Technical Report

UCAM-CL-TR-672
ISSN 1476-2986

Number 672

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Decomposing file data into discernible items

Calicrates Policroniades-Borraz

August 2006

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted December 2005 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Hughes Hall.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/TechReports/

ISSN 1476-2986

Summary

The development of the different persistent data models shows a constant pattern: the higher the level of abstraction a storage system exposes the greater the payoff for programmers. The file API offers a simple storage model that is agnostic of any structure or data types in file contents. As a result, developers employ substantial programming effort in writing persistent code. At the other extreme, orthogonally persistent programming languages reduce the impedance mismatch between the volatile and the persistent data spaces by exposing persistent data as conventional programming objects. Consequently, developers spend considerably less effort in developing persistent code.

This dissertation addresses the lack of ability in the file API to exploit the advantages of gaining access to the logical composition of file content. It argues that the trade-off between efficiency and ease of programmability of persistent code in the context of the file API is unbalanced. Accordingly, in this dissertation I present and evaluate two practical strategies to disclose structure and type in file data.

First, I investigate to what extent it is possible to identify specific portions of file content in diverse data sets through the implementation and evaluation of techniques for data redundancy detection. This study is interesting not only because it characterises redundancy levels in storage systems content, but also because redundant portions of data at a sub-file level can be an indication of internal file data structure. Although these techniques have been used by previous work, my analysis of data redundancy is the first that makes an in-depth comparison of them and highlights the trade-offs in their employment.

Second, I introduce a novel storage system API, called Datom, that departs from the view of file content as a monolithic object. Through a minimal set of commonly-used abstract data types, it discloses a judicious degree of structure and type in the logical composition of files and makes the data access semantics of applications explicit. The design of the Datom API weighs the addition of advanced functionality and the overheads introduced by their employment, taking into account the requirements of the target application domain. The implementation of the Datom API is evaluated according to different criteria such as usability, impact at the source-code level, and performance. The experimental results demonstrate that the Datom API reduces work-effort and improves software quality by providing a storage interface based on high-level abstractions.

Acknowledgements

With the completion of this dissertation, there are several people to whom I am indebted. They have all made of my time at Cambridge a once-in-a-lifetime experience.

In the first place I would like to express my gratitude to my supervisor Ian Pratt. His advice and guidance were of fundamental importance for the completion of this project. I would also like to thank Alan Blackwell, Steven Clarke, Fred Douglis, and Tim Harris who directly influenced my research and whose invaluable comments and suggestions are reflected in this dissertation.

I owe many thanks to my proof readers and reviewers, Tim Harris, Alex Ho, Anil Madhavapeddy, Tim Moreton, Ian Pratt, and Pablo Vidales. Any remaining errors in this dissertation are all my responsibility. Furthermore, I am glad I had the opportunity to exchange interesting ideas and experiences at the Computer Lab with Iñaki Berenguer, Rajiv Chakravorty, Boris Dragovic, Evangelia Kalyvianaki, and Enrique Rodriguez.

I would also like to thank Annita for being there when it most mattered. I was incredibly fortunate to have my old friends Emmanuel, Pablo, and Zyntya with me in England during my PhD; our friendship was strengthened with all the things we lived together here, at the other side of the Atlantic. My newer friends, Luisa, Vito, and "the Mexicans" have all made my life outside the Lab extremely pleasant with tequila, salsa, and football.

Throughout this project my parents, Temistocles and Blanca, and my sisters, Palas and Damaris, have constantly supported me. They have always provided me with their advice and guidance in all the important aspects of my life.

Last but not least, my gratitude is with my sponsors. Without the support of the Mexican Government through the National Council for Science and Technology (CONACyT), it would have been impossible to accomplish this work. In addition, I thank the Brockmann Foundation and the Cambridge Overseas Trust for co-sponsoring my studies in the early stages of my PhD. Finally, I would like to express my gratitude to my College, Hughes Hall, for providing partial support in the form of travel grants that enabled me to present my work at different conferences.

Contents

List of Figures

List of Tables

Glossary

ACID	Atomicity, Consistency, Isolation, and Durability
ADO	ActiveX Data Objects
ADT	Abstract Data Type
API	Application Programming Interface
ATU	Address Translation Unit
AWT	Abstract Window Toolkit
CAD	Computer Aided Design
CAM	Computer Aided Machines
CASE	Computer Aided Software Engineering
CIM	Computer Integrated Manufacturing
CLR	Common Language Runtime
CORBA	Common Object Request Broker Architecture
DDL	Data Definition Language
DERD	Delta-Encoding via Resemblance Detection
DML	Data Manipulation Language
DOM	Document Object Model
DTD	Document Type Definition
ELF	Executable and Linking Format
FIFO	First-In, First-Out
GIS	Geographical Information Systems
HTML	Hypertext Markup Language
IDL	Interface Definition Language
IO	Input/Output
JAR	Java Archive
JDBC	Java Database Connectivity
JVM	Java Virtual Machine
LBFS	Low-Bandwidth File System
LIFO	Last-In, First-Out
LINQ	Language Integrated Query

LON	Local Object Name
LORE	Lightweight Object Repository
LRU	Least-Recently Used
MPEG	Moving Picture Experts Group
NTFS	New Technology File System
ODBC	Open Database Connectivity
ODL	Object Definition Language
ODMG	Object Data Management Group
OEM	Object Exchange Model
OLE	Object Linking and Embedding
OMG	Object Management Group
OML	Object Manipulation Language
OQL	Object Query Language
PAS	Persistent Application Systems
PCMT	Persistent Code Measurement Tool
PDC	Persistent Data Composer
PDF	Portable Document Format
PID	Persistent Identifier
PIDLAM	Persistent Identifier to Local Address Map
POMS	Persistent Object Management System
PPL	Persistent Programming Language
RDF	Resource Descriptor Framework
REBL	Redundancy Elimination at the Block Level
ROT	Resident Object Table
RVM	Recoverable Virtual Memory
SAX	Simple API for XML
SIS	Single Instance Storage
SQL	Structured Query Language
UFS	UNIX File System
UnQL	Unstructured Query Language
URI	Uniform Resource Identifier
VBWC	Value-Based Web Caching
W3C	World Wide Web Consortium
WinFS	Windows File System
WWW	World Wide Web
XML	Extensible Markup Language
XSL	Extensible Stylesheet Language
XSLT	XSL Transformations

Chapter 1

Introduction

This dissertation is concerned with the study of practical mechanisms used to disclose and manipulate structure of data in storage systems contents. It focuses on answering one main question: Do current storage systems' programmatic interfaces provide adequate support to developers dealing with file data rich in structure and type? To answer this question, my work follows two main research threads in which the first one poses issues addressed by the second one.

First, I assess the degree of internal structure and inter-file relationships that can be disclosed with techniques that employ exclusively the basic file system API (Application Programming Interface). This study exposes a number of limitations that the current file system API presents, specially in providing a judicious degree of transparency of persistent data types and structure.

Second, I have designed and implemented a semantically rich API called *Datom*, which deliberately unveils persistent data abstractions and their relationships. Datom represents an API that not only facilitates data persistence tasks but also provides useful hints about applications' data access patterns to its underlying implementation.

This introductory chapter is organised as follows. In Section 1.1, I present the main issues that motivated my research and the thesis that steered my work. Then, in Section 1.2, I summarise the contributions of this dissertation. Finally, I outline the organisation of the rest of this document in Section 1.3.

1.1 Motivation

The emergence of current persistence models has been an evolutionary process which has been guided by well-defined application requirements. *File systems* are the most general storage abstraction; they provide a flat storage space and an API to efficiently operate on arrays of bytes. *Relational databases* manipulate and represent tabular data; they support advanced query capabilities, transactions, and are meant to provide support for data-intensive application contexts.

Object-oriented database systems aim to provide all the well-known advantages of relational databases and, additionally, to efficiently support more complex data representations through the use of the object-oriented data model. Additionally, *semistructured data* appears as a way to support seamless data exchange and as a solution to bring into the database context information that due to its decentralised management cannot be restrained to strict database schemas.

