary.

4.2 Data Models

Many temporal data models have emerged during the years. A presentation of these is outside the scope of this chapter, and we will only give a brief overview of the most important aspects of these

models.

4.2.1 The Time Domain

Time models can be linear, branching, or cyclic. In a linear time model, time advances from the past to the future in an ordered step by step fashion. In a branching time model, time can split into several time lines, each representing possible event sequences. In a cyclic time model, we can also have recurrence. One example is a week, where each day recurs every week [218].

The time line itself can be either discrete or continuous. If discrete, each point in time has a single successor, like natural numbers. If continuous, there are no gaps, similar to real numbers. In most models, a discrete time line is used, where we have a non-decomposable time interval of some fixed, minimal duration of time called a *chronon*. Important special types of chronons include valid-time, transaction-time, and bitemporal chronons. A data model will typically leave the particular chronon duration unspecified, to be fixed later by the individual applications, within the restrictions posed by the implementation of the data model.

4.2.2 Aspects of Time

The most common aspects of time in temporal DBMSs is transaction time and valid time.

Transaction time: A database fact is stored in a database at some point in time, and after it is stored, it is *current* until logically deleted. The transaction time of a database fact is the time when the fact is current in the database and may be retrieved. Transaction times are consistent with the serialization order of the transactions. Transaction-time values cannot be later than the current transaction time. Also, as it is impossible to change the past, transaction times cannot be changed. Transaction times may be implemented using transaction commit times, and are system-generated and -supplied. It is important to note that each update of an object creates a new current version. We call the non-current versions *historical versions*.

Valid Time: The valid time of a fact is the time when the fact is true in the modeled reality. A fact may have associated any number of instants and time intervals, with single instants and intervals being important special cases. Valid times are usually supplied by the user.

Valid times can be open-ended intervals. One example of this, is the existence of a house. We know when it was built, but now when it will be removed.

Bitemporal: Temporal DBMSs can also support a combination of these aspects, bitemporal data. Bitemporal data have exactly one system supported valid time interval, and exactly one system-supported transaction time.

A bitemporal interval is a region, with sides parallel to the axes in a two-space of valid time and transaction time. When a bitemporal interval is associated in the database with some fact, it identifies when that fact was true in reality (during the specified interval of valid time), and when it was logically in the database (during the specified interval of transaction time).

A good example to illustrate the use of transaction and valid time, is a GIS database. In this database, objects such as houses and roads are stored. When an object is stored in the database, it is timestamped with the transaction time. However, the time this actual object existed in the real world is

in general different from the time it was entered into the DBMS. For this purpose, it is also necessary to store the valid time of the objects in the database.

4.3 Temporal Queries and Query Languages

Several query models for temporal databases have been proposed, and others are likely to be proposed in the future. In practice, most research on temporal databases is now based on *TSQL2* [191], an extension of SQL-92. TSQL2 provides language constructs for schema definition, schema evolution and versioning, and querying and updating temporal relations. The goal of the language design was to form a common core for future research, more than designing a language for the commercial market, but work is currently under way to incorporate TSQL2 into SQL3 [190].

TSQL2 employs a simple data model, based on the relational data model. In the conceptual model, the *bitemporal conceptual data model*, tuples are timestamped with a bitemporal interval.

Queries are performed on a collection of tuples. In addition to the traditional relational operators, temporal operations are also needed. We will now present the most important temporal operations, with examples based on TSQL2 [191].

4.3.1 Temporal Selection

With temporal selection, it is possible to retrieve data valid at a certain time, *valid time selection*, or current at a certain time, *transaction-time selection*. It is also possible to do a selection based on both valid and transaction time, *bitemporal selection*.

Several new operators are included in TSQL2 to be used in the temporal selection predicates, including operators for comparison of timestamps:

- **FIRST(event, event)**
- **element PRECEDES element**
- **period CONTAINS period**

To illustrate temporal selection, consider an example query from an employee database that lists all of the employees who worked during all of 1991:

```
SELECT Name
FROM Employee
WHERE VALID(Employee) CONTAINS
        PERIOD(DATE '01/01/1991', DATE '12/31/1991')
```

4.3.2 Temporal Projection

In a query or update statement, temporal projection pairs the computed facts with their associated timestamps,[1] usually derived from the associated timestamps of the underlying facts. The generic notion of temporal projection may be applied to various specific time dimensions. For example, valid-time projection associates with derived facts the times at which they are valid, usually based on the valid times of the underlying facts.

[1] Note that a timestamp can also be an interval.

4.3.3 Temporal Join

A temporal natural join is a binary operator that generalizes the snapshot natural join to incorporate one or more time dimensions. Tuples in a temporal natural join are merged if their explicit join attribute values match, and they are temporally coincident in the given time dimensions. As in the snapshot natural join, the relation schema resulting from a temporal natural join is the union of the explicit attribute values present in both operand schemas, along with one or more timestamps. The value of a result timestamp is the temporal intersection of the input timestamps, that is, the instants contained in both.

4.3.4 Coalescing

Associated with each tuple in a temporal relation is a timestamp, denoting some period of time. In a temporal database, information is "uncoalesced" when tuples have identical attribute values and their timestamps are either adjacent in time or share some time in common. Coalescing is similar to duplicate elimination in conventional databases, although potentially more expensive [25]. Its purpose is to effect a kind of normalization of a temporal relation with respect to one or multiple time dimensions. This is achieved by packing as many value-equivalent tuples as possible into a single value-equivalent one.

Example: Given two tuples with the same non-temporal attributes and valid in the time intervals [40, 50> and [45, 60>, respectively.[2] The result of a coalescing these tuples is *one* tuple, with the same non-temporal attribute values as the two input tuples, and the time interval [40, 60>.

4.3.5 Temporal Aggregation and Grouping

The main difference between traditional value-based aggregation and grouping, and temporal aggregation and grouping, is the inclusion of time in the domain of aggregates, and the possibility to group on time. For example, **MIN(VALID(R))** can be used to select the value of the oldest or earliest tuple in a table. In addition to the traditional aggregate functions, new functions can be useful in temporal databases. One example is the **RISING** operator in TSQL2, which is defined to return the longest period which a numeric value was monotonically rising.

Time can be used as basis for the partitioning in the grouping part of the aggregation. The timeline can be divided into partitions, i.e., into time periods. For example, to compute the average salary for each 3 month period along with the start date of the period, the following query can be used:

```
SELECT AVG(Salary), BEGIN(VALID(E))
FROM Employee AS E
GROUP BY VALID(E) USING 3 MONTH
```

4.4 Programming Language Bindings

In non-temporal ODBMSs, ODMG's OQL or similar query languages can be used for ad-hoc queries. Similar to the way OQL is a superset of the part of standard SQL that deals with databases queries, it is possible to design a temporal OQL that is a superset of TSQL2. One such approach has been described by Fegaras and Elmasri [67]. However, one of the main advantages of ODBMSs is the avoidance of

[2] $[T_1, T_2>$ is short for the time interval from T_1 to T_2, including T_1 but not T_2 (open-ended upper bound).

the language mismatch by providing computationally complete data manipulation languages with no mismatch between language and storage. In the ODMG standard, language bindings based on C++, Java and Smalltalk are described. Such language bindings are also needed for temporal ODBMSs. It should also be noted that in order to use methods in queries, these issues have to be resolved.

A general purpose programming language is only designed for current data. Integrating support for access of historical data into a programming language introduces a lot of interesting but difficult issues, including:

- Which object interface/signature to use when accessing a historical object version. The schema might have been changed since the historical version was created, so that the current interface to the class is different from the one previously used.
- Which method implementation to use when calling methods in historical objects. One straightforward approach is to use the implementation that was current at the same time as the actual object version was current. However, this is not necessarily what we want, if the reason for a new implementation of a method was a bug in the previous version. This problem can be solved by providing the necessary information at schema change time.
- How to integrate *time* into the syntax of the programming language.

In the rest of this section, we will discuss the integration of access to historical data into a general-purpose programming language.

4.4.1 Temporal C++ Binding

In this section, we describe two approaches that extend the C++ language binding with support for access to historical data in a transaction-time ODBMS. The first approach is based on the language binding used in POST/C++ [202], while the second is to our knowledge new. The concepts of these approaches can also be employed for a Java language binding.

Explicit Object Version Access

The easiest way to integrate object version access into the programming language is to provide explicit access to the versions. This is the way it is done in POST/C++ [202]. Given an OID, the program can be given a pointer to a historical version valid at a particular time by calling a function `snapshot(OID,time)`. It is also possible to create iterators that can be used to navigate the versions of an object in chronological sequence.

This approach should be easy to use and understand, but if it should be possible to call a method in a historical object version that accesses other objects, the historical version must itself do the necessary operations in order to retrieve the objects valid at the same time as when the version was created.

The Explicit Snapshot Approach

A better and "cleaner" alternative than the one described above is to use explicit snapshots. Before calling a method in a historical version current at time `ts`, we set the snapshot time with a call to the function `set_snapshot(d_Timestamp ts)`. After the `set_snapshot()` function has been called, an access to a particular object will be to the object version current at time `ts`, *even though the*

reference is through a `d_Ref`.[3] A call to `set_current()` will set accesses back to normal, i.e., an access to a particular object will be to the current object version. Methods called in historical objects should in general be immutable, i.e., read-only methods. The advantage of this approach is that all object versions accessed will be object versions valid at the same time.

All access, creation, modification and deletion of persistent objects must be done within a transaction. In the ODMG C++ binding, transactions are implemented as objects of the class `d_Transaction` [43]:

```
class d_Transaction{
public:
      d_Transaction();
      ~d_Transaction();
      void begin();
      void commit();
      void abort();
      void checkpoint();
      ...
private:
      ...
};
```

The `set_snapshot(d_Timestamp ts)` and `set_current()` functions are performed in the scope of a certain transaction, so it is reasonable to extend the ordinary C++ transaction class with these methods, for example with a derived class based on `d_Transaction`, which includes these functions as methods:

```
class d_TTransaction:public d_Transaction{
public:
      void set_snapshot(d_Timestamp ts);
      void set_current();

private:
      ...
};
```

Each temporal object can be viewed as a collection of object versions. A collection interface should exist to make it possible to iterate through the object versions in a flexible way. This collection interface is also used when assigning a value to a `d_HRef` variable (a reference to a particular version), i.e., assigning an object version to the `d_HRef`.

4.4.2 To Bind or not to Bind?

We have now outlined how objects could be accessed through a standard language binding. It should be noted that the problems involved in this integration also can be an argument *against* doing this. It is possible that only allowing access to historical versions through a temporal query language is less error prone and more efficient than providing access through an explicit language binding. A more in-depth study of the language binding, and whether to have it at all, is interesting further work.

[3] A `d_Ref` is a reference to an object.

4.5 Vacuuming

When an object has been deleted in a snapshot database, it can not be accessed later. Usually, the space occupied by the object will be overwritten by new data after it has been deleted. Temporal databases, however, follow a non-deletion strategy, where logically deleted data are kept in the database. Even though storage cost is decreasing, storing an ever growing database can still be too costly in many application areas. A large database can also slow down the speed of the database system by increasing the height of index trees (even though this can be avoided with multi-level indexes, at the cost of a more complex system). As a consequence, it is desirable to be able to physically delete data that has been logically deleted, and delete non-current versions of data that is not deleted. This is called *vacuuming*. Note that the term *vacuuming* has also been used for the migration of historical data from secondary storage to cheaper tertiary storage. In this thesis, we will use the term for *physical deletion* only.

4.6 Implementation Issues

We will do a more detailed discussion of some implementation issues in temporal DBMS later in this thesis. In this section, we will restrict the discussion to an overview of some of the most important work in the area.

4.6.1 Partitioned Storage

Storage of data in a temporal DBMS is not very different from storage of data in a traditional DBMS. However, because current data tend to be more frequently accessed than historical data, data is often partitioned into a *current store* and a *history store*. The two stores can utilize different storage formats, and even reside on different storage media [4]. In this way, frequently accessed data is clustered together, stored on fast storage media, while historical versions can be stored on slower but cheaper storage media. The total storage cost is reduced, similar to the goal of general storage hierarchies.

4.6.2 Timestamp Representation

Timestamps can be viewed from two levels: logical and physical level. The logical level is the user's view of the values, for example from a query language. At the logical level, the timestamp may look like "Dec 4 22:14:44 1998". It can also look like "1998/12/04", in a different format, at a lower granularity. However, physically, the timestamp is usually represented differently. We have several goals we want to achieve:

1. High precision. For many applications, precision down to day or hour is enough, while other applications need finer granularity. This is especially important for transaction-time databases, where we want objects from different transactions to have unique timestamps.