Finally, *persistent programming languages* merge the programming language and the data store into one system at runtime. Recent implementations of this technology accomplish persistence in such a way that developers cannot perceive the difference between persistent and volatile data. In this evolution of persistent data models, there seems to be a constant pattern: the more abstraction a storage system API is able to expose the greater the payoff for the programmer.

This observation becomes obvious if both extremes in the range of storage systems APIs are compared. On the one side there is the file system API; it provides simplicity and generality and is agnostic of any structure or type in file contents. In order to manage applications' persistent data, developers have to repeatedly write large amounts of code using an API that makes this task a tedious and error prone process. On the other side there is the persistent programming languages' approach; it aims to reduce at a maximum the impedance mismatch between run-time and persistent abstractions by enabling orthogonal persistence. As a result, programmers spend considerably less effort developing persistent code (i.e. code concerned with loading and storing persistent data).

Whether the trade-off between efficiency and programmability in the context of file systems applications is well balanced has received especial attention recently. Core to this discussion is the lack of ability that the file system API exhibits in handling any degree of abstraction in persistent data. This represents an important disadvantage for developers in terms of the simplicity in which sophisticated data access strategies are programmed.

The analysis of persistent data in applications that employ file systems as their persistent substrate indicates that there is a considerable amount of data rich in structure and type. Furthermore, it suggests that data stored in files is amenable to structural decomposition; even binary data such as bitmaps or sound files can be included in a multimedia document where they can be explicitly distinguished from each other under certain file organisations.

Exploiting these abstractions may be useful not only to improve some of the most fundamental underlying mechanisms of storage systems such as caching, data prefetching, and concurrency control but also to improve the development of code related with the management of persistent data. Although the two aforementioned issues are recurrent in the course of this dissertation, the research effort is mainly focused on providing an answer to the latter.

Unfortunately, relying on the type of facilities that are offered by the current storage APIs in order to improve the manipulation of file contents presents a group of important shortcomings:

- **Addition of overheads**. Databases introduce overheads related to the use of high-level query languages to access data. Furthermore, system complexity considerably increases by the implementation of transactional frameworks that support ACID semantics into its design. More intricacies are added to the system if the object-oriented data model is supported.
- **Loss of generality**. Providing orthogonal persistence through the use of a persistent programming language confines type support to a specific language compiler and implies the adoption of a programming model. As a consequence, developers are restricted into specific programming languages with their very own features which may be difficult to implement or may not be available in other programming languages. Ultimately, this reduces not only the longevity of the persistent data but also the generality of this type of solutions.
- **Inefficient storage APIs**. The group of programmatic APIs that rely heavily on the stream-based data manipulation approach coined by semistructured data tend to be inefficient. They consume memory resources aggressively or require programmers to write code in order to disclose typed contents or internal file data layouts.

There are a number of application contexts running on top of file systems that could benefit from higher levels of abstraction but in which paying the price of extra overheads or inefficiency is not ideal. Instead, these applications demand a data-centric approach and are characterised by well-defined access patterns with varying recoverability and consistency requirements.

This dissertation addresses the question of how to systematically augment the degree of abstraction managed by a storage system API, whose application context relates to that of applications using the file API, without heavily unbalancing the before mentioned trade-offs. The strategy followed to attack this problem is twofold.

First, I assess the amount of internal structure and data correlation that is feasible to unveil through the implementation and evaluation of diverse data redundancy detection techniques; these methods employ the generic IO services provided by the file system API. This study represents an original piece of work whose results provide important guidelines for the design and implementation of storage systems. It also shows that there are important limitations in using the file API to expose persistent data abstractions such as structure and data type. Accordingly, in the second line of research of this dissertation I propose, implement, and evaluate a storage API whose main tenet is the use of high-level abstractions in the manipulation of persistent data.

It is my thesis that data manipulation strategies departing from the fundamental flat file paradigm to a more abstract data representation, capable of retaining a sensible amount of structure and data type, have important advantages for programmers. They improve software quality, ease developers' programming duties, and support the creation of more sophisticated data access strategies with less effort.

Data manipulation tools and interfaces in existing storage technologies prove to be badly suited for an important group of applications; there is a mismatch between their offer of persistence and the requirements of practical applications. In contrast, my approach favours simplicity, efficiency, and generality while providing to application developers tools to systematically and gradually permeate their programs with semantically rich persistent data abstractions.

1.2 Contribution

The key contributions of this dissertation are two. First, I have assessed the degree of internal structure and inter-file relationship that can be disclosed through the use of the facilities offered by the generic file system API. With this purpose, I have implemented and evaluated three techniques that inspect file content at different granularities such as whole-file analysis, page-aligned data segments, and content-defined data chunks in search of meaningful portions of data.

Second, I have designed, implemented, and evaluated a novel storage system API that captures a sensible degree of structure and data type. There are two main distinctive features of this API. First, it enforces type safety in the manipulation of persistent abstractions. Second, it simplifies the management of persistent data layouts through a minimal set of general data abstractions that resemble the semantics of *list*, *map*, *matrix*, *stack*, and *queue* objects. The manipulation of persistent data through this API impacts in a positive way the interactions of programmers with persistent code.

1.3 Outline

The rest of this dissertation is organised as follows. In Chapter 2, I provide the background and research context that is useful for the rest of this document. The rationale behind existing storage paradigms is presented and the trade-offs faced by each technology are examined. Finally, I discuss some of the issues later addressed in my proposal of a novel storage system API.

In Chapter 3, I evaluate the use of the file system API as a tool to unveil structure and data correlations in file system contents. An essential observation is that data redundancy at a sub-file level might well suggest internal file data organisations and structure. Therefore, I study to what extent is possible to identify specific portions of data in considerably large data sets following a systematic approach.

In contrast with the techniques analysed in the previous chapter, in Chapter 4 I introduce Datom, a storage API that departs from the view of file content as a monolithic object by unveiling data types and internal data organisations. Its main goal is to assist programmers to create advanced application-specific data access strategies in a simple yet effective manner. Furthermore, it avoids the addition of functionality that may add unnecessary overheads and compromises for its target application domain.

In Chapter 5, I describe the architecture and implementation of the Datom API. The prototype represents a fully operational implementation of the API that advocates the data manipulation paradigm of the Datom API. The strength of the implementation of the Datom API is levered by the provision of a persistence model that supports its data model in an effective and efficient manner.

In Chapter 6, I present the evaluation of the Datom API according to different criteria including its impact at the source-code level, performance, and usability aspects. The results demonstrate that the Datom API reduces work-effort and improves software quality by providing a storage interface based on high-level abstractions.

Finally, in Chapter 7, I conclude this dissertation and present ideas for future research.

Chapter 2

Background

In this chapter I present background information related to data access and manipulation services enabled by today's storage systems. The goal of this chapter is twofold. First, to provide a programmer's point of view of the main data storage technologies and their programmatic APIs. And second, to highlight how particular storage systems' architectures are shaped by both their data model and the data access facilities they present to programmers.

2.1 Introduction

This chapter puts this dissertation in perspective with the main persistent data manipulation paradigms; Figure 2.1 shows the road map including the major sections that comprise this chapter. The main goal in this analysis is to recognise the mechanisms that different technologies employ to disclose abstractions present in storage systems content.

The technologies presented in Sections 2.2, 2.3, and 2.5 are analysed from three perspectives: data model, data manipulation facilities, and implementation and systems. The *data model* supported by a storage system has the biggest impact on its potential to capture application persistent abstractions such as type, structure, or data relationships. In addition, the *data manipulation* facilities offered by each storage paradigm represent the main conduit to the manipulation of the underlying storage system data model. These two features shape the way in which developers manipulate persistent data abstractions at the application level. The combination of a data model together with the data manipulation facilities that a particular storage system supports has also direct impact on its *implementation and systems.*

Section 2.4 is dedicated to persistent programming languages and follows the same lines of analysis. However, a more detailed presentation of the most influential works is done for two reasons. First, persistent programming languages represent the peak of evolution in the manipulation of persistent data in the form of programming abstractions. Second, the ideas supporting these systems highly influenced the design of Datom. The adoption of a persistent technology is based upon application-specific requirements. The trade-offs between programmability, overhead, and applications' functional requirements abound. This chapter contains the necessary background to promote this discussion; a critical review of the different storage paradigms analysed through this chapter is presented in Section 2.6.

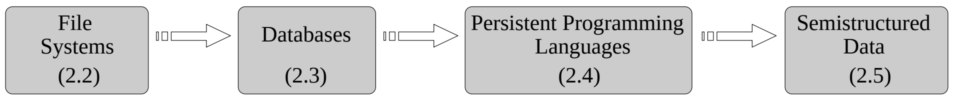

File Systems (2.2)
Databases (2.3)
Persistent Programming Languages (2.4)
Semistructured Data (2.5)

Figure 2.1: Road map of different storage technologies analysed in this chapter.

2.2 File Systems

File systems trace their roots to a system proposal for secondary storage for the Multics Operating System dated back in 1965 [DN65]. This proposal later became the foundation of the UNIX File System (UFS) which in turn influenced subsequent file systems designs. A core goal of this proposition was to hide from programmers the complexities related to explicitly manipulating physical addresses in the hierarchy of secondary storage devices, leaving all this work to the file system itself.

In this way, developers would be able to focus their programming efforts on developing code to recover applications' persistent data without explicitly managing the physical location of it. In words of the authors, "any consideration which is not basic to a user's ability to manipulate this information should be invisible to him...". File systems were, undoubtedly, an important improvement to the way in which programmers accessed data in secondary storage. The initial view and associated file system abstractions remain practically unchanged even in current implementations of this technology.

Data Model

File systems are one of the most general approaches to storing data. They present two fairly simple data abstractions to programmers, *files* and *directories*. These elements are uniquely identified through the use of symbolic names. The file system structure is a tree-like hierarchy composed of files and directories. Thus, a directory can be thought as a set of logically associated files and other directories.

From a programmer's perspective, a file is simply a named collection of data stored in a device. Storage devices store linearly addressed blocks of bytes. The file system provides an abstraction on top of storage blocks to offer a data structure that logically represents a collection of information to programmers. Conceptually, such an abstraction allows programmers to store and retrieve an arbitrary stream of linearly addressed bytes on the block-oriented storage device.

Data Manipulation

The file system API offers a basic set of calls to operate on the stream of bytes that are part of a file. Each byte in the stream is addressed with a non-negative integer and the file position is used to reference specific bytes in the file. To modify the content of a file, programmers make use of the well-known `read/write` file system API [Kle86] in which explicit handling of file position and byte indexes are required.

In general, the majority of file system applications have a well-defined behavioural pattern. They open one or more files, process the data stored in them, and then write the result of the computation back to a file again. Examples of this pattern exist in large quantities: a word processing application opens a document, presents it for editing, and then saves changes back to disk; the Java compiler reads one or more source code files, transforms these files to bytecode, and writes the output to the corresponding class files. Even the general UNIX process model with the use of `stdin`, `stdout`, and `stderr` devices resembles the aforementioned operational pattern.

Under this computational model various observations are important. First, programmers have to repeatedly *write code from scratch* to translate byte streams into run-time data structures. In the vast majority of cases this translation is to be done using a third generation programming language that, although shown to be practical, leaves an important amount of work still to be done by the programmer. This *impedance mismatch* [CM84] between two completely different data representations (i.e. files and programming abstractions) imposes extra work on programmers which is tedious and error prone. Second, when applications operate on data, they rely on the presence of *structure and data types*; these two features are well embodied at run-time in collections of meaningful programming abstractions such as structures, objects, or records.

Implementation and Systems

Leaving the translation work to applications is a deliberate design decision which trades programmability in favour of efficiency. Thus, along the evolution of file systems there has been constant work in improving performance. Initial file systems used to exploit only 4-5% of raw disk bandwidth and compare very poorly with modern implementations, which are reported to use near-raw IO performance [BFH02]. However, the file abstraction has remained without major change for decades regardless of the different on-disk data formats, underlying implementations, or data location strategies that have been investigated [San86, Sat89, Nag97, MT03].

In its simplest form a file is a collection of ordered bytes. However, some systems place additional structure on files. Rather than reading single bytes or seeking to arbitrary offsets, files are accessed in terms of records. Organising a file in records implies the existence of a schema either embedded in the file in a way that the operating system can read or separately in the system. Record-based files impose a structure on the data and allow the operating system to keep that structure intact. The main drawback of this approach is that it lacks flexibility since adding a new format or translating between formats is frequently difficult. Historically this kind of file was used in old mainframe systems [Nut00].

In an attempt to move from the traditional file system API to expose a finer degree of data granularity the OLE Structured Storage Model [Bro94] was developed by Microsoft; it represents a solution to the problem of internal file structure. The API isolates programmers from managing the exact placement of application's data within files. Programmers also avoid the burden of directly dealing with internal file fragmentation issues leaving this task to the file system infrastructure itself. However, the abstractions presented to programmers (i.e. *compound file*, *storage* and *stream*) have a strong parallel with directories and files respectively: they are organised hierarchically, and their respective interfaces show strong similarities to a conventional file system API.

Other groups have also explored the utility of data structures and higher-level abstractions as a storage infrastructure replacing the traditional view of a flat file. One of the main advantages of this approach is a more structured interface in which operations are applied to a data structure instead of to a range of bytes. Ultimately, this enables programmers to concentrate on creating application services and focus on the application logic. Gribble et al. [GBHC00] propose distributed data structures (e.g. hash tables, B-trees) as a persistent data management layer to replace Internet services commonly provided by file systems' infrastructure.

Under the same line of reasoning, the Boxwood project [MMN$^+$04] explores the utility of high-level data structures as storage infrastructure. The rationale behind both projects is that higher-level abstractions and structural information inherent to the data abstractions can enable the system to perform better load-balancing, data prefetching, or informed caching. They also point out that programmers' coding duties are alleviated by having at hand high-level abstractions instead of the file system API. However, no experimental proof of this assertion is provided.

In summary, file systems represent a fundamental abstraction for storing applications' persistent data. The potential benefits in efficiency and low overheads enabled by their underlying infrastructure are highly valued in certain application domains. In contrast, file systems' data model is extremely simple and offers few advantages to applications dealing with persistent data rich in structure and type.

2.3 Databases

2.3.1 Relational Databases

Before the relational data model came into existence in the early 1970s, there were two paradigms to build database systems: the hierarchical and the network data models. In the *hierarchical data model* [EN00a] all data records are assembled into a collection of trees. The logical model specifies a well-defined group of root records with the rest of the records having unique parent records. The query language permits to navigate the hierarchy by accessing one record at a time [McG77]. In a different way, the *network data model* [EN00b] proposes the organisation of data records into a directed graph. Similar to the hierarchical data model, a navigational query language is used to move through records in the graph [Bac73].

These two models exhibited important shortcomings for the sort of data access requirements of the applications they were aimed to service. Presumably, their main weakness was that of having a low-level programming interface between application programs and the database system. Thus, changes in query requirements, application's data types, or structural composition of the database used to involve costly code updates and program maintenance. Both the hierarchical and the network data models are currently in disuse and are briefly mentioned here for completeness reasons.

Data Model

The **relational data model** [Cod70] proposed by E. F. Codd offered a conceptually different approach to data storage. Codd suggested to represent data as *tabular* data structures to be accessed through a *high-level non-procedural query language*. Thus, relational databases replaced the navigational model seen in previous database APIs with a fully associative model.

All data in a relational database can be uniquely addressed by means of the relation name, primary key value, and attribute name. Associative addressing of this form isolates programmers from explicit manipulation of the low-level details of data placement and of any explicit indication of the appropriate access paths for retrieving data [Cod82].

Relational databases, and database systems in general, are designed to fulfil the data access requirements of a particular group of applications [SSU91]. These applications are representatives of *data-intensive* environments in which large amounts of data have to be accessed, queried, and updated efficiently. Furthermore, their data semantics are highly sensitive to risks associated with *concurrency* and *data contention*, thus needing support to enforce a consistent policy to control data access.

Additionally, *persistent data has to survive* in the face of hardware crashes or software errors. Usually, these applications manipulate relatively simple abstractions that can be mapped into the tabular data model offered by relational databases.

Data Manipulation

Over the years, the database research community has extensively investigated, amongst other issues, the creation of database APIs to define, query, and manipulate databases. In general, a database language is made of a data definition language (DDL) to specify the database schema and a data manipulation language (DML) to express database queries and updates.