2. Large range. In the case of valid time, a timestamp should ideally be capable of representing all points in time, from the Big Bang to Armageddon. However, in a transaction-time database, we can accept a smaller range, from the day the system is first used, and to "some time in the future".

3. Low storage cost. To keep storage costs down, the number of bytes used to represent a timestamp should be as small as possible, given the other constraints.

4. Low processing cost, for example when creating timestamps, comparing timestamps (including ordering of timestamps), and translating between different calendar representations.

As can be observed, high precision and large ranges conflict with low storage cost. Given a certain storage cost, high precision and large ranges are conflicting goals. Low storage cost conflicts with low processing cost, because efficient storage of a timestamp will often imply transformation before and after processing.

Alternative timestamp representations can be classified as:

1. One-field alternative, often used in operating systems. In Unix, 32 bits are used to represent the seconds since its origin. This format is very space efficient, and results in low processing cost. However, the range, 136 years, is too small for a general purpose valid time temporal database. Although the range is large enough for a transaction-time temporal database, one should keep in mind that some of the data stored in temporal databases will be used some time far in the future, so that one should consider a larger range. In many applications, the precision (seconds) is too small as well. However, both precision and range can easily be increased by increasing the number of bits in the timestamp.

2. Multi-field timestamps, as used to represent time in many commercial RDBMS. In this case, there are separate fields in the timestamp for year, month, day etc. In each field, the actual year/month/day can be stored by using packed decimals or a string representation.

In a study of this issue, Dyreson and Snodgrass [60, 191], proposed a new timestamp format to solve the problems above. In their timestamp format, special values designate special times as *now* and *forever*. They also made the observation that users have a telescoping view of time, times close to *now* should be represented with finer granularity than times further in the past or in the future. They can be represented with an extended range and coarser granularity. The proposed timestamp representation can have different lengths: 32, 64, and 96 bits.

4.6.3 Indexing Temporal Databases

To support efficient retrieval of temporal data, indexing is necessary. Much research has been done in this area, and a comprehensive survey of indexing time-evolving data has been done by Salzberg and Tsotras [178]. This issue will be discussed in more detail in Chapter 8.

4.6.4 Temporal Query Processing

Even though most other aspects of temporal databases now seems to be well explored areas, the amount of publications on temporal query processing is still relatively small. One of the reasons for this, is that much of the other work (and implementations) have used a *stratum approach*, in which a layer converts temporal query language statements into conventional statements executed by an underlying DBMS [100]. Although this approach makes the introduction of temporal support into existing DBMSs easier, we do not see it as a long-term solution, because temporal query processing with this approach can be very costly.

Previous work on temporal query processing includes the work by Leung and Muntz [122, 123], which was a study of query execution on a data stream with tuples with increasing timestamps. That work was also done in the context of multiprocessor database machines [124]. An algorithm for evaluation of valid-time natural join has been presented by Soo et al. in [192]. Optimization of

partitioning in temporal joins has been described by Zurek [220]. Other important work includes aggregation algorithms [116], a study of parallel aggregation [72], and coalescing [25].

In the context of query processing in temporal ODBMSs, we are only aware of one paper, on parallel query processing strategies for temporal ODBMS by Hyun and Su [94].

4.7 Temporal ODBMSs

The area of temporal *ODBMSs* is still immature, as is evident from the amount of research in this area, summarized in the *Temporal Database Bibliography*, last published in 1998 [216]. The main reason for this low research activity is probably the number of problem still unsolved in the less complex case of temporal RDBMSs.

Most of the work in the area of temporal ODBMSs has been done in data modeling, while less have been done on implementation issues. systems have been implemented [24]. Common for most of these, is that they have only been tested on small amounts of data, which makes the scalability of the systems questionable. In most of the application areas where temporal support is needed, the amount of data will be large, and scalability is an important issue.

In the area of temporal ODBMS, we are only aware of one prototype, POST/C++ [202]. However, the indexing technique used in POST/C++ is not scalable. Good performance is only possible as long as the OID index fits in main memory (see Section 8.3.2 for a description of the indexing in POST/C++). In addition, there are implementations of temporal object data models on top of traditional ODBMSs, for example TOM, built on top of O_2 [194].

4.8 Summary

We have in this chapter given a short introduction to the terminology and most important issues in temporal database management. For more in depth discussion of the issues, we refer to the publications cited in this chapter.

Chapter 5

Log-Only Database Management Systems

Most current database systems are based on in-place updating of data combined with write-ahead logging. In this chapter we describe the alternative log-only approach, and describe its advantages and disadvantages. We describe the page-based and object-based alternatives, and why we consider the object-based alternative as the most interesting. We finish the chapter with an overview of systems that are based on log-only or related techniques.

5.1 The Log-Only Approach

In a log-only approach, data as well as metadata are written contiguously to the log. Already written data is never modified, new versions of objects or pages are simply appended to the log. Logically, the log is an infinite length resource, but the size of the physical storage is of course not infinite. This problem is solved by dividing the physical storage into large, equal sized, physical *segments*, as illustrated in Figure 5.1. A typical segment size will be in the order of 512 KB to 1 MB. When all data residing in one segment is outdated or moved to another segment, the segment can be reused. In the description in this chapter, we will assume that the log resides on disk only, but in general, the log can also reside on tertiary storage. The log also contains *checkpoint blocks*, which are used to store checkpoint information. The checkpoint blocks are stored in fixed positions in the log.

Writes are always done sequentially, normally one segment at a time. This is done by writing data and index nodes, possibly from many transactions, in one write operation. The segment size is a tradeoff between different, partly conflicting, goals: to improve write efficiency, it is desirable that the segments written are as large as possible. On the other hand, large segments can make response time longer, because writing large segments will block for read operations, and we have to wait for more transactions during group commit. Smaller segments reduce the blocking time for waiting read operations, but they also result in less efficient writing, and a larger number of segments (which means more overhead).

Because data is always written to a new place after having been updated, an index is necessary to be able to subsequently retrieve the data. This index structure is also written to the log, interleaved with the data. If the granularity of data is objects, an OID index (OIDX) is needed, and if the granularity is pages, a page index is needed (in the latter case, a page identifier is part of the OID of an object, similar to physical OIDs in traditional ODBMSs). The granularity of reading is object or pages. When a stored object or page has to be retrieved, only the desired object or page is read, not the whole

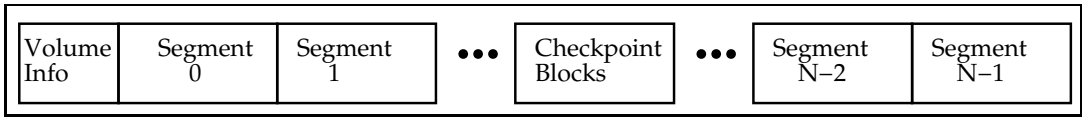


Figure 5.1: Disk volume structure.

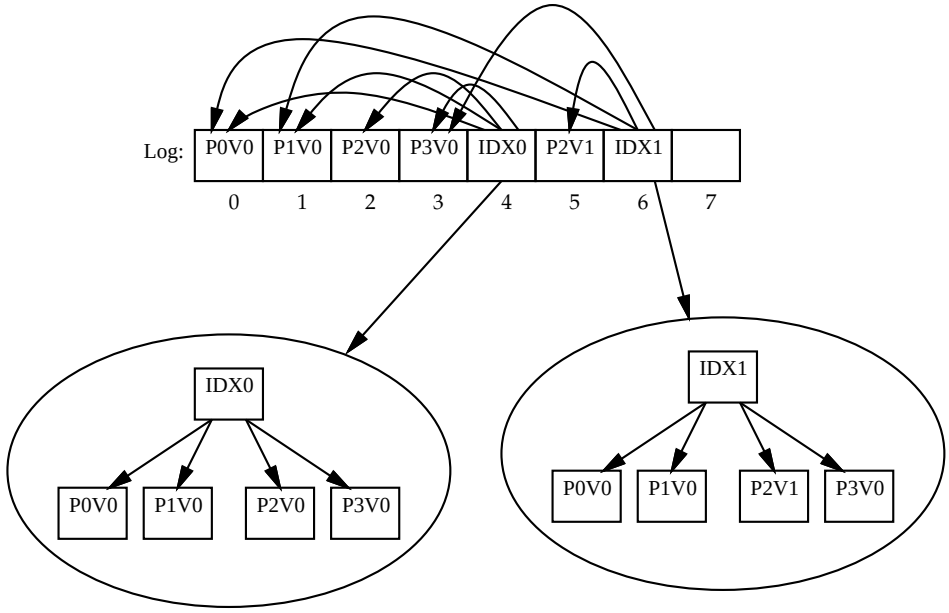


Figure 5.2: Data and index in a log-only ODBMS.

segment it is stored in.

5.1.1 Example of Log Writing

We now give an example to illustrate the log writing. In this example, the data granularity is a page, and a page index is interleaved in the log. Which page to retrieve when an object is requested is given from the OID of the object, which contains a page identifier.

Figure 5.2 illustrates how data pages and index nodes are interleaved in the log. On top of the figure is the logical log, which is a sequence of pages. Pages denoted PiVj are data pages, where i is the page number, and j is the version number or timestamp of the page. Pages denoted IDXi are index pages. The index pages will in general be part of an index tree, but to keep this example simple, we assume the number of pages is low enough to be able to store all index entries on one page.

At time t_0, a transaction allocates four pages, which are written to the log. After the transaction commits, the index node IDX0 is written, so that pages can later be accessed via this index. Later, at time t_1, a new transaction modifies page number 2 (whose first version was denoted P2V0). The new version of the page (page P2V1) and a new version of the index (index node IDX1) are written to the log.

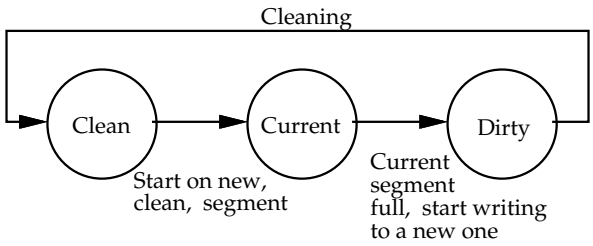


Figure 5.3: Segment states.

As can be seen from the figure, the previous versions of pages are still stored in the log, in addition to the current versions. The two versions of the database are illustrated in the figure, with arrows from the respective index nodes. Index node IDX0 indexes the database as of time t_0, and index node IDX1 indexes the database as of time t_1. As illustrated in this figure, only the current versions can be accessed from IDX1. If we want to be able to access old versions of data, we can use a multiversion index.

Note that even we in this example write data and commit sequentially, this is not necessary in practice. As will be described in more detail later in this thesis, data from different transactions and committing transactions can be interleaved.

5.1.2 Log Operations

A segment can be in one of three states, as illustrated in Figure 5.3. A segment starts in a *clean state*, i.e., it contains no data. The segment currently being written to, is called the *current* segment. When the segment is full, we start writing into a new segment. The new segment now goes from the *clean* state, to *current*. The previous segment is now *dirty*, it contains valid data (note that dirty in this context has nothing to do with main-memory state versus disk state, as the term is most frequently used). Information about the status of the segments is kept in the *segment status table* (SST), which is kept in main memory during normal operation.

If system load is low, or transactions are mostly read-only, only small amounts of new data will be created. In this case, update transactions in the commit phase, waiting for data to be written to disk, will experience long delays if we try to fill up the segments before we write. This is not acceptable, and can be solved by writing subsegments (also called partial segments). When writing subsegments, we write more than one logical segment into one physical segment. For example, if the physical segment size is 512 KB, we can instead write 4 subsegments of size 128 KB into the physical segment.

At regular times, a checkpoint operation is performed. In the checkpoint operation, we write enough information to the log to make the current position in the log a consistent starting point for recovery.

Recovery in a log-only database can be performed very fast, since there is no need to redo or undo any data. Only segments written after the last checkpoint need to be processed. At recovery time we simply do an analysis pass from the last known checkpoint to the end of the log, where the crash occurred.[1]

As data is updated or deleted, old segments can be reused. Updated an deleted data will leave

[1]However, as we shall see later, it can be beneficial to extend the amount of data that needs to be read at recovery time, to increase performance under normal operation.

behind a lot of partially filled segments, the data in these near empty segments can be collected and moved to the current segment, thus freeing up space in the old segments and making the old segments available for reuse. This process is called *cleaning*. For each segment, the SST contains a live byte counter. When data is deleted, this counter is decremented, so that we know which segments are good candidates for cleaning.