DMLs can be subdivided in two categories [EN00c]. The first category is commonly defined as high-level *non-procedural* DMLs (also referred as declarative DMLs in the literature) whereas the second type is better defined as low-level or *procedural* DMLs. The next paragraphs focus on discussing mainly high-level non-procedural DMLs as they are the favoured mechanism employed in relational databases. Procedural DMLs in the context of databases are more related with object oriented database systems and persistence programming languages; for that reason, they are discussed later in Section 2.3.2 and Section 2.4, respectively.

The Structured Query Language (SQL) is certainly the most popular example of a non-procedural DML; it was designed from the outset to work with the relational data model. SQL represents a comprehensive database language with statements for data definition, query, and update. Additionally, it has facilities for defining integrity constraints, for specifying transaction control, for defining views on the database, and for declaring security and authorisation. Since its creation, it has undergone many enhancements which have led to different standards known as SQL [ANS86], SQL2 [ANS92], and SQL3 [ISO99].

From a programmer's point of view, to manipulate the database it is necessary to include database statements in a host language. In general, two mechanisms are used with this purpose. The first option, presumably the most popular due to the isolation that it provides between the host programming language and the database system, facilitates an *intermediate layer* in the form of API calls that send SQL statements to the database (e.g. ODBC [Gei95] or JDBC [FEB03]).

In this case, the role of the host language is limited to invoking API calls to send queries to the database server and to receive the result generated by those invocations. Therefore, similar to the file system API, programmers have to create a set of routines and write code to perform any pertinent conversion between the data returned by the database and the application run-time abstractions in the host programming language.

The second strategy *extends the host language* syntax to embed database calls in the application code [EM99]. Application developers can use the augmented language to access and update the database; all query processing is done by the database system. Application programs must be precompiled in order to replace embedded SQL statements with host-language declarations and procedure calls that allow run-time execution of the database accesses [Moo91, PRO02a, PRO02b]. Finally, the resulting program must be compiled by the host-language compiler. In general, query results are stored in a temporary relation in the database and fetched from the host language on a per-record basis using a combination of looping control statements and traversal pointers provided by either the extensions to the host programming language or the programming language itself (e.g. iterators or database cursors).

Implementation and Systems

Having a high-level non-procedural DML such as SQL to manipulate database content has direct implications on the way in which databases are designed. To properly support relational database data access patterns [DYC95] in which dynamic querying of large volumes of data is commonplace, it is necessary to provide specialised functionality in charge of *query execution* [Gra93], and *optimisation* [JK84, RSSB00, DSRS01].

In addition, to manage data contention database systems rely on *transactional frameworks* [GR93] which guarantee that data is read and updated safely and efficiently regardless of *concurrent access* or *media failure* [KH98, LGWJ01]. Therefore, databases are complex systems made of various specialised software modules, which altogether add important overheads in exchange of essential functionality for those applications that employ them.

The main trend in the evolution of SQL has been the addition of features to provide to developers a more comprehensive storage model; aiming to leave behind the exclusive manipulation of tabular data by directly including in the language and its API facilities for more complex data representations such as those used in the object oriented world [SRL$^+$90]. Thus, database schemas were incorporated from SQL to SQL2.

In turn, the SQL3 standard enriched SQL with object-oriented features and extended the syntax with programming language facilities such as control structures to make of SQL a computationally complete language. The addition of object-oriented concepts into relational databases has gained importance and therefore it is discussed separately in Section 2.3.3. SQL is supported by a large number of commercial databases such as IBM DB2 [Pad02, DB205], Oracle [Jak02, ORA05], Microsoft SQL Server [ABC$^+$02, SQL05], Sybase [SYB05], and MySQL [MYS05]; just to mention the most important.

2.3.2 Object Oriented Databases

Relational databases' design was initially driven by requirements of applications in business and administrative areas. These applications are characterised by relatively simple data representations in which the tabular data model offered by relational databases sufficed. Their application data is characterised by few record types which are fairly small in size and their values are atomic (or can be decomposed to be atomic).

However, relational databases failed to properly support other types of applications. Examples of these applications are commonly associated with engineering applications such as CAD/CAM (Computer Aided Design/Computer Aided Machines), CASE (Computer Aided Software Engineering), and CIM (Computer Integrated Manufacturing), or to multimedia systems such as GIS (Geographical Information Systems), image management systems, medical systems, and decision support systems [BM93].

Data Model

An **object oriented database** system is built based on the object oriented paradigm. They have been influenced by and finally adopted many of the ideas that were originally created in the context of programming languages. Object oriented programming languages are designed to better model the mapping between real-world and programming abstractions as well as their interactions. Although the initial debate on which language and type system features should be included into a full database object model is still under consideration [LOS02], at the present time it is possible to distinguish a set of properties that can be recognised as core requirements for a database object model.

These features were initially pointed out by Atkinson et al. [ABD$^+$89] and can be considered to be a definition of an object oriented database system. Application data in an object oriented database is modeled as *objects* that have associated a *unique identifier* [KC86]. An object is made of a group of attributes which can include other objects, which enables the construction of more *complex objects* and data representations. Additionally, objects *encapsulate* interfaces and methods that can be accessed and invoked by other objects.

Objects have associated a *type* [Gut02] meaning mainly an Abstract Data Type (ADT) of the object [ES90] or a *class* referring to a run-time notion through which objects can be created and manipulated [GR83]. To a certain extent, both concepts can be used to summarise the common features in a set of objects. To augment the expressive power of the database system, complete hierarchies of objects can be created through *inheritance*. A class can be defined as an instance of one or more existing classes and thus inheriting the attributes and the methods of the base class.

Inheritance has been identified as one the most powerful concepts of object oriented programming as it enables reusability and augments the ability of the database system to express complex abstractions [Ban88]. Finally, different methods can be defined to share the same name leaving to the system itself to determine which method should be called by using *overloading*, *overriding*, and *late binding* properties.

A common object model has been historically pointed out as one of the main weaknesses of this kind of technology. At the time of writing, the Object Data Management Group (ODMG) [ODM05] has already proposed a standard which is made of several parts [CBB$^+$00]: the object model, the Object Definition Language (ODL), the Object Query Language (OQL), and the bindings to programming languages. The object model provides the data types, type constructors, and other concepts that can be used in the ODL to specify object database schemas. The ODL[1] is independent of any particular programming language and is designed to support the semantics defined by the ODMG object model. It is mainly used to create object specifications such as classes and interfaces.

The OQL is very close to SQL with extensions to support ODMG's object oriented concepts such as object identity, complex objects, path expressions, polymorphism, method invocation, and late binding. An OQL statement written inside any of the programming languages for which a binding is defined can return objects that match the type system of the host language. However, the type system of the ODMG's object model may be thought only as a subset of the

[1]The ODL is designed to be compatible with the Object Management Group (OMG) Interface Definition Language (IDL) of the Common Object Request Broker (CORBA) [OMG05].

full type system that could be offered by the host programming language. The OML is the language used for retrieving objects from the database and modifying them. At the present time OML bindings have been defined for C++, Smalltalk, and Java.

Data Manipulation

In object oriented databases, programmers manipulate data objects and define the implementation of their methods through the programming languages supported by the database. In this respect, it is possible to distinguish three main approaches. First, some systems provide support for various programming languages through a data definition language and bindings to different programming languages, specially those commercially oriented and thus compliant to the ODMG object model (e.g. [GEM05, OBJ05]). Others support a more integrated view of the programming language and data objects through explicit insertions and modifications to the host programming language [Car86, RCS93].

Finally, those in the third category provide tight integration of the database and the programming language type systems with minimal modifications to the semantics of the host programming language [ABC$^+$83, SKW92, JA98]. These systems are designed to support *orthogonal persistence* [AM95]. Section 2.4 presents the main trends in database programming languages in which a detailed analysis of the programmatic facilities provided by these systems as well as of their underlying implementations is done.

Implementation and Systems

From a databases point of view, it is indispensable to support the characteristic functionality of a database system and adapt it to the application domain for which object oriented databases are targeted. An object oriented database must provide *transactions* and *concurrency control* not only for regular transactions but ideally also for long transactions [KS90, Mil99]. Object databases have commonly implemented concurrency control through two-phase locking protocols [Deu91], and varying between optimistic and pessimistic concurrency control schemes according to the amount of potential data contention [BOS91].