5.2 Advantages of a Log-Only Approach

Because the log-only, no-overwrite approach, is radically different from the techniques used in current systems, it is appropriate to describe the advantages of this approach.

Transparent Compression of Data. Objects are not written to the same physical location every time, and as a result, there is no need to reserve space for the maximum size of the compressed object. Even if compression ratio and the corresponding storage size change, no space is wasted.

Easier On-Line Backup. The written segments are time stamped, and with a no overwrite strategy, it is enough to know the last time of backup to know where backup should be started now. Backup could also be done on-line, and again, even if we stop backup when the load is high, we know where to continue in less busy periods.

Flash Memory. Very high performance can be achieved if we use fast non-volatile memory instead of disk. One example of such an storage technology is flash memory. Flash memory is byte readable, and fast, but write/erase has to be done blockwise. This suits a storage strategy with no in-data modifications.

Write-Once Memory. With write-once storage, for example optical disks, there is a need for a no-overwrite strategy.

RAID Technology. Disk access times and bandwidth improves at a much lower rate than main memory, and parallel disk systems are necessary to get high performance. To benefit from RAID technology, the write blocks have to be much larger than those used in traditional systems. In addition, in normal systems, sequential writes are only about 3-5 times faster than random writes, while in RAID, sequential writes can be up to 20 times faster than random writes [196]. One of the reasons for this difference is the writing of parity blocks, which is necessary in order to be able to do media recovery in the case of a disk failure.

High-Bandwidth Applications. In many supercomputing applications, and more recently also in OLAP applications, computations are done on large matrixes and arrays. To be able to do operations on these large structures, it is often necessary to break them into chunks which can be processed independently. It is necessary to retrieve and store these chunks efficiently. The same applies to storage of multimedia data, for example video. Until now, only file systems have been able to offer the desired performance. However, there is a demand for some of the services offered by database systems in these areas: access control, concurrency control, and recovery. However, performance close to file system performance is necessary for a DBMS to be applicable.

Group Commit. Group commit, in addition to giving us larger writes, also gives opportunity for more intelligent clustering of objects from different transactions.

Fast Crash Recovery. The log-only approach has similarities to shadow storage. Even though the use of shadow storage can result in performance problems, it also has a very nice and interesting feature: very fast crash recovery. By never updating in-place, recovery issues can be solved much easier.

Temporal Database Management Systems. Keeping old versions comes at little extra cost in a log-only DBMS. Given a log-only DBMS, realizing a temporal DBMS should not add much extra cost.

Cache Coherence. Versioning/timestamping can be exploited in cache coherence protocols in client-server environments, as is done in BOSS [119].

Nomadic Computing. Objects and segments are timestamped. This can also be utilized to maintain consistency in client databases that are off-line part of the time. If these at regular times are connected to a server, they can be made consistent by uploading changes since the last connect.

The advantages listed above indicate that the log-only approach is highly interesting, but it should not come as a surprise that these advantages do not come for free. The log-only approach has similarities to other no-overwrite strategies, for example the shadow page approach, and it also inherits the nasty side of shadowing: after a while, data becomes unclustered. However, for several reasons, we expect that this will be less of a problem with our approach:

- The increased amount of main memory can to a large degree compensate for the lack of clustering.
- The access pattern is supposed to be more direct and navigational in an ODBMS than in a RDBMS.
- Some systems are mainly write-once systems (for example many SSDBs), and if a large batch is loaded at a time, we can get very efficient clustering.

It is also possible to *recluster* the database when needed, although this can be costly with large amounts of data. This can be done as a part of the cleaning process, which is performed asynchronously. While this at first glance might look as if we have to do twice the work to get the same result, compared to other systems, it is not necessarily so. If you use write-ahead logging (WAL), you also have to write the data twice, to the log as well as to the database itself. It is also important to remember that reclustering is also necessary in traditional DBMSs. Traditional clustering works well as long as the access pattern is static, but if the access pattern changes, the database have to be reclustered.

The cleaning process can also be utilized to do dynamic and adaptive clustering. With the advent of persistent programming languages that need efficient support for garbage collection, for example persistent Java, it is possible that the garbage collection can be done as a part of the cleaning process. In this way, the effective cost of the cleaning can be reduced.

In the case of a temporal DBMS, a kind of continuous reclustering is also needed in traditional systems. If you want to keep previous versions of data, and still want to keep the current set clustered, you have to move the old version before inserting the new.

5.3 Alternative Realizations

There are two alternative ways to realize a log-only ODBMS: *page-based*, and *object-based*. The most important difference between these two is how objects are indexed. In an object-based design, indexing is done at object granularity, with logical object identifiers, while in a page-based design, only the pages are indexed, and the page an object resides on, is hardcoded into its object identifier.

5.3.1 Page-Based Designs

In page-based designs, the log is seen as one large persistent address space. When an object is created, it is allocated space from this address space. The pages are written to the log, similar to the example in Figure 5.2. The objects are referenced by a persistent-memory address (a page identifier is included in the OID), and are retrieved via the page index which is interleaved in the log. If an object is modified, a new version of the page(s) it resides on is written back to the log.

The main advantage of the page-based approach is ease of implementation. However, it has some of the same problems as traditional page servers:

- Even if only a small part of the page is modified, the whole page has to be written back. If objects are not well clustered, this will give low effective write bandwidth.
- With bad clustering, main-memory buffer utilization will be bad as well.
- Reclustering is difficult, the indexing in a page-based design is similar to the use of physical object identifiers in a traditional system, even though the location of the pages in a log-only system changes, an object is bound to one page during its lifetime.
- Variable sized objects are difficult to integrate into the page approach, since the space is allocated when the objects are created. This makes it difficult to employ compression.

In addition to these well known problems from traditional page servers, a page-based log-only ODBMS also makes transaction management difficult. To avoid page level locking, you essentially need to have 1) an additional log to keep track of page updates, or 2) use ad-hoc techniques to solve the problem. Both solutions are likely to hurt performance and increase complexity.

Two page-based ODBMSs are Texas [189] and Grasshopper [91]. Based on the documentation of the commercial ODBMS MATISSE [134], it is also possible that MATISSE is page-based, but this has not been possible to verify.

5.3.2 Object-Based Designs

The alternative to a page-based design, is to index objects directly. In this case, only the object (or a delta object) needs to be written to the log when an object is modified. This is especially useful if good clustering is difficult. Dynamic clustering can be employed to give a good clustering. This is possible because clustering can change with time, according to the access pattern. It is also possible to get all of the other benefits of log-only systems, as described previously in this chapter.

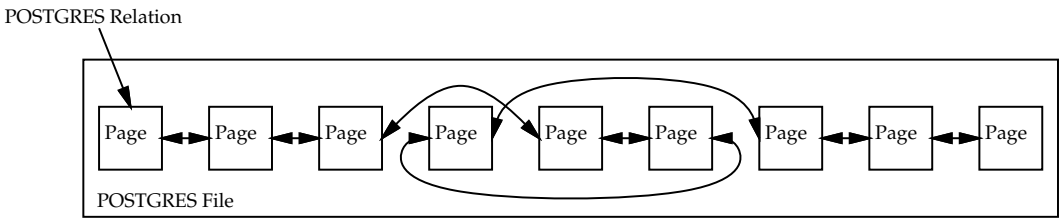


Figure 5.4: POSTGRES file.

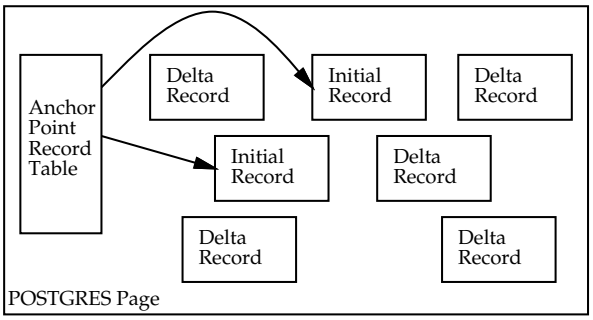


Figure 5.5: POSTGRES page.

The most important disadvantage with the object-based approach, is that more index updates might be needed, because individual objects are indexed, and not whole pages. We will later in this thesis develop techniques to minimize this disadvantage.

Based on the advantages and possibilities of an object-based design, we see it as the most interesting approach, and Vagabond, which will be described in detail in the rest of this thesis, is object-based. In the rest of this thesis, we write *log-only* as short for a log-only object-based design.

5.4 Systems Based on Log-Only Related Techniques

We will now present systems that employs log-only related techniques: POSTGRES, log-structured file systems (LFS), DBMS based on LFS, and the log-structured history data access method (LHAM). We also present other work, based on the ideas presented in these sections.

5.4.1 POSTGRES

No-overwrite strategies have a long history, for example in shadow-paging recovery schemes, like the one used in System R [5]. The best known log-only DBMS, and probably the first as well, was POSTGRES [195]. POSTGRES was an *extended relational database system*, which actually can be said to employ a no-log strategy rather than log-only.

Data in POSTGRES were stored in relations, which were stored in files. Pages were allocated or deallocated for a file on demand, and were linked together, as illustrated in Figure 5.4.

As illustrated in Figure 5.5, each page in POSTGRES had an *anchor table*, used to retrieve records stored on a page. When a record was created, space was allocated for the record. When records were

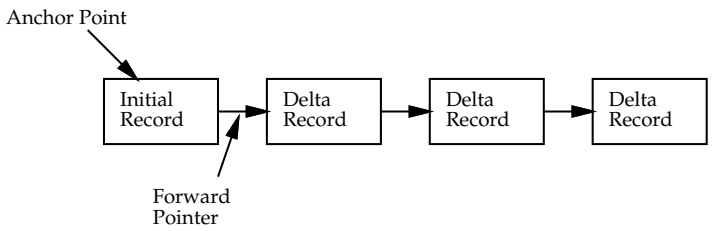


Figure 5.6: POSTGRES record.

updated, they were not updated in-place on the page, rather, a delta record was created, which recorded the changes from the previous version (illustrated in Figure 5.6). When a record was to be read, the whole chain from the first record had to be traversed and processed. POSTGRES was optimized for small records,[2] and delta records should be on the same page as the initial record.

Although POSTGRES introduced many novel ideas, the storage strategies did not gain much success at that time. The main reasons for this, were some serious problems resulting from the way records were stored:

- Read operations could be very expensive, because of the delta chains.
- POSTGRES used a *force buffer* policy. At commit, all data modified by the transaction had to be written, giving a very high commit cost.
- Even though POSTGRES could be used as a basis for a temporal DBMS, the use of append-only linked lists for each record was too inflexible and inefficient, an additional index was needed in most cases, increasing the overhead.
- In common with other no-overwrite strategies, POSTGRES also held the risk of declustering relations.

5.4.2 Log-Structured File Systems

The no-overwrite idea was borrowed from POSTGRES and used in log-structured file systems (LFS), first presented by Rosenblum and Ousterhout [177] in the Sprite LFS, and later refined by Seltzer et al. in BSD-LFS [185]. LFS has also been the basis for other systems, for example Spiralog [102, 212].

In an LFS, file and directory information is interleaved in a log. File identifying information is kept in inodes, similarly to Unix, and an inode map is used to locate the position of an inode in the log. It is assumed that the active portions of the inode maps can be kept in main memory. The granularity of writing (and indexing) in LFS is pages.

LFS has also been shown to be able to benefit from the advantages listed earlier in this chapter. It has been shown to be very well suited for tertiary storage management [69, 117], in on-line backup systems [78], and on-line data compression [36]. Implementing transactional support in LFS is described in several papers by Seltzer et.al. [182, 184].

The most important bottleneck in an LFS is cleaning, especially under heavy load, or when there is little free space on disk. This has been studied in several LFS performance improvement studies [19, 144]. This work has also resulted in improving the cleaning techniques and cleaning heuristics, and on self-tuning.

[2] A large object interface was added later.

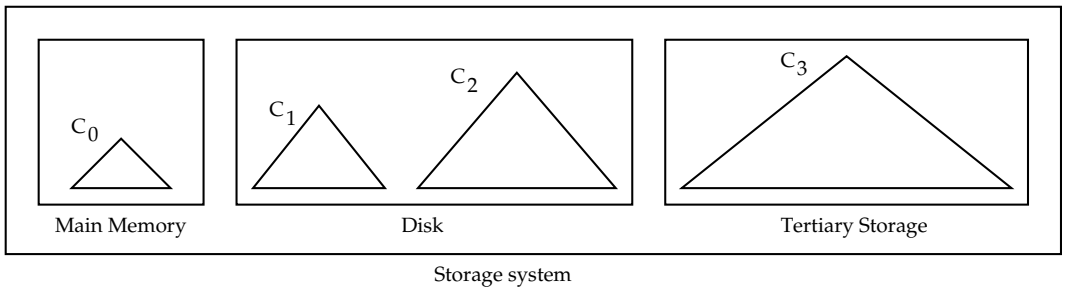


Figure 5.7: LSM with four components.

5.4.3 Log-Structured History Data Access Method

The *log-structured history data access method* (LHAM) [143, 170] is based on the *log-structured merge-tree* (LSM-Tree) [169].

A LSM is based on a hierarchy of index components, where the index component at each level has a larger size than the index component at the previous level in the hierarchy. We denote an index component C_i. Component C_i indexes a subset of component C_{i+1}, i.e., component C_i is (much) smaller than C_{i+1}. Component C_0 is in main memory,[3] $C_1 \ldots C_i$ are typically on disk and tertiary storage (see Figure 5.7).

Updates are only done to component C_0. Entries from C_0 not yet migrated to component C_1 are merged into component C_1 in batch, as a background process. In general, the same is the case for all the components in the hierarchy, entries from C_i not yet migrated to component C_{i+1} are merged into component C_{i+1} as a background process.

The most frequently updated entries will typically be in the lower levels of the hierarchy, because each update of an entry will result in an insertion into C_0.

As a result of the way updates are merged into the higher numbered trees, entries in component C_i are always as least as recent as entries in component C_{i+1}. When we search for an entry, we start the search in component C_0. If we do not find the searched entry there, we continue with component C_1, component C_2 and so on, until we find the entry, or have reached the last component. As long as we do the search as described, we are also sure to get the most recent version, even though the higher numbered components might contain outdated values.

Inserts and updates are only done to the first level index, and the contents of one level in the index are asynchronously migrated to the next level. As a result, all data inserted or modified during a certain time period will be in the same level. Search for data written at a certain time is efficient, but searching for the most recent version of certain data can be costly.

The main advantage of LHAM is support for high insertion rates, while also being competitive in terms of performance.

5.4.4 Other Related Work

Most no-undo/no-redo recovery approaches share some of the characteristics of the log-only approach. We have already mentioned the shadow-paging algorithm, used in System R. Other techniques are *differential files* (also called deferred update and side files), and the *database cache*. In the differential file approach, updates are done to a new and previously unused location, in a side file [77]. At regular

[3] Logging has to be used to be able to recover from main memory failure.

intervals, the contents from the side file are copied back to the original file. The database cache [63] uses a similar approach, but by assuming large main memory buffers, new algorithms can be used to avoid some of the problems of the shadow page and side file approaches.

Other relevant work includes approaches to deferred update, for example the BOSS approach [119]. BOSS employs WAL, but updates to the database itself can be deferred. The difference between deferred updates as used in BOSS and a log-only approach is that the log-only approach takes it to the extreme, there is *only* the log. The advantage with the log-only approach is that the updates to the database are avoided.

5.5 Summary

This chapter outlined the basic principles behind a log-only ODBMS based on LFS techniques. As described in the overview of other systems based on log-only techniques, the log-only approach is not new in itself, and even log-only systems can be said to be based on earlier techniques, for example shadow-paging.

The summary of advantages of a log-only systems should serve as a motivation to explore this strategy further, but it should be emphasized that whether a system might be able to achieve high performance has to be verified by analytical modeling or simulations. In Chapter 14 we will use analytical modeling to study the possible gain from using a log-only approach. However, real evidence can only be gained from an implementation of the approach, used in real applications. This is left as further work.

Part II

The Design of Vagabond

Chapter 6

An Overview of Vagabond

In this section, we briefly describe the architecture of the Vagabond temporal ODBMS, which is used as the context for the following chapters. The server architecture is described, and we give a summary of techniques that can be used to reduce the read costs in such a system. These techniques will be described in more detail in the rest of this thesis. We describe storage objects, and we describe how Vagabond can be incorporated into a parallel and distributed architecture. We emphasize that this is not the description of an implemented system, only a framework for the design presented in the rest of this thesis.

6.1 Server Architecture

Similar to the Shore ODBMS [39], we also use a peer-to-peer architecture. All application programs in the system are connected to one server, running on the same machine as the application. This server is the gateway to the DBMS, including remote servers (cf. the multiple client/multiple server architecture in Section 3.4). Not all servers have data stored locally. If all servers did, including those running on office workstations, that would make it impossible to achieve high availability. However, even if no data is stored locally, a server must be running on that node to make it possible for the application program to access the DBMS. One advantage of this approach is that it makes it possible for several clients running on the same machine to utilize a common server-side cache. On the client, client-side caching will be employed as well.

6.1.1 Client/Server Communication

A Vagabond server is an *object server* (see Section 3.6.1). The architecture of the server is illustrated in Figure 6.1. A client normally operates against the Vagabond API, a client-side stub which provides the mechanisms to communicate with the server. The communication with the server is done via the *messenger*.

6.1.2 Server-Side Operation

The server is multithreaded, and all subservers run as separate threads. There is also one thread for each transaction. To reduce thread creation costs, we use recyclable threads. Recycling of threads is done by having a fixed (but expandable) number of threads of a particular type. These threads are created when the server is started. When idle, they are waiting in an *thread pool*. When a task is to be started, it can be allocated one of the idle threads. When the thread has finished its task, the thread

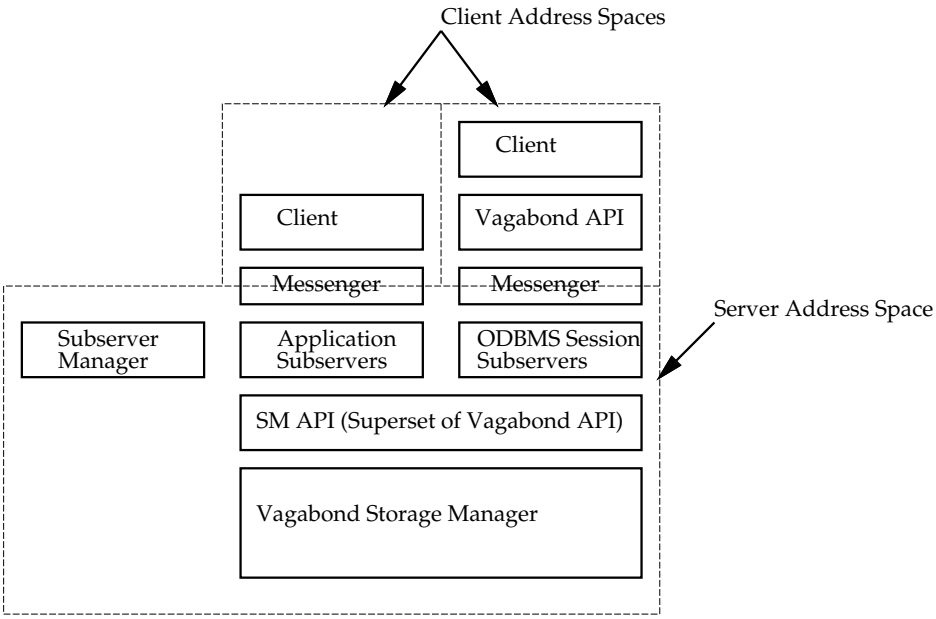


Figure 6.1: The Vagabond server.

is returned to the thread pool. This saves the overhead of creating threads, and the cleanup cost after thread termination. Even though the use of threads incurs extra cost compared with an event driven system, for example more locking overhead, thread-administration, and thread-switching overhead, multithreading makes it easier to exploit multiprocessor computers.

Each client that connects to the server starts the session by connecting to the *Subserver manager*, which allocates either an *ODBMS session subserver* thread or an *application subserver* thread to the client. The allocated thread operates in the server address space on behalf of the client. Commands and data are communicated through the *messenger*.

6.1.3 Subservers and Server Extensibility

All communication between clients and the storage manager is done via subservers. We have three classes of subservers:

- *Subserver manager*.
- *ODBMS session subservers*.
- *Application subserver*.

The subserver manager is only used when a session is started, to allocate the appropriate subserver, as described above.

ODBMS session subservers and application subservers access the storage manager (SM) through the SM API. Normally, ODBMS session subservers are used. Application subservers are extensions to the system, making it easy to extend Vagabond. This is similar to features found in other systems, for example the Value Added Server concept in Shore, and DataBlades/Cartridges in commercial systems. However, for such a concept to be really beneficial, the ODBMS has to be object shipping

rather than page shipping, so that the system is able to filter out objects and do operations on the objects, something which is impossible or difficult on page server systems.

One interesting point here, is that the SM API is a superset of the Vagabond API, the client interface stub. This feature makes it easier to implement and test subservers as clients, before they are added to the server. As subservers, they can communicate with clients through a messenger, as illustrated in Figure 6.1.

6.1.4 Storage Manager

The storage manager is responsible for permanent storage of objects. Its most important operations include transaction management, secondary and tertiary storage management, and indexing.

Buffering data in main memory is done to reduce the amount of data needed to be transfered between main memory and disk, and between individual servers. Important buffers include the object buffer, index node buffer, and object descriptor buffer (an object descriptor is an entry in the OIDX, see Section 3.1.2, and will be described in more detail in Section 8.1.2). These buffers can be dynamically resized, to get optimal performance with changing access patterns. The cost functions derived later in this thesis, and the papers in the appendixes, can aid in dynamically deciding optimal buffer sizes.

6.1.5 Permanent Storage

All data in a Vagabond server is stored in a logical log, which is stored in one logical *data volume*. A data volume consists of one or more storage devices. A storage device can be a secondary as well as a tertiary storage device. Typical examples of storage devices are:

- A raw disk partition (on magnetic disk).
- A fixed size (but extendible) file on the native file system simulating a disk partition. Running our own system on top of the native file system gives an extra level of indirection. However, allocating disk space on an (almost) empty disk will on most modern file systems give a mostly sequentially allocated file. This is done by creating a disk file, and writing as many blocks to it as the size specified. The file can be extended or shrunk by any integral number of segments.
- Optical disk.
- Tape.
- Flash memory.

Devices can be dynamically added to or removed from a data volume. Adding a device basically increases the number of available segments in the volume, while removing a device is done by first moving all data currently residing on that device to another device.

Even if disk space is cheap, it is still necessary for some applications to have data on tertiary storage. This can be done transparently in Vagabond. Tertiary storage is most often removable media, for example optical disk or tape, which can be used in disk and tape robots.

One of the devices in the data volume is called the *root device*. On this device, the most important volume information is stored, and it also contains the checkpoint block. The checkpoint blocks should be on a rewritable medium, so the root device will typically be a magnetic disk.

6.2 Objects in Vagabond

In Vagabond, all objects smaller than a certain threshold are written as one contiguous object. They are not segmented into pages as is done in other systems. Objects larger than this threshold are segmented into *subobjects*, and a *large-object index* is maintained for each of these large objects (this is done transparently for the user/application). There are several reasons for doing it this way:

- Writing one very large object should not block all other transactions during that time.
- A segmented object is useful later, when only parts of the object is to be read or modified.
- Parts of the object can reside on different physical devices, even on different levels in the storage hierarchy.

The value of the threshold can be set independently for different object classes. This is very useful, because different object classes can have different object retrieval characteristics. Typical examples are a video and an index. When playing a video, you want to retrieve one large segment of the video each time. On the other hand, when searching an index tree, you only want to retrieve single nodes, which usually have a small size. Similar for both video and index retrieval is that you only want a part of the object. For other objects, the whole object will be needed at once. One example is images. In order to be able to display the image, the whole object is needed. In that case, storing the image as one contiguous object will be advantageous.

Every object version in Vagabond has an associated object descriptor (OD), which contains the OID, physical location, timestamp, and other administrative information. In addition, every subobject has an associated subobject descriptor (SOD). ODs and SODs will be described in more detail in Section 8.1.2 and Section 8.5.2.

6.2.1 Typed Objects

It is not strictly necessary for the storage system to know the type of an object. Actually, in most systems, an object is simply a chunk of bytes from the storage manager's point of view, and page servers do not even have to know about objects, pages are all they care about. However, storing type information in the system can improve efficiency and performance considerably:

- It is useful for type checking.
- It makes it easier to employ hierarchical concurrency control techniques.
- If the server knows the attributes and attribute sizes of an object, it is easier to support vertical fragmentation in a parallel or distributed system. The alternative is that the application gives "partitioning hints", for example where in an untyped object it is possible to partition it. Some minimal information is also needed for reclustering and garbage collection (at least one needs to know where the pointers are).
- Typed objects are also necessary if the server shall be able to do some kind of server-side filtering or method execution.