Version management and *dynamic schema evolution* are used to access multiple versions of an object, and to maintain historical databases; these features have been supported in various object databases [CDG$^+$90, BOS91, LLOW91]. *Recovery* and protection from media failures are also implemented in the majority of these databases [FZT$^+$92]. In general, object oriented databases are more complex than traditional relational systems due to long-duration transactions and the coexistence of different valid states (i.e. versions) of a particular object. Furthermore, they are equipped to support highly complex data relationships.

Detailed comparative analyses of the features supported by different object database systems can be seen in [Sol92, ZCC95, CO96]. A more technical review of particular storage management architectures has been done by Weddell in [Wed91]. Various object oriented database systems have been developed as research projects such as EXODUS/EXTRA [CDG$^+$90], and some of them have been been commercially exploited and have presence in the database market at present including GemStone [BMO$^+$89, BOS91, GEM05], ObjectStore [LLOW91, OBJ05], O_2[Deu91], and Versant [VER05].

2.3.3 Object Relational Databases

The success of the relational data model lies on a very robust infrastructure and maturity of the mechanisms that have been developed to support it. However, the relational data model and their programmatic APIs fell short in providing adequate support for more complex kinds of data and manipulation requirements for which the object oriented data model is better suited. This kind of applications required support for complex data representations and rich querying services.

Data Model

Object relational databases are mainly characterised by a set of features and properties defined by Stonebraker et al. [SRL$^+$90]. Three tenets are the basis of this proposal. The *first tenet* is concerned with the addition of richer objects and rules to the database system. It stands for a rich type system, inheritance, database procedures, methods, and encapsulation. Moreover, it includes constructors such as lists and bags to operate on objects or collections of objects.

The *second tenet* subsumes the traditional relational model into the object relational data model and it is directly concerned with how database functions should be written and accessed. This implies that all programmatic access to a database must be done through a non-procedural high-level language (e.g.

SQL-like) while supporting data independence; any kind of low-level data access dependent on the physical implementation of the database must be avoided. Additionally, object relational databases should keep backward compatibility to the traditional relational model.

The *third tenet* deals with the implementation of open systems and with the API exposed by the new breed of object relational databases. It supports SQL as the base high-level language from which access to the database has to be done. Persistence in programming languages should be supported on top of a common database system by compiler extensions to the programming language and a run-time system.

As noted, many of these ideas are in opposition to those presented by Atkinson et al. (Section 2.3.2), specially those related to the way in which object oriented abstractions should be presented and manipulated by programmers. Object relational databases incline towards query-based data access instead of physical navigation through low-level procedural interfaces favoured in object oriented databases.

Furthermore, a multi-lingual database environment contrasts with the tight integration of particular programming languages to the database system; data access strategies should use SQL and extend it where needed to facilitate connectivity and interoperability between different software vendors. To a broader extent, the core point of disagreement is that instead of creating from scratch a whole new infrastructure for database systems, the object relational community proposes to build on top of the already tested and matured relational technology.

Darwen and Date [DD95] manifesto presents a different view on the future of database systems to those given by Stonebraker et al. [SRL+90] and Atkinson et al. [ABD+89]. In general, they all agree on augmenting the database type system with object oriented features. However, Darwen and Date criticise the dismissal of the relational model in Atkinson et al. proposal. They discuss the orthogonality of object oriented features and the relational model as initially presented by Codd [Cod70] stating that the relational model does not need to be extended to support them.

They agree with Stonebraker et al. on the use of the relational model and the development of a new high-level language. However, they significantly differ on the use of SQL as the database programming language of the future. For Darwen and Date SQL should be rejected unequivocally, and instead, a new database language that adheres firmly to the relational model of data should be designed.

Data Manipulation

According to the generally accepted relational database philosophy, query languages must be built on the ideas already proved in SQL. Thus, it is possible to observe how different query languages augment features to the original relational language. The O_2 query language [BCD89] is an object oriented extension of SQL implemented in the O_2 object oriented database system [Deu91]. Similarly XSQL [KKS92] is an object oriented extension to SQL that integrates extended path expressions as well as type correctness.

UniSQL [DJ96] is a database-centric approach that provides discretionary object oriented modeling features for inheritance, encapsulation, pointers, and collections. It was conceived as a system to unify the relational and object oriented data models in the context of databases. It empowers programmers with navigational object access and an ad-hoc non-procedural language called SQL/X which integrates class composition, collections, encapsulation, and inheritance into the SQL language.

As part of the evolution of SQL, the SQL3 [ISO99] standard has been defined. The SQL3 standard enriches SQL with object-oriented features, and extends the syntax with programming language facilities such as control structures to make of SQL a computationally complete language. SQL3 includes object oriented features such as inheritance, overloading, resolution of functions based on the type of the arguments, as well as user-defined data types, type constructors, collection types, user-defined functions, and procedures.

Implementation and Systems

Ideally, an object relational database aims to exploit all the operational assets of conventional relational technology (see Section 2.3.1). Consequently, the augmented level of abstraction enabled by the object relational model is commonly supported directly on top of traditional relational database technology.

A number of object relational database systems have been developed to date. POSTGRES [SK91] is an early implementation of an object relational database system. Its initial aim was to keep all the features of the relational model and to make them work with enriched object oriented capabilities. The commercial realisation of POSTGRES can be traced in the subsequent database products that followed it (i.e. Illustra, Informix [INF05]). Similarly, the Iris [KLMW90] database system provides object oriented extensions on top of a relational database system. OZ+ [WL89] is an object oriented database built on top of EMPRESS, a relational database management system as the underlying disk storage and access mechanism.

The largest database management systems vendors seem to support object oriented features on top of the extensively tested and well-known relational infrastructure [McC97], while the object oriented database market is shared among companies which are mainly offsprings of academic research. Although object oriented databases are better adapted for the object oriented application domain, today's database market is dominated by the extensive support provided by large database vendors to the object relational paradigm.

2.4 Persistent Programming Languages

Persistent Programming Languages (PPL) appear in the scene to fill the gap between database systems and programming languages. As highlighted in the previous two sections, database systems are primarily concerned with the creation and maintenance of large collections of data with particular data access requirements (i.e. dynamic query, data sharing, and performance). On the other hand, programming languages provide support for procedural control, and built-in support for data type definitions and abstractions [CNTR97]. PPLs attempt to merge the database and the programming language at runtime.

Historically, PPLs have been initially implemented as stand alone systems which aimed to provide persistence to a programming language without giving too much emphasis to data access requirements of database systems. Consequently, they were built mainly by transforming particular programming languages and not defined by any standards. As research in PPLs and database systems progressed (specially object oriented databases), PPLs began to provide access to full-fledged database systems.

The goal of PPLs is to fulfil the requirements of a type of system often referred in the literature as Persistent Application Systems (PAS). Typical examples of these systems are CAD/CAM systems, office automation, CASE tools, software engineering environments, large scientific databases and programs that analyse them, and process modelling systems [AM95]. They require to support complex data models, and concurrent access to large bodies of data and programs.

The first attempt to provide some level of database functionality in a programming language was Pascal/R [Sch77]. In this work, the programming language is extended with two new data types: *database* and *relation*. From a modern perspective of PPLs, this premier attempt suffered several downsides. First, a database was restricted to store exclusively objects of type relation, hampering the possibility of including any of the other Pascal types. Additionally, relations were tuples whose fields had to be atomic base types such as `int` and `char`.

Given the important limitations in terms of the data types supported, programmers had to convert applications' rich run-time structures into fairly simple relations; similar to the way in which data has to be serialised to be stored in a flat file. Furthermore, as databases were implemented on top of conventional files, only one process was able to access the store. Finally, programs were run in a single transaction which made data sharing impossible.

Plenty of research has been done in this direction since the appearance of Pascal/R to overcome these and other problems. Next, a review of the most influential work in the area is discussed; the systems described below are presented in chronological order to provide an evolutionary perspective of advance in the field. The analysis concentrates on the data model presented to programmers, and their implementation strategies.