In Vagabond, meta information for an object class or type is stored in the database as an object, called a *class descriptor object* (CDO). CDOs can be versioned as other objects, which simplifies support for class version management.

CID
Special object type
(Data field reserved for special object manager)
Maintain signatures?
Signature size
Large-object threshold
Subobject size
`NavigDesc` size
Vacuuming age
Metadata – Attribute information – Signature calculation information

Figure 6.2: Class descriptor (CDO).

Every object in the system belongs to a "class" (not only a programming language class, for example, it can also be an index class) as described in a CDO. A CDO is uniquely identified by a *class identifier* (CID), and the CID is used in object descriptors (ODs) to identify the class an object belongs to. The structure and contents of the CID are discussed further in Section 8.1.2.

The structure of a class descriptors object (CDO) is summarized in Figure 6.2. In the case of objects to be handled by special object handlers (see Section 6.2.4), the *special object type* identifies which special object handler should be used. The *reserved field* is reserved for the special object handler, and can be used to identify index variants, for example different key types in the case of an index. The `NavigDesc` *size* is the size of the *navigational descriptor*. A navigational descriptor exists in some index objects, for example B-trees, where it is a `(key,pointer)` tuple. The use of the `NavigDesc` will be further explained in Section 8.5.2 and Chapter 10.

The *large-object threshold* can be set to different values for different object classes, making subobject partitioning very flexible. The *subobject size* is the size of the subobjects in a large object (this can be different for different classes).

The *vacuuming age* is used for lazy vacuuming (see Section 12.6). If the timestamp of the object is older than the vacuuming age, the object can be removed. The default value of this attribute is a null value, i.e., the objects in this class can not be vacuumed.

The CDOs can hold other associated meta information as well, typically attribute and value offset information. This is also the place to store which attributes should be used to create the object signature if that is enabled for the particular class (hash-based signatures can be used to reduce the number of objects that need to be retrieved from disk, this will be described in more detail in Section 7.1). Whether to maintain signatures or not, is defined by the *maintain signatures* field. The size of the signature is stored in *signature size*. Note that using a *signature size* of zero to imply that no signature should be maintained, in stead of using a *maintain signatures* field, is not possible. The reason is that we want to make it possible to later disable signature maintenance for a class, without changing the size of the ODs, which include the signature. If this functionality was implemented by setting the signature size to zero, we would have to reorganize the relevant part of the OIDX to reflect the changed size of the ODs.

The CDOs are stored in the log as objects. When a new class is created, the class descriptor object is stored on all participating servers (full replication). The number of classes is in general small, so

the space used for this information will not represent a problem. Additionally, the information in a CDO will be frequently used, and it is therefore beneficial to have this resident at all servers. Creation of classes is an infrequent operation, compared to object creations, and the replication of CDOs will not represent a performance problem. It is not likely that an application needs real time response to class creations.

6.2.2 Temporal Aspects

Our storage structure is intended to be suited as a basis for a temporal DBMS. We maintain the temporal information in the index, which makes retrieval efficient even without additional temporal indexes.

6.2.3 Isochronous Retrieval

Some applications, for example video servers, do not want all of the objects delivered at once. Rather, they want part of it delivered at an appropriate rate, *isochronous retrieval*. One possible strategy to solve this is two queues in the I/O system. One for "normal" data, and one for high-priority audio/video data.

6.2.4 Special Objects

A large object can be viewed as an array of bytes, and retrieval of part of the object is done by retrieving a certain byte range of the object. This is not flexible enough for some of the structures that are stored as large objects, for example indexes. These structures are stored as large objects, but the subobject index has additional information to support more complex indexes. They can also have different concurrency control and recovery characteristics. These objects, which we call *special objects*, are handled by *special object handlers*, which will be treated in more detail in Chapter 10.

6.2.5 Examples of Special Objects

Class descriptor objects, persistent roots, collections, index structures and spatial data structures are also stored as ordinary data objects. This has the advantage of making them an integral part of the object system.

Collections

A collection is a collection of objects, for examples a set, bag, array or list,[1] with supporting methods for inserting, removing, and testing for the existence of a certain element. It also supports the use of an iterator to access the elements of a collection.

Secondary Indexes

To make an ODBMS efficient, we need secondary indexes in addition to the OIDX. One-dimensional as well as multi-dimensional indexes, which can be suitable for temporal queries as well as for spatial data, should be supported.

In Vagabond, indexes are also supposed to be stored as objects. An index will often be a large object. In many systems, all indexes have the same index node (block) size. In Vagabond, this can

[1] Collections are in some literature also called *containers.*

be tailored, so that different indexes, for different object classes, can have different index node sizes, depending on expected and actual access patterns.

To be able to use this system as a basis for a GIS, it is necessary to have support for spatial data structures. Only very recently have commercial DBMS with spatial support emerged, some with the data structure implemented in BLOBs, other with more integrated extensions, such as Informix Universal Server's geodetic DataBlade module, DB2's Spatial Extender, and Oracle 8i/Spatial Cartridge.

Most ODBMS vendors do not have the infrastructure or the architecture necessary to support scalable spatial data management, and a client-side index solution is necessary. One example of this is ObjectStore.

Persistent Roots

To be able to access the objects later, we need some handles into the database. This is typically done by the use of *persistent roots*. A persistent root is a *named object*, i.e., a tuple consisting of a name (a string), and an OID. The persistent roots are stored in a persistent root object, which is an object with a predefined OID. The persistent root object itself is an index.

Multidimensional Arrays

The storage scheme we have described here is particularly applicable to arrays, which are heavily used in scientific computing. Subarrays are stored as contiguous chunks in the segments, which will give very good performance, even for read-only transactions.

6.3 Read and Write Efficiency Issues

The system is write-optimized, and as a result, object retrieval and index lookup can become a serious bottleneck. We employ some techniques to reduce the number and size of read operations. These techniques can improve performance considerably, with none or marginal write penalties:

- Careful layout of objects.
- Hash-based signatures.
- Clustered index.
- Object compression.

The last one, object compression, will also improve write efficiency, as it reduces the amount of data that needs to be written to disk. In addition to the techniques listed above, we also employ writing of delta objects to further reduce the amount of data needed to be written to disk (this technique reduces the write cost, but increases read cost), and a number of index optimizations as will be discussed in Chapter 8 and Chapter 9.

6.3.1 Careful Layout of Objects

Several strategies are used to store objects on disk in a way that reduces read cost. One important strategy is to try to store related objects close to each other when objects are stored on disk. Because we employ a no-overwrite strategy, heuristics can be used to reorder objects in segments that are to be written to disk, and during cleaning of segments.

Another possible strategy is to use heuristics to arrange objects in a segment so that they do not span more disk blocks than necessary (boundary alignment and reordering). In this way, a minimal number of disk blocks needs to be read when data is retrieved from disk.

6.3.2 Signatures

We employ a technique similar to signature files to reduce the number of objects that needs to be retrieved. This can be done with a very small extra cost in Vagabond. This is described in detail in Section 7.1.

6.3.3 Clustered Index

The OIDX is organized in a way that clusters OIDX entries for object belonging to a physical container (a collection of related objects, to be described in more detail in Section 8.1.1). A physical container can for example be used to store all objects from a class (and implicitly maintaining class extents), or all objects in a collection. This can makes set-based queries more efficient. If using signatures as well, the actual number of retrieved objects can in many queries be very low, as we in this case only have to scan the relevant part of the index and the objects with matching signatures.

6.3.4 Object Compression

To further reduce storage space, and disk bandwidth, objects can be compressed before they are written. This is described in Section 7.2. With a log-only approach, objects are written to a new location every time, so that we only use as much space as the size of the current version that is written.

6.3.5 Delta Objects

Often, only a small part of an object is changed when a new version is created. In this case, much can be gained if only the changes are written. This is especially the case if an object is a write hot spot object. An object that only contains the changes from the last version of the object, is called a *delta object*. Unlike traditional systems, that only use delta objects to reduce the log writing, a delta object in a log-only database system can be an object version on its own, i.e., the complete version will not necessarily be written.

The delta object itself can be made at a low cost, for example by using the following algorithm:

1. Do a bitwise XOR on the new and the old version of the object. The resulting bit string will now have 1's only in positions where there is difference between the old and new version.

2. The resulting bit string can be run-length encoded, and the resulting delta object will be very small if there has been only small changes between the two versions.

This algorithm is most beneficial when the objects have the same object length and fixed size attributes. The algorithm can easily be extended and improved, for example by only considering updated attributes.

In general, generating and writing a delta object is only relevant if the previous version is already in memory. This is usually the case. If not, an inefficient installation read of the previous version would be needed to be able to generate the delta object.

It is not always beneficial to write delta objects in the case of large objects. Many large objects are relatively static objects, and when updates are done, large parts of the affected subobjects (a large

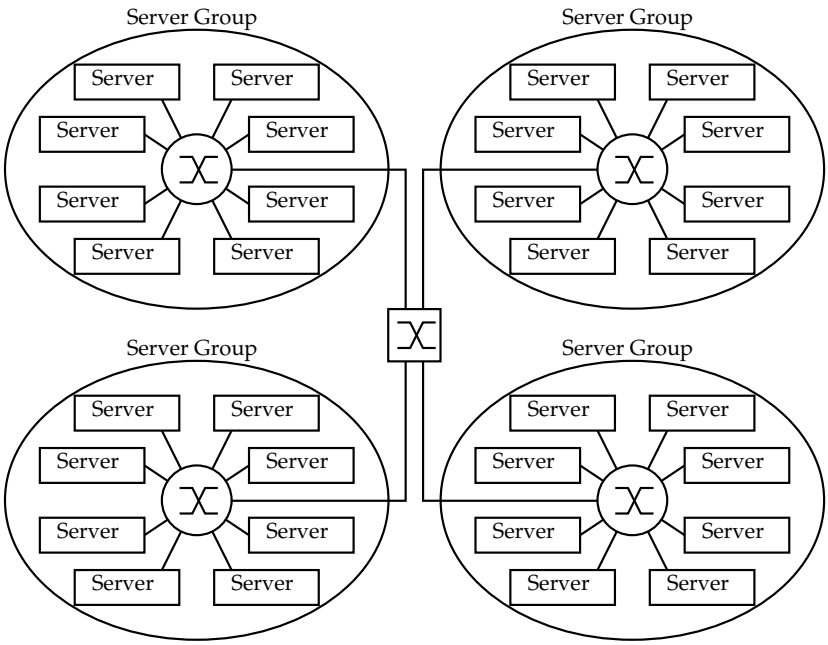


Figure 6.3: Vagabond system architecture.

object is physically partitioned into subobjects) are modified. Other large objects, for example index structures, are usually more dynamic, and updates only affect a small part of one subobject. For such objects, writing delta objects is beneficial.

The disadvantage of writing only a delta object is, of course, that previous versions have to be retrieved to reconstruct an object at read time. This problem can be reduced by writing the complete version of an object when an object is to be replaced in the buffer due to buffer replacement policy (this is done in addition to writing delta objects, i.e., if an object has been updated several times before it is discarded from the buffer, several delta objects might have been written). In this way, reading can be done efficiently later (note that because of the log-only strategy, this writing is relatively cheap). Reading the chain of objects is only needed if the DBMS has crashed before a non-delta object has been written. This strategy is most beneficial for non-temporal objects. In the case of temporal objects, we also need to reconstruct previous versions, and not only the most recent version. The described strategy does not solve that problem, but the problem can be reduced by writing complete versions at regular times, for example a full object version for every n'th delta version.

6.4 Parallelism and Distribution in Vagabond

The Vagabond architecture is a system designed for high performance, and one strategy to achieve this, is to base the design on the use of parallel servers. Objects are declustered over a set of servers, which we call a *server group*. The declustering is done according to some declustering strategy, this is further discussed in Section 11. The servers in a server group can cooperate on the same task, and in this way, it is possible to get a data bandwidth close to the aggregate bandwidth of the cooperating servers. To benefit from the use of a parallel server groups, it is supposed that the servers in one server group are connected by some kind of high speed communication.

The demand for support of *distributed databases* is increasing, and to satisfy this, we use a hybrid solution: *a distributed system, with server groups* (Figure 6.3). In this way, objects are clustered on server groups based on locality as is common in traditional distributed ODBMSs, but one server group can contain more than one computer (a kind of "super server").

6.5 Summary

This chapter described the overall architecture of the Vagabond ODBMS, and the main features. The rest of this thesis will concentrate on *how* to make a system that can deliver support for these features. We will identify potential bottlenecks, and describe how the impact of these bottlenecks can be reduced.