PS-Algol

PS-Algol [ABC+83] was built by extending the facilities of a conventional programming language, i.e. S-Algol [CM82]. It is generally acknowledged as the first programming language that supported the notion of **orthogonal persistence** [AM95]. As a result, any object in PS-Algol can persist irrespective of its type. Furthermore, persistent type instances are of the same type as their volatile versions, and can be operated in the same way. The definition of orthogonal persistence highlights three principles:

- **Persistence Independence.** The form of a program is independent of the longevity of the data that it manipulates. Programs look the same whether they manipulate short-term or long-term data.
- **Data Type Orthogonality.** All data objects should be allowed the full range of persistence irrespective of their type. There are no special cases

where objects are not allowed to be long-lived or are not allowed to be transient.

- **Persistence Identification.** The choice of how to identify and provide persistent objects is orthogonal to the universe of discourse of the system. The mechanism for identifying persistent objects is not related to the type system.

The persistent store is modeled as a collection of database files in which persistent programming objects are stored and can be manipulated transparently once the database file has been opened. Opening a database results in a *root* object being returned. The root acts as a name service directory from which other elements can be accessed. All objects *reachable* from a root of persistence will be saved into the store at commit time. Updated and new objects are moved automatically from the store to memory and vice versa by the runtime support system. Objects are updated on a program completion call or by explicit closing of the store by the programmer.

The best known implementation of PS-Algol is the Persistent Object Management System (POMS) [CAC+90]. In this implementation objects are identified with a 32-bit reference which can be either a persistent identifier (PID) or a local object name (LON); a PID is identified by having the most significant bit of the reference set to 1. **Object faulting and swizzling** are used to move data between memory and stable storage. When in disk, objects can be located using the PID which in turn has to be swizzled and converted to a LON format, i.e. a memory reference.

This process is supported by a two-way accessible table called Persistent IDentifier to Local Address Map (PIDLAM). This data structure has an entry for every object in memory containing a mapping between the PID and its LON, or vice versa. An attempt by a program to dereference a PID is trapped by the PS-Algol interpreter which triggers a lookup in the PIDLAM. If an entry is found, i.e. the object is already in memory, then the PID reference is replaced by its corresponding LON. Otherwise, it has to be loaded from disk and a new entry has to be created in the PIDLAM.

As POMS implemented a lazy strategy to solve persistent references, newly fetched objects can hold PID references to other objects. These references will not be resolved until the program attempts to use the objects to which they refer. When an object has to be unmapped from memory, any swizzled references referring to that object must be unswizzled so that future faults of the object can be detected. To avoid looking for the references of the evicted object in the set of resident objects, unmapping objects takes place only at program termination.

CPOMS [BC85] uses the same underlying mechanisms as described above with two optimisations. First, references to objects in the stack never contain a PID. This is accomplished by adopting an eager swizzling strategy; all references in an object are resolved at the moment of inclusion in the stack. Furthermore, LONs are not addresses but the indexes of the corresponding object entry in the PIDLAM. Thus, accessing an object always involves an indirection via the PIDLAM. However, when an object has to be unmapped, it is only necessary to update its PIDLAM entry to allow future faults of the object to be detected. It enables efficient removal of objects from memory at any time in the execution of the system. Napier88 [DCBM89] is recognised as a successor of PS-Algol, and it was built on the same principles.

Amber

Amber [Car86] is a descendant of the ML programming language [Mil84]. Among its main goals was to blend static typing with the dynamic requirements of a programming language. Thus, multiple inheritance and persistent objects were integrated in a strongly typed programming language.

In Amber any value in the programming language (i.e. value, data, or program) can be exported to persistent storage and later imported and coerced, even during different program sessions. Complex data structures can be exported and imported without having to write ad-hoc routines to serialise and deserialise them. The store and the programming language are **weakly coupled**; the store serves only the purpose of persistent value archival. Amber's persistent store sits on top of the underlying file system; files are created accordingly to store the persistent representation of objects. No support for persistent data sharing is supported as the system is designed to be used in single-user environments.

Two routines are offered to programmers to manage data access in the store. The *export* routine saves a value in the store under a given name. Later, programmers can use the *import* routine to recover values under associated string identifiers. In Amber **persistence is explicit**, which means that the programmer has to write code indicating his/her intension to send a value to the store. There is no effective restriction on the type of objects that can be stored in the database. However, to be persistent objects have to be converted into instances of the `Dynamic` type. This means that persistent objects, once read, have to be coerced back into a copy of their volatile type to be properly manipulated. Updates to these objects will only reach the store if indicated explicitly by the programmer.

E

The E programming language [RCS93] is an extension to C++ [ES90] developed as part of the EXODUS database management system [CDF+86, CDG+90]. Although originally designed to support the creation of database systems, it evolved into a more general PPL. In addition to the persistence extension to C++, E also provides support for generic classes and an iterator control abstraction inspired by iterators in the CLU programming language [Lis93].

From a programmer perspective, the interaction with the data store is fairly transparent and it is effectively manipulated as a natural extension of the C++ data space. However, orthogonal persistence is not supported due to the introduction of a number of features in the programming language. E presents a **new persistent type hierarchy** which mirrors that of C++, a new storage class (`persistent`), and a predefined generic type (`collection`).

Database types in E can reside either in volatile memory or in the store. Any data type that can be defined in C++ can also be defined as a database type. E provides a set of fundamental database types such as `dbint`, `dbchar`, and a group of database type constructors such as `dbclass` and `dbstruct` from which more complex persistent types can be created. If a database type instance is declared to be of `persistent` class, it will persist and its name will act as a handle for those objects in the persistent store.

Additionally, programmers can also make use of the database class `collection` to dynamically include objects into the persistent space. Objects in the collection are persistent as long as they remain in the collection and the collection exists. As garbage collection is not supported in E, programmers have to explicitly manage (i.e. allocate and deallocate) objects in the collection. Iterators were introduced in the programming language with the main purpose to support structured query processing. However, they later proved their usefulness as a general programming construct and were often used by programmers to operate over elements of a persistent collection.

The explicit distinction between the two groups of types, i.e. normal C++ types and E database types, is motivated by the optimisation of access to objects which are known not to be persistent. In this way, overheads are incurred just in the case in which persistent objects have to be managed. Although it would be possible to build an application relying solely on database types, the authors warn about the serious degradation in performance.

E's storage manager pins persistent objects before they are needed by a program [RC89]; it **avoids object faulting** and ensures that an object will be resident when needed. This feature is accomplished with direct support of the E compiler that generates code to pin and unpin objects in the storage manager's buffers [Ric90]. A pointer to an object of database type is composed of two values: a persistent identifier of the object and an offset within the object. The persistent identifier includes the physical address of the object while the offset allows embedded objects and arrays of persistent objects to be supported. Non-persistent instances of database types are addressed with a pointer of similar format. They have a special persistent identifier and an offset to the object's memory location.

E's memory management strategy presents a trade-off. Access to a persistent object is performed at normal computation time. However, accessing a resident object implies the object to be previously pinned. Thus, efficient access to resident objects is favoured in place of fast retrieval of persistent items. More recent implementations of the E language [SCD90] are based on the E Persistent Virtual Machine (EPVM) which serves as the interface between the compiler and the storage manager. It adds a cache to minimise the number of storage manager calls required and introduces a form of reference swizzling.

OPAL

The OPAL language [BMO$^+$89] is a derivative of the Smalltalk programming language and data model developed by Servio Logic Corporation. The language was initially designed to correct some of the evident deficiencies seen in Smalltalk [GR83] in the context of object-oriented database environments. From the features introduced next, it is possible to recognise how OPAL effectively brings together the worlds of databases and programming languages.

As a conventional programming language, OPAL can be used to create data definitions, to manipulate objects, and to execute general computations. As a database programming language, it supports transactional manipulation of the store and **associative data access**. From a programmer point of view, no explicit distinction is made in the language between volatile and persistent objects. The persistence model used in OPAL is based on reachability from a root object. Accordingly, any object in OPAL is potentially persistent with no distinction of its type. OPAL extends the Smalltalk heap into the persistent store; all types and instances persist indefinitely as long as a reference to them exists. A garbage collection technology is employed to manage the store.