Chapter 7

Reducing the Data Transfer Volume

In this chapter we describe two techniques that can be used to reduce the amount of data transfer between memory and disk as well as between different servers: signatures and data compression. The techniques are well-known, but previously, only limited success has been achieved from using these techniques. However, some of the factors that previously have reduced their practical application are not present in a log-only system, making the gain from using these techniques larger in a log-only system than in a system based on in-place updating.

7.1 Signatures

A signature[1] is a bit string, which is generated by applying some hash function on some or all of the attributes of an object. By applying this hash function, we get a signature of F bits, with $m \leq F$ bits set to 1. If we denote the attributes of an object as $A_1, A_2, \ldots, A_n$, the signature of object i is:

$$s_i = S_h(A_j, \ldots, A_k)$$

where S_h is a hash value generating function, and $A_j, \ldots A_k$ are some or all of the attributes of the object (not necessarily including *all* of $A_j, \ldots, A_k$, S_h does not necessarily use all its arguments). Similar to hashing in general, two objects with the same signature may or may not have the same (shallow) value, *but objects with different signatures are guaranteed to differ*. The size of the signature is usually much smaller than the object itself, and it has traditionally been stored separately from the object, in a signature file.

When searching for objects that match a particular value, it is possible to decide from the signature of an object whether the object is a possible match. By first checking the signatures when doing a *perfect-match query*, the number of objects that has to be retrieved can be reduced. This can considerably reduce the total retrieval cost, because the size of the signature file is smaller than the total size of the objects involved in the query.

A typical example of the use of signatures is a query Q to find all objects in a set where the attributes match a certain number of values:

$$A_j = v_j, \ldots, A_k = v_k$$

This can be done by calculating the query signature s_q of the query:

[1] Note that the term *signature* is also used in other contexts, e.g., function signatures and implementation signatures.

$$s_q = S_h(A_j = v_j, \ldots, A_k = v_k)$$

The query signature s_q is then compared to all the signatures s_i in the signature file in order to find possible matching objects. A possible matching object, a *drop*, is an object whose signature s_i is equal to s_q (in the case of signatures generated by superimposition, which will be discussed below, a drop is a signature where all bit positions set to 1 in the query signature are set to 1 in the object's signature). The drops form a set of candidate objects. An object can have a matching signature even if it does not match the values searched for, so all candidate objects have to be retrieved and matched against the value set that is searched for. The candidate objects that do not match, i.e., objects with the same signature as the query signature, but not matching the query, are called *false drops*.

Signature files have previously been shown to be an alternative to indexing, especially in the context of text retrieval [15, 66]. They can also be used in general query processing, although this is still an immature research area. The main drawback of signature files, is that signature file maintenance can be relatively costly; every time the contents of an object change, the signature file has to be updated as well. To be beneficial, a high read to write ratio is necessary. In addition, high selectivity is needed at query time to make it beneficial to read the signature file in addition to the candidate objects.

We will now describe in more detail how signatures are generated, signature storage alternatives, and how signatures can be used in an ODBMS without requiring a high read to write ratio.

7.1.1 Signature Generation

The methods used for generating the signature depends on the intended use of the signature. We will now discuss some relevant methods.

Whole Object Signature

In this case, we generate a hash value from the whole object. This value can later be used in a perfect-match search that includes all attributes of the object. This method is only useful for a limited set of queries, where all the attributes of the object are involved in the perfect-match search.

One/Multi-Attribute Signatures

A more useful method is to computer the hash value of only one attribute of the object. This can be used for perfect-match search on a specific attribute. Often, a query is on perfect match of a subset of the attributes, similar to the example above. If such queries are expected to be frequent, it is possible to generate the signature from these attributes, again only looking at the subset of attributes as a sequence of bits. This method can be used as a filtering technique in more complex queries, where the results from this filtering can be applied to the rest of the query predicate.

The one/multi-attribute signature method is not very flexible, as it can only be used for queries on the exact set of attributes used to generate the signature. In the case of small sized attributes in a traditional system, an index would in general be more suitable. Its search performance will be better, and it supports range queries. In the case of large attributes, it is possible to use the signature instead of the whole attribute in the index. Using one/multi-attribute signatures when these signatures can be embedded into the OIDX, can still prove to be beneficial (see Appendix F).

Superimposed Coding Methods

The real power of signatures comes when the *superimposed coding* technique is used. When this techniques is employed, we get a signature that can be used for different perfect-match queries, where the different queries involve different sets of attributes.

When superimposed coding is used, we first compute a separate attribute signature $S_h(A_i)$ for each attribute in the object. The object signature itself is generated by performing a bitwise OR on each attribute signature. For example, for an object with 3 attributes, the object signature is calculated as:

$$s_i = S_h(A_0) \text{ OR } S_h(A_1) \text{ OR } S_h(A_2)$$

This results in a signature that can be very flexible in use, we can do a perfect-match search on any subset of attributes. When comparing a search signature with object signatures generated by superimposed coding, an object is a drop if all bit positions set to 1 in the query signature are set to 1 in the object's signature. It is also possible that other bit positions in the object's signature are set to 1, but that is not relevant for the actual query. The other bits set to 1 have been set as a result from attributes not part of the query.

Superimposed coding can also be used on set-valued attributes (a set-valued attribute is an attribute that itself is a set). In this case, a signature is generated for each member of the set. These signatures are OR-ed together to generated the attribute signature [96, 114]. By using this technique, queries of the type *is-subset*, *has-subset*, *has-intersection* and *is-equal*, can be answered efficiently, in many cases with less cost than alternative methods, for example using nested indexes.[2]

7.1.2 Signature Storage

Traditionally, the signatures have been stored in separate files, outside the indexes and objects themselves. A signature file contains the signatures s_i for all objects i in the relevant set. The size of a signature file is in general much smaller than the size of the relation/set of objects that the signatures were generated from, and a scan of the signature file is much less costly than a scan of the whole relation/set of objects. The most well-know storage structures for signatures are *Sequential Signature Files* (SSF) and *Bit-Sliced Signature Files* (BSSF), which are most suitable for relatively static data [66]. To better support inserts, deletes, and updates, several dynamic signature file methods have been proposed, based on multi-way trees and hash files.

Sequential Signature Files

In the simplest signature-file organization, SSF, the signatures are stored sequentially in a file. A separate *pointer file* is used to provide the mapping between the signatures and the objects. In an ODBMS, this pointer file will typically be a file with OIDs, one for each signature. During each search for perfect match, the whole signature file has to be read. When an object is updated, one entry in the signature file needs to be updated.

Bit-Sliced Signature Files

With BSSF, each bit of the signature is stored in a separate file, so that with a signature size F, the signatures are distributed over F files, instead of one file as in the SSF approach. This is especially

[2] A nested index is a B-tree variant where the leaf node entries are composed of a key value and the OIDs of the objects that have this key value in the indexed attribute [14].

useful if we have large signatures. In this case, we only have to search the files corresponding to the bit fields where the query signature has a "1". This can reduce the search time considerably. However, each update implies updating up to F files, which is expensive. So, even if retrieval cost has been shown to be much smaller for BSSF, the update cost is much higher. Thus, BSSF based approaches are most appropriate for relatively static data.

Several improvements of the BSSF have been proposed, most of them imply some vertical or horizontal decomposition [87, 113, 172]. Variants that use signature compression and multi-level signatures also exist.

7.1.3 Signatures for Fast Text Access.

Fast text access has been the main application of signatures, and most of the publications on signatures have been related to text access methods [15, 65, 66, 120, 206, 217]. In this case, the signature is used to avoid full text scanning of each document, for example in a search for certain words occurring in a particular document.

Documents are first divided into logical blocks, which are pieces of text that contain a constant number of distinct words (if most documents are small and have approximately the same size, this step is not strictly necessary). A separate signature is generated for each of these logical blocks, i.e, there is in general more than one signature for each document. In order to generate a block signature, a word signature is generated for each word in the block, and the block signature is generated by OR'ing these word signatures.

When searching for documents containing one or more particular words, the signature file is read first, and each block signature is compared with the query signature (the signature generated from the query words). This gives us a set of candidate documents (or candidate blocks), where the actual search words might occur. These documents have to be retrieved and searched.

Example

To illustrate the advantage of using signatures for fast text access, consider a collection of 1024 technical documents. The average document has a size of 64 KB, and contains 600 distinct words. Without signatures (or an index), all documents have to be retrieved if we want to find which documents contain one or more specific words. The total data volume to read will be $1024 * 64$ KB$= 64$ MB.

The data volume to be read can be reduced if signatures are employed. In this example, assume a signature size of $F = 4096$ bits, and that these are stored in an SSF. We do not divide the documents into logical blocks. The size of the signature file will be $S = 1024\frac{F}{8} = 1024 * 512 = 512$ KB.[3]

When searching for documents containing one or several words, we first read the signature file. For all documents with matching signature, we have to read the document. The probability that a retrieved document does not contain the actual word, is equal to the false drop probability [66]:

$$F_d = (\frac{1}{2})^m, \text{where } m = \frac{F \ln 2}{D}$$

In the case of a text document, D is the number of distinct words in a block. In this example, only $F_d = 0.037$, i.e., 3.7% of the retrieved documents, will be false drops. Thus, instead of reading 64 MB, we can satisfy the query by only reading the signature file and the candidate documents. The number of documents to retrieve depends on the selectivity of the query. If we search for a very

[3] If a document identifier is included in the signature file, its size will be slightly larger than this.

common word, most of the documents have to be retrieved, but if we search for a combination of words, the number of documents to retrieve will in most cases be low.

How to find the optimal signature size is an issue when using signatures, and the size depends on several factors, including the number of candidate documents. A large signature reduces the false drop rate, but increases the size of the signature file, which has to be read in its entirety for all queries.

In the example above, we assumed that the signatures were stored in an SSF. Another alternative is to use BSSF. In this case, the signature files would occupy the same amount of disk space, but on average, only half of them had to be read to answer a query. This might at first seem like an advantage, but in practice, accessing extra files implies large overheads, so BSSF would not be beneficial with a small number of signatures as in this example.

7.1.4 Storing Signatures in the OIDX

In a write-optimized system, object retrieval can become a bottleneck. This bottleneck can be reduced by including the object's signature in the OIDX. In Vagabond, the OIDX is updated every time an object is modified, and if we store the signature in the object's object descriptor (OD) in the OIDX, the additional signature maintenance cost is only marginal. This is different from traditional systems, where the signature file has to be updated every time an object is updated, reducing its effect. In those systems, a large read to update ratio is necessary if the use of signature should be beneficial.

Perfect-match queries can use the signatures in the OIDX to reduce the number of objects that have to be retrieved, as only the candidate objects, with matching signature, have to be retrieved. When the signature is stored in the OD, scan queries can be done efficiently by simply doing a scan over the relevant part of the OIDX, and only the candidate objects need to be retrieved. Because the OD is accessed on every object access in any case, the additional signature-retrieval cost is only marginal.

Optimal signature size is very dependent of data and query types. In some cases, we can manage with a very small signature, in other cases, for example in the case of text documents, we want a much larger signature size. It is therefore desirable to be able to use different signature sizes for different object classes. In any case, we have a tradeoff between signature size and additional signature-maintenance cost. Even though a small signature has only marginal effect on OIDX access cost, using larger signatures will increase the cost to a significant level.

The maintenance of object signatures implies computational overhead, and is not always required or desired. Whether to maintain signatures or not can be decided on a per class basis. This is also the case with which attributes to use when calculating the signature. This information is stored separately for each class, in the class descriptor object (see Section 6.2.1). To avoid complex index node management, all ODs in a physical container have the same signature length.

A more detailed study of performance aspects of storing signatures in the OIDX is presented in the paper included in Appendix F.

7.1.5 Signature Caching

The signature could also be stored together with the object on disk. In this way, the additional update cost is small. This is obviously of no use if the signature is discarded from the buffer at the same time as the object is discarded. However, it is possible to store the signatures of frequently accessed objects in a *signature cache*. Because the signature size is small compared to the object size, reducing the number of objects that fit in the object buffer and instead using the memory for buffering signatures, can improve the performance.

The signature cache approach is particularly interesting for page server ODBMSs using physical OIDs, and in [156] we have shown that in such systems the average object access cost can be significantly reduced by the use of a signature cache. Signature caching can also be used in order to reduce the communication costs in a parallel ODBMS [154].

7.2 Object Compression

To reduce storage space, as well as the amount of disk bandwidth used, objects can be compressed before they are written to disk. Compression/decompression can be transparent to applications, which means that compressed objects are decompressed by the server before they are made available to the applications. In many application areas, for example SSDBs, it is typical to have objects (or tuples) with a very large number of attributes, of which many of them have null values. By compressing these objects, it is possible to reduce both storage space and read/write disk bandwidth. Another example application is text, which can usually be compressed down to less than 50% of the uncompressed size.

The idea of compression in databases is not new, and some work exists, especially in the context of SSDBs. A more general study and overview of support for compression of data in databases has been given by Iyer and Wilhite [98]. They also analyze different design options with different data sets.

In traditional systems, compression has been difficult to employ efficiently. The reason for this is that the effective compression ratio changes with the contents of the object, so that different versions of an object can have very different sizes after compression. As it is impossible to know the size of future versions of a compressed object, it is necessary to reserve as much space as the maximum size of a compressed object when updating in-place. When using a log-only approach, an object is written to a new location every time. In this way, a version only needs to occupy as much space as the size of the compressed version.

Better compression can be achieved if knowledge of the structure of the objects is available. One example of how easy this can done, is the use of a bit mask for each object, with one bit for every attribute of the object. A bit is set to zero if the attribute is null, if not, it is set to one. In this way, attributes with a null value need not to be stored at all. A variant of this technique, a descriptor containing the offset of the start of each non-null field, was used in System R [5] and POSTGRES [199].

Even without knowledge of the object structure, good results can be achieved. In this case, objects are simply treated as bit streams. To avoid using too much CPU resources, a low cost compression algorithm should be used, for example run-length encoding.

The fact that compressed data have to be decompressed before used, implies that queries accessing large amounts of data only to check for a match in one or more attributes can be costly. Without compression, such queries are usually I/O bound, but can easy become CPU bound if data is compressed. By combining signatures and compression, this problem can be reduced. When signatures are maintained, perfect-match queries on attributes in large sets of objects can be done efficiently even without decompressing the objects.

7.3 Summary

In this chapter we have described the use of signatures and data compression. In the past, these techniques have only had limited success in DBMSs. However, we expect that they can be more beneficial in a log-only system than in an system based on in-place updating, and in particular, when used together in such a system.

Chapter 8

Object-Identifier Indexing

In an ODBMS, an object is uniquely identified by an object identifier (OID), which is also used as a "key" when retrieving the object. As discussed in Section 3.1, OIDs can be physical or logical. In a log-only ODBMS, objects are never written back to the same place. This means that logical OIDs have to be used, and an OID index (OIDX) is necessary. The number of objects in a database can be very large, and a fast and efficient index structure is necessary to avoid OID indexing becoming a serious bottleneck. This chapter describes an object-index structure suitable for indexing OIDs in a temporal ODBMS, and provides algorithms for efficient access to the index.

8.1 Contents and Structure of the OID Index

The OIDX contains the necessary information to map from a logical OID to the physical location where the object is stored. The physical location, together with the timestamp and other administrative information, are stored in index entries which we call *object descriptors* (ODs). A new OD will be created for each new version of an object, so that for each object, there can be more than one OD in the OIDX, corresponding to the number of versions of the object.

In general, an ODBMS can manage multiple logical databases. The logical databases can be represented as one or several *physical* databases. In Vagabond, we use one physical database for each logical database, and each logical database has a separate index.

An OID is only unique inside one database, thus, object identifiers in different databases will represent different objects. All database sessions are performed against a certain database, which database to access is given implicitly, and it is not necessary to contain database identifying information in the OID.

As discussed in Section 3.1.2, the OIDX in a traditional system is usually realized as a hash file or as a multi-way tree structure. In a log-only system, a tree structure is the only reasonable alternative, since an index node will be rewritten to a new location every time it is updated. If a hash file was used, an additional tree index on top of it would be necessary. We would then effectively end up with a tree structure anyway. The same is the case if we wanted to use the direct mapping technique. For this reason, all index structures considered in this chapter are variants of multi-way trees.

We will in the rest of this section describe the structure and contents of the OIDs and ODs in Vagabond.

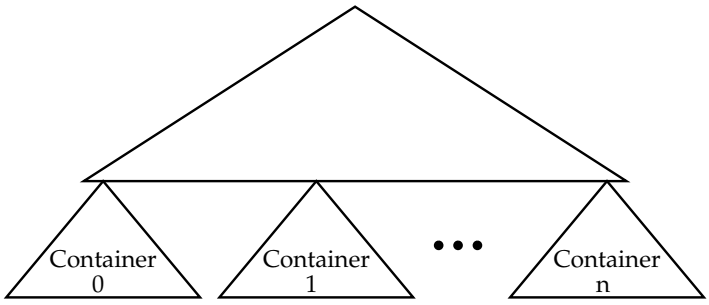


Figure 8.1: OIDX with containers.

8.1.1 Object Identifiers in Vagabond

In Vagabond, an OID is composed of three parts:

1. *SGID:* Server group identifier. The *SGID* is the identifier of the server group (see Section 6.4) where the object was created.

2. *CONTID:* Container identifier. The *CONTID* identifies the container the object belongs to (see below), similar to a file in other DBMSs.

3. *USN:* Unique serial number. Each object created on a particular server *SGID* and to be included in container *CONTID* gets a *USN* which is one larger than the previous *USN* allocated in this container.

The reasons for including a *SGID* and an *USN* in the OID should be obvious, but the rationale behind the *CONTID* needs some further elaboration.

In many page server ODBMSs, the objects are stored in containers, also called files, or relations (for example Objectivity, see Section 3.1). Which container to put an object in, is decided when the object is created, and this decision is often made according to some clustering strategy (see Section 3.3). In a traditional ODBMS, a part of the OID is often used to identify in which container the object is stored (see Section 3.1).

To benefit from the log-only approach, objects can not be stored in distinct files or clustered together in the same way as is beneficial in a system using in-place updates. This would make it difficult to achieve long, sequential writes. However, an approach similar to physical clustering of objects can be used for the OD in the OIDX. Similar to the way object clustering in page servers reduces the number of pages to read and update, clustering together ODs that are expected to be accessed together close in time, will reduce the cost of OIDX accesses. This is achieved by associating every object in a database with a container (see Figure 8.1). Which container an object belongs to, is encoded into its OID.

Note that the storage of objects is independent of which containers they belong to (for example, there is no relation between a segment and a container), the use of containers is only a way to cluster related ODs.

An object can be migrated from one container to another. If this is done, forwarding information is stored inside the OD representing the current version in the original container. When a migrated object is to be retrieved, two OIDX lookups are needed in order to retrieve the OD: one lookup in the original container, and one lookup in the container the object currently belongs to.

Field	Size (bits)
OID:	
SGID	32 (Only present when outside OIDX nodes)
CONTID	32 (Only present when outside OIDX nodes)
USN	64
Physical location	64
Object size	32
Create timestamp	64
End timestamp	64 (Only present when outside the OIDX)
Class identifier (CID)	24
Delta object?	1
Large object?	1
Temporal object?	1
Compressed object?	1
Inlined object?	1
First version?	1
Migrated to another server group?	1
Migrated to another container?	1
(Signature)	Optional field, variable size

Table 8.1: Contents and size of fields in the object descriptor.

In addition to using the containers as a way to cluster the ODs of objects that are expected to be accessed together, but in other ways are unrelated (i.e., different classes), they are also useful as a way to realize logical collections of objects from the same class, for example sets/relations, bags and class extents.[1] For example, one collection can be stored in one container. When this is done, scans and queries against these collections can then be executed efficiently. When using signatures as well, it will for many queries only be necessary to read a small proportion of the objects.

Another interesting use of containers is to have more flexibility in deciding the length of the search path for ODs. This can be achieved by storing hot-spot objects in small containers (i.e., containers with only a few objects) to get shorter search and update paths.

8.1.2 Object Descriptor Structure

The contents of an OD are summarized in Table 8.1, together with the size of the individual fields (Fields occupying one bit are used for boolean values).

The ODs are stored both in the OIDX and together with the objects in the segments. The reason for storing them in the segments as well, is to help identifying objects during cleaning, and it also works as a kind of write-ahead logging of ODs, in order to avoid synchronous updates of the OIDX at commit time.

The information in the OD gives a high degree of flexibility and efficiency, and even though it contains many fields, most of them can be stored in a compact way, many of them occupying only one bit. As will be described later in this chapter, the *SGID* and the *CONTID* are given implicitly

[1] A class extent is a collection of all the objects of a certain class in a database.

when stored in the OIDX, so they need not to be stored. In addition, the *USN*s of the ODs stored in one index node will be from a limited integer range, so that prefix compression can be used. When in main memory, and outside an index node, the *CONTID* must also be included in the OD. ODs for objects from other server groups are treated as a special case, so that the *SGID* is only used for the "remote" ODs. We will now describe in detail the contents and function of these fields.

Physical Location

This is the location of the object in the log. If the object is a large object, this location is actually the location of the root of the subobject index of the object (see Section 8.5.2). If the physical location is NULL, but the OD contains a valid timestamp, this OD is a tombstone OD (the object is deleted, but previous versions exist).

If an object is moved, for example during cleaning, the physical location in its OD has to be updated.

Object Size

All objects smaller than a certain threshold are written as one contiguous object, while objects larger than this threshold are segmented into subobjects, and a large object index is maintained for each of these large objects. The *object size* field in the OD is the size of a small object when in the log.

If the object is compressed, the actual size can be larger the *object size*. In the case of fixed-size objects, the size of the uncompressed object can be found from the class-descriptor object. If it is a variable-sized object, the size is stored together with the compression information in the compressed version of the object.

In the case of large objects, the object-size field is not used (if necessary, the object size can be found from the subobject index). Note that the fact that only 32 bits is used to store the object size only restricts the maximum size of a "small object", large objects can be larger than this.

Create Timestamp

This is the commit time of the transaction that created this version. Each transaction needs distinct timestamps, so a very fine timestamp granularity has to be used. A 64 bit timestamp is more than what is actually needed, but a 32 bit timestamp is not sufficient, and using a size between 32 and 64 bits is not efficient.

Timestamps with the most significant bit set are reserved for the case when a transaction identifier is used instead of the timestamp in the OD. This will be further explained in Chapter 12.

End Timestamp

The end timestamp is the time when the next version was created, or the time of deletion in case there are no new subsequent versions. The create and end timestamps give the interval an object was valid. If the OD is the OD of the current version of an object, the end timestamp is NOW, which is represented by the value NULL.

When in the OIDX, the end timestamp is given implicitly from the create timestamp in the OD of the next version of the object. This OD will in most cases reside on the same index node, so it is not necessary to store it here. However, when the OD is outside the OIDX,[2] the end timestamp is

[2]This also includes the PCache, to be presented in Chapter 9, where the ODs also include the end timestamp.

included. This makes certain operations and buffering of ODs more efficient (see Section 12.2.5).

Delta Object

Delta object is set to true if this is a delta object (see Section 6.3.5).

Large Object

This is true if this version of the object is a large object. An object can be a "plain data object" as well as a special objects (for example an index, as described in Section 6.2). If it is a special object, the relevant information is stored in the object's class descriptor (CDO).

Temporal Object

Temporal object is set to true if this is an object where we want to keep old versions when the object is modified or deleted. This is decided for each object at object creation time, but can be changed later (although this is not always a good idea, for example, this must be done with care with respect to cleaning).

It should be noted that this information could also be stored in the class-descriptor object. In that case, all objects in a class are either temporal or not. Which approach to use depends on whether orthogonality with respect to temporality is desired or not.

Compressed Object

In many cases, it is worth using some extra CPU cycles to try to reduce the size of an object before storing it in the log (see Section 7.2). An object is only stored compressed if it is beneficial, and in this case, compressed object is set to true.

This field is not used for large objects, where subobjects are independently compressed. The compression information is stored in separate subobject descriptors, which will be described in Section 8.5.2.

Class Identifier

In Vagabond, information about a class is stored in a class descriptor object (CDO) (see Section 6.2.1). The class identifier (CID) in an OD is actually the OID of the CDO of the class that the object belongs to.

It is important to note that the class identifier is a temporal property of an object, it can change during the lifetime of the object. With class migration, an object can belong to different object classes at different points in time.

The number of classes, and as a consequence, the number of CDOs, is assumed to be much smaller than the number of objects, so we can manage with a smaller size of the CID than the size of the OID. The class, and the description of it, is global for a database, so there is no need to use a *SGID*. The *CONTID* is also given implicit, we assume the ODs of the CDOs are stored in a separate container. The size of the class identifier is 24 bits, enough to represent over 16 million object classes in one database.