To safely manipulate the database, programmers handle a snapshot of the database content; actions of other programmers are not visible in a given snapshot and they do not affect the execution of other users' programs. This functionality is supported by providing shadow copies of the objects in the database together with transactional concurrency control that may be varied between pessimistic and optimistic schemes according to objects' type and potential for contention.

In general, the data model presented by OPAL [Ser90] to programmers is very similar to the one offered by Smalltalk. However, some additions were done mainly to better support database features such as associative access. Associative access is supported by typing (i.e. class-kind constraints), path expressions, and a limited calculus sublanguage. The latter feature was designed to be seen as procedural OPAL code with little impedance between the query sublanguage and the OPAL programming language.

The built-in types *Set* and *Bag* support three kinds of data-structuring mechanisms in OPAL: collections, records, and arrays. Unordered collection classes have the standard semantics of set operations; named records keep a fixed number of fields which are addressed by name; and finally, arrays maintain a group of numerically indexed objects that can be inserted or deleted from the array. This contrasts with the initial design of Smalltalk which only supports arrays and records, and with conventional database languages which normally support only records and unordered collections.

OPAL is supported by GemStone [MSOP86, BOS91], a full-fledged object-oriented database management system that aggregates secondary storage management, concurrency control, authorisation, transactions, and recovery. GemStone manages individual instances of objects through *object-oriented pointers*, which act as unique surrogates for real objects. It uses an object table to map these pointers to physical locations. Similar to the Smalltalk's memory model, GemStone uses reachability to remove temporary objects before a transaction commits. Objects that have been created during a given transaction and cannot be accessed transitively from a root of persistence are temporary. Persistent objects are garbage collected using a mark-sweep strategy.

Texas/C++

Texas [SKW92] is a persistent storage system for C++ [ES90] developed at the University of Texas at Austin. Texas supports a limited form of orthogonal persistence[2] that enables applications to access transient and persistent objects in the same way; the types of transient and persistent objects are the same. Although, it enables a high degree of transparency in the use of the programming language, it defines a special interface to create persistent object instances.

This language interface **overloads the C++ `new` operator** to allow objects to be created in a persistent heap that corresponds to a specific persistent store [Tex96]. Programmers using Texas manipulate persistent data through named and unnamed persistent objects. Named objects are the entry points to the store; they reside in a root table and are associated with a name. Unnamed objects are those that do not need to be explicitly distinguished by name but can be referred by means of named objects (i.e. persistent roots).

The data store in Texas is built on top of a conventional file. The implementation strategy followed by Texas uses virtual memory protection fault trapping together with swizzling referred in the literature as **pointer swizzling at page fault time** [WK92]. Texas organises the data store as a collection of persistent pages in which objects are stored. Resident objects contain references to other objects in the form of virtual addresses; opposite to persistent objects which hold references in the form of physical addresses. When an attempt to access a non-resident object occurs, the page in which this object resides has to be memory mapped. This is done by handling virtual memory page faults and loading the requested page from the persistent store.

In addition to swizzling the reference contained in the requested object, Texas will swizzle *for free* all the references contained in the mapped page. This has the side effect of allocating further virtual memory pages for any non-resident persistent page to which such addresses refer. As a consequence, programs never

[2]There seems to be an inaccuracy in the literature on the precise use of the *orthogonal persistence* concept. Although a truly orthogonal system has to provide a form of object identification by reachability [AM95], authors also employ this term even when the Persistence Independence and Persistence Identification principles are slightly disregarded.

handle persistent addresses. All references will have been swizzled and will refer to either resident or protected pages [Kak98]. The location of addresses inside an object are computed and precisely identified using type descriptors which describe the layout of each class. Texas also includes a checkpointing facility that employs a write-ahead logging scheme in order to securely store the content of the persistent heap.

PJama

The PJama[3] platform [ADJ+96] is a collaboration between the University of Glasgow and Sun Microsystems. It implements an orthogonally persistent version of the Java programming language [AGH00]. Java was chosen due to specific features in the language such as strong typing, single-inheritance, automatic memory management, and an object-oriented model, as well as its increasing prominence and platform neutrality.

The PJama design stresses a minimum of interference with the standard Java programming language; PJama constitutes in this respect the closest implementation of **orthogonal persistence** to date. Values, whatever their type, have equal rights to persistence including all base types, objects constructed from that types, arrays, as well as classes and code. Persistence independence is enabled by keeping the language semantics the same regardless of the longevity of the data. Additionally, with respect to persistence identification PJama ensures that objects continue to naturally exist for as long as they are transitively reachable from a distinguished persistent root.

A PJama program usually executes together with an existing and populated store specified by a default, an environment variable, or a command-line parameter. Programmers can access data in a store by manipulating an instance of the class `PJavaStore`; it has a set of methods to manipulate the store and to register new objects, which in turn act as applications' roots of persistence. All the objects reachable from a persistent root are preserved and saved in the store. In this way, the lifetime of the graph of persistence is naturally extended beyond a given execution of a program. Objects in the graph of persistence can be manipulated with the semantics and operations of any standard Java object.

[3]Although initially named PJava, it changed its name to PJama after Sun trademarked PJava to denote Personal Java.

However, matching programming language semantics with the general view of orthogonal persistence creates some exceptions that have been pointed out in the implementation of the system. For example, to manipulate external state associated with native code, it is necessary to implement complex mechanisms to coordinate the closed world provided by the Java platform and the native state outside the Java environment. Furthermore, core features of the language such as the Abstract Window Toolkit (AWT) library heavily rely on native code. This creates a tension between the programming language and the completeness of orthogonal persistence provided by the implementation of PJama.

Another complex issue is that of reconciling the difference between the traditional transaction semantics and the concurrency control enforced by the programming language. The addition of a transaction model to PJama as used in the database arena requires significant changes to the Java language and may imply drastic changes to the current approach to concurrency control enforced by the programming language [Jor96, JA98].

PJama's persistence is mainly supported by an object cache and a buffer pool [AJDS96]. PJama uses a level of indirection to handle persistent objects. Objects in the object's cache are addressed via handles, which hold their PIDs, and grouped together in a resident object table (ROT). **Different swizzling strategies** are adopted according to the object type to be loaded; eager indirect swizzling is used for objects and lazy swizzling for arrays and methods [DA97]. This decision was taken because of the potential performance gains due mainly to the observed size of these objects.

The buffer pool is implemented on top of the Recoverable Virtual Memory tool (RVM) [SMK+93] using a no steal policy, i.e. memory pages that were modified by a transaction cannot be evicted from the buffer pool until the transaction commits or aborts. Objects are copied from the buffer pool to the object cache as they are faulted in and are copied back to the buffer pool when a stabilisation of the store occurs.

Objects are *promoted* as part of a stabilisation of the store; promotion is the process of creating a copy of an object on disk for the first time. It causes the allocation of handles in the ROT for the new objects. Although the design of a custom-made store layer for PJama has been published [PAD+97], no actual implementation has yet been reported.

2.5 Semistructured Data

Semistructured data, also called unstructured data, is commonly referred as *schemaless* or *self-describing*. These terms indicate that there is not a separate description of the type or structure of data, i.e. the schema is embedded with the data and no a priori structure is assumed. This situation contrasts with the traditional database approach in which it is necessary to know the data definition (i.e. type or schema) before its manipulation.

The are three main aspects of research [Bun97] in the area of semistructured data. First, the necessity to bring *new types of data* into the ambit of conventional database technology. This kind of data contains a degree of structure and data formats but cannot be constrained under a data schema due to its unmanaged nature. Probably the best known example of this sort of data is the World Wide Web (WWW) in which data type diversity is commonplace.

A second motivation is that of *data exchange and transformation*. The Tsimmis project pointed out the necessity to define a model of data exchange able to capture most kinds of data and provide a common substrate in which almost any other data structure could be represented [PGMW95]. It proposes the Object Exchange Model (OEM) as an internal data structure for exchange of data between database systems[4].

Finally, the third motivation is to be able to *query* database content without full knowledge of its schema; or even more, to be able to query database content and schema simultaneously. Although in the context of relational and object oriented databases, query languages including these features [KKS92] had been suggested, they fell short in providing the flexibility to express complex constraints on data paths, i.e. how to reach variables from the root or from any other node in a document.