Signature

This optional field contains the object's signature (see Section 7.1.4). The signature field can have variable-length size, if present. Information on signature maintenance and signature size is stored in the CDO.

Inlined Object

In total, 12 bytes in the OD are used to store the physical location and size of an object. If the size of the object is less than 12 bytes, it is better to store it in the OD instead, in the physical location and object size fields. We call this an *inlined object*. Up to 11 bytes is used for the object, while the last byte is used to store the length of the inlined object.

Even though it is also possible to use the signature field for this purpose, that would complicate signature access queries, because the signature would have to be created on the fly every time.

First Version

This bit is set in the OD of the first version of an object. In some temporal operations, this can be utilized to avoid index accesses when we have this OD in main memory.

Migrated to Another Server Group

An object can migrate from the server group where it was created to another server group. In this case, the *migrated to another server group* is set to true, and the server identifier of the new server is stored in the physical location field. If the object is migrated a second time, only the OD on the server on which it was created will be updated. In this way, only one indirection is needed. By caching remote ODs on a server, this will only infrequently require network traffic.

Migrated to Another Container

Similar to migration to another server, an object can be migrated to another container. The new container identifier is stored in the physical location field. In the case the object is migrated a second time, only the OD for the first container will be updated (although it can be wise to update both, to avoid problems with ongoing updates). In this way, only one indirection is needed, similar to the case of server group migration.

8.2 Declustering

One database can be distributed over several server groups (see Section 6.4). In this case, we use one OIDX for each server, and this OIDX indexes the objects stored on this server. In the case of a multi-server system, the OID, which contains the server group identifier, is used to identify which server group to access in order to retrieve an object.

In the case of a server group (see Section 6.4), where data is declustered over the servers in the group, the declustering strategy (for example hashing of OIDs), is used to determine which server in the server group stores that object. In this way, the OIDX is implicitly partitioned.

Figure 8.2 illustrates a distributed system with server groups. In this configuration, we have 4 server groups, and each server group consists of 8 servers. Even though all server groups in this configuration contain the same number of servers, this need not be the case in general. Objects in

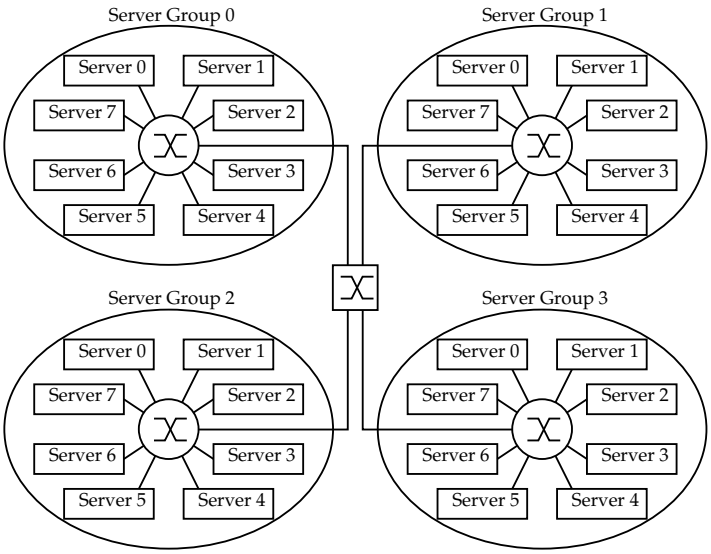


Figure 8.2: Distributed system, with server groups and servers.

one server group are declustered over the servers according to the hash value of their unique serial numbers. If we want to access an object with an OID where $SGID = 2$, $CONTID = 425$, and $USN = 84623$, the server group to access is server group 2, and the actual server in the server group is 84623 MOD 8 = 7. We emphasize that the OIDX of this server *only* indexes the objects that have been created in this server group, i.e., *SGID*$= 2$, *and* have *USN* MOD 8 equal to 7.

When indexing ODs of temporal objects, it is possible that the simple hashing strategy used in this example is not sufficient, and that other declustering schemes can be useful. This will be discussed in Section 11.

8.3 Temporal OID Indexing

In Vagabond, we use one OD for each version of an object (two ODs for migrated objects). The OIDX has to support access to ODs of current as well as historical versions of the objects, and we consider the following requirements as very important for a temporal OIDX in Vagabond:

- Support for temporal data, while still having index performance close to a non-temporal (one-version) DBMS for non-temporal data. Even if the use of other kinds of indexes could give better support for temporal operations, we believe efficient non-temporal operations to be crucial, as they will still be the most frequent operations.
- Efficient object-relational operations. This is expected to be achieved by the use of containers.
- Easy migration of partitions of the index to tertiary storage.

Before we present our temporal OIDX, we take a closer look at some characteristics of OIDs and OID search, and analyze the following four alternatives to OID indexing in a *transaction-time* temporal ODBMS:

1. One index, which indexes ODs of current as well as historical versions of the objects.

2. One index for ODs of current versions, with links to the ODs of historical versions.

3. Nested-tree index, which is one index with *version subindexes*.

4. Two separate indexes, one for ODs of current versions, and one for ODs of historical versions.

8.3.1 Characteristics of OIDs and OID Search

When considering appropriate index structures and operations on these indexes, it is important to keep in mind the properties of an OID:

- The keys in the index, the OIDs, are not uniformly distributed over a domain as keys commonly are assumed to be. If we assume the unique part of an OID to be an integer, new OIDs are in general assigned monotonically increasing values inside a container. In this case, there will never be inserts of new key (OID) values between existing keys (OIDs) in the container. In addition, OIDs will be clustered, in one cluster for each container.

- If an object is deleted, the OID will never be reused.

In a tree-based non-temporal OIDX, new entries will be added append-only. By combining the knowledge of the OIDX properties and using tuned splitting, which will be described in Section 8.4.1, an index space utilization close to 1.0 can be achieved. If container clustering is used, however, inserts in between entries occur, and space utilization will decrease. This can be avoided by using a hierarchy of multi-way tree indexes, as will be shown later.

Without container clustering, index accesses will mostly be for perfect match, there will be no key-range search (in this case a range of OIDs). With container clustering, there will be two search classes: search for perfect match during object-navigation queries, and search for all entries in one container in the case of a container scan. Accessing objects in a container will often result in additional navigational accesses to referenced objects. It is important to remember that there will in general be no correlation between OID and an object-key attribute (if defined), so that an ordinary object key-range search will not imply an OID-range search in the OIDX. If value-based range searches on keys (or other attributes in objects) are frequent, additional secondary indexes should be employed, for example B^+-trees or temporal secondary indexes. In this case, the OIDs (and time in the case of temporal queries) resulted from the key search will be sorted and then used to access the objects by lookups in the OIDX.

In a temporal ODBMS, the existence of object versions increases the complexity. For example, we need to be able to efficiently retrieve ODs of historical as well as current versions of objects, and support time-range search, i.e., retrieve all ODs for objects valid in a certain time interval. To do this, we need a more complex index structure than what is sufficient for a non-temporal ODBMS.

8.3.2 One Index Structure

If only one index is used, we have the choice of using a composite index, which is an extension of the tree-based indexes used in non-temporal ODBMSs, and using one of the general multiversion access methods.

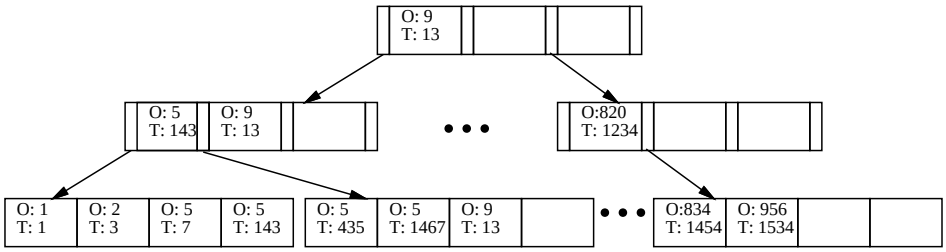


Figure 8.3: One-index structure using the concatenation of OID and commit time, $OID||TIME$, as the index key.

Composite Index

With this alternative, we use the concatenation of OID and commit time $OID||TIME$ as the index key, as illustrated in Figure 8.3. By doing this, the ODs of the different versions of an object will be clustered together in the leaf nodes, sorted on commit time. As a result, search for the OD of the current version of a particular object as well as retrieval of ODs for objects created during a particular time interval can be done efficiently.

This is also a useful solution if versioning is used for multiversion concurrency control as well. In that case, both current and *recent* objects will be frequently accessed. It is also possible that many of the future applications of temporal DBMS will access more of the historical data than has been the case until today, something that might make this alternative useful in the future. However, there are two serious drawbacks with this alternative:

1. Even in an index organized in containers, leaf nodes will contain a mix of current and historical ODs. The ODs of current versions are not clustered together, something that makes a scan over the ODs of current versions inefficient.

2. An OIDX is space consuming, a size in the order of 20% of the size of the database itself is not unexpected [62]. In the case of migration of old versions of objects to tertiary storage, it is desirable, and in practice necessary, that parts of the OIDX itself can be migrated as well to avoid the need for large amounts of disk space for the OIDX of the migrated objects. This is difficult when current and historical versions reside on the same leaf pages.

Temporal OID Indexing in POST/C++. One temporal ODBMS using a composite OIDX is the POST/C++ temporal object store [202], which is based on the Texas object store [189]. In POST/C++, objects are indexed with physical OIDs, and a variant of the composite-index structure is used to index historical versions. Because of the use of physical OIDs, a new object is created to hold the previous version when an object is updated. After the previous version has been copied into the new object, the new version is stored where the previous object had previously resided. A positive side effect of doing it this way, is that current and historical objects are separated, and that clustering does not deteriorate.

To be able to access the historical versions, a separate B^+-tree-based history index is used. This index uses the OID of the current-version object, concatenated with time, as the index key. The leaf node entry is the OID of the current version of the object, the time interval where this version was valid, and the OID of the historical version. The location of the historical version is given through the OID in the leaf node.

In a database using physical OIDs, this hybrid index structure is not a bad choice. By doing it this way, current versions will still be clustered together, and having the historical index separated from the current index (in this case no index), makes it easier to migrate historical objects to tertiary storage. The temporal index, on the other hand, can not easily be migrated.

Use of General Multiversion Access Methods

Using one of the general spatial, multidimensional, or multiversion access methods is also an alternative. However, considering the indexing problem simply as spatial/multidimensional indexing with the two dimensions OID and time will not be efficient. An object version is not only valid at a certain time (a point in the multidimensional space), but in a certain time interval (until the next version is created). In addition, a lookup for the current version of an object can be difficult with these index approaches, because time is a constantly expanding dimension. Multiversion access methods are more suitable, but the existing methods have drawbacks. We will here consider three of the most interesting methods: the TSB-tree [132],[3] R-tree [84], and LHAM [143].[4] TSB-trees and R-trees both have efficient support for time-key range search, while LHAM has a very low update cost. However, each of these access methods has drawbacks:

- LHAM is of limited use for OID indexing, because it can have a high lookup cost when the current version is to be searched for. As this will be a very frequently used operation, LHAM is not suitable for our purpose. In addition, which access method to use in the index components is still an issue.
- When indexing ODs, most queries will be OID lookups, and in this case support for key-range search is of little use.
- R-trees are best suited for indexing data that exhibits a high degree of natural clustering in multiple dimensions [178]. This is not the case when indexing ODs, and one of the results is a high degree of overlap in the search regions of the non-leaf nodes. Although a *segment R-tree* can reduce this problem, it will have a higher insert cost [178]. The fact that we do not know the end-time of a new OD further complicates the use of an R-tree.
- Using a TSB-tree or segment R-tree increases the storage space because some entries are resident in more than one node.
- In the TSB-tree, heuristics have to be used to determine when to split by time and when to split by key, and in R-trees, heuristics have to be used to determine bounding rectangles. This makes the performance vulnerable to changing access patterns.

TSB-trees and R-trees have both good support for time-key range search, and make index partitioning possible. However, when indexing ODs, most queries will be OID lookups, and when OID is the key, support for key-range search is of little use. Even if the use of TSB- or R-trees could give better support for temporal operations, we believe efficient non-temporal operations to be crucial, as they will probably still be the most frequently used operations. These multiversion access methods will increase storage space and insert cost considerably, and this contradicts our important goal of supporting temporal data, while still having index performance close to a non-temporal ODB. As

[3]There are also other B-tree-based temporal indexes, including the Write-Once B-tree, the Persistent B-tree and the Multiversion B-tree, but they do not support migration of historical data [178].

[4]See description in Section 5.4.3.