Data Model

The data model associated with **semistructured data** is that of an *edge labelled graph* (see Figure 2.2); although the unifying idea in semistructured data is the representation of data as a tree structure. Semistructured data is represented by

[4]As history testifies, the temptation of querying data in its OEM format (e.g. XML) proved irresistible.

a collection of objects whose type can be atomic or complex. The value of an atomic object is of some base type such as integer, string, image, or sound.

The value of a complex object is a set of (`attribute`, `object`) pairs. An attribute is any string representing a name for the object. The data graph is then composed of nodes representing the objects and of edges labelled with attributes, and in which some leaf nodes have assigned an atomic value. The graph has a distinguishable object root from which all other objects are accessible [Suc98].

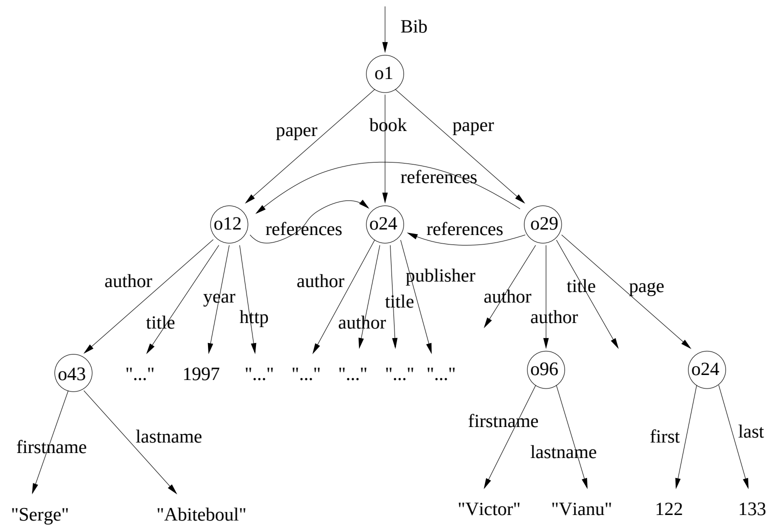


Figure 2.2: Edge-labelled graph representing semistructured data [Suc98].

This data representation model can be used to encode both relational and object oriented data in a fairly simply way [ABS00]. The notion of *object identity* is introduced into the model in order to be able to construct structures with references to other objects. In practice, the meaning of particular object identifiers is restricted to a certain domain. For example, for data loaded in memory, an object identifier can be seen as a pointer to the memory address corresponding to the referred node. For data stored on disk, an object identifier may be an address on disk for the node. In the context of data interchange in the WWW, an object identifier becomes part of a global namespace and has associated a Uniform Resource Identifier (URI).

Data Manipulation

XML (eXtensible Markup Language) [BPS$^+$04] is the standard adopted by the World Wide Web Consortium (W3C) to complement HTML (Hypertext Markup Language) for data exchange on the Web. The basic XML syntax is well suited to describe semistructured data. As SQL is the dominant language for querying relational data, XML is becoming the dominant format for manipulating and exchanging semistructured data; although its initial roots are as a document markup language. A general perspective of query languages for semistructured data is analysed next and then linked with the development and application of these concepts into the XML context.

Although there have been many proposals for XML query languages such as XML-QL [DFF$^+$99], XMAS [BCG$^+$99], XQL [RLS98], XDuce [HP01], and Quilt [CRF01], two of the most influential paradigms in semistructured query languages are found in Lorel and UnQL. Lorel [AQM$^+$97] is a query language developed as part of LORE (Lightweight Object REpository), a general purpose data management system. A query in Lorel consists of two parts, a *pattern* used to extract bindings for a set of variables and a *construct* clause indicating how to build the answer from the set of bindings found through the given pattern. The pattern indicates how to reach the variables from the root or from each other node by regular path expressions.

In Lorel, the depth of nesting of the construct clause is statically determined by the syntax of the query. In contrast, UnQL [BDHS96] (Unstructured Query Language) overcomes this limitation by using *structural recursion*, a construct for programming with sets, bags, and lists [BNTW95]. For semistructured data, this mechanism is particularly attractive as it enables the user to express both queries and transformations using the same formalisms. In this approach, queries simply return a subset of nodes from the input data while transformations may construct a new graph. A more detailed analysis of the query languages for semistructured data and XML is beyond the scope of this work; a comprehensive study on the topic can be found at [Abi97].

It is possible to trace direct influence of these two main paradigms on the implementation of different W3C proposals and data manipulation tools for semistructured data represented as XML documents. XQuery [BCF$^+$05], a proposed standard for querying XML data, was particularly influenced by Quilt which is a combination of features seen in XML-QL, a descendant of Lorel, and

XQL. Queries written in XQuery look similar to SQL queries and are organised into FLWR expressions comprising four sections: for, let, where, and return.

XSLT [Cla99] was designed as part of the XSL style sheet system to control the formatting of XML data into HTML or its transformation to a different XML structure. It employs the concept of structural recursion introduced by UnQL and can also be used as a query language. It employs the XPath [CD99] W3C standard to create path expressions to address parts of an XML document.

The paraphernalia around XML extends to many other aspects. Different proposals to document schemas have been used to overcome the limitations of the self-describing nature of the pure semistructured data model such as storage inefficiency and difficulties in query evaluation and formulation. The Document Type Definition (DTD) serves mainly as a grammar for the underlying XML document. However, it proved to lack expressive power and flexibility in many situations as it only supports limited type definitions. Thus, the XMLSchema [FW04] constitutes a more sophisticated schema language which adds types, enables the creation of user-defined types, allows uniqueness and foreign keys constraints, and adds a form of inheritance. These are just the main features added to the functionality provided by DTDs.

The XLink language [DMOW05] (XML Linking Language) allows elements to be inserted into XML documents in order to create and describe links between web resources. XPointer [GMMW03] is an extension of XPath which is used by XLink to locate link resources on the web. The Resource Descriptor Framework (RDF) is used for representing metadata in XML documents. The XML standards and tools mentioned before are only the most common and they are not representative of the whole activity done by the W3C in the XML context.

There are two standard models for programmatic manipulation of XML data, each currently available in many programming languages. The Document Object Model (DOM) [HHW$^+$04] presents an object oriented tree view of XML documents. Each element in the XML document takes the form of a node with a well-defined interface. Programmers can access specific parts of the document in a navigational way.

For example, the Java DOM API provides a `Node` interface which defines methods such as `getParentNode()` or `getFirstChild()`. Subelements of an element can be accessed by name using `getElementsByTagName(name)` which returns a list of child elements with the given name. In turn, elements in the list can be accessed using the method `item(i)` which returns the ith element in the list.

Elements' attributes can be obtained by using the method `getAttribute(name)`. Additionally, the DOM API provides a number of methods to update and create documents by setting node values, or adding and deleting attributes and elements.

The other dominant programming interface for XML documents is the Simple API for XML (SAX) [SAX05]. It presents an event model aimed to provide a common interface between XML parsers and applications (much in the way of JDBC or ODBC for relational databases). Programmers using the SAX API have to define event handlers to associate functions with parsing events. Parsing events correspond to the identification of specific parts of an XML document. SAX works at a lower level than DOM, and thus it is more efficient and gives more control to the programmer. However, it requires more work than DOM even for the simplest of the tasks. An important feature of SAX, that may be a considerable disadvantage in several scenarios, is that it processes XML documents sequentially and therefore backward navigation is not possible. The only workaround is to store processed information in appropriate data structures.

Implementation and Systems

There are many approaches and strategies for storing XML data. Since XML is a file format, a common storage medium for XML documents may be a file. A range of tools provides support to access and query XML data with relative ease. Currently, there are three prevalent ways to store and access XML documents in a database environment.

The first approach exploits the infrastructure provided by relational database systems [DFS99, KR01, SGT+99]. In general, converting XML data to its relational form is a simple task, specially if the XML data was generated from a relational database and it provides a schema. However, relational technology proves inefficient if the data mappings produce many relations or if queries require complex XML constructs which translate into many SQL queries. Another major drawback is that even simple XML queries can lead to complex query patterns that require multiple database joins to be evaluated.

The second approach uses native XML database engines. It offers as its key advantage a more natural data model which is typically mapped to graphical or hierarchical data representations [KM00, Sch01, JAKC+02, PAKC+03]. Finally, object oriented technology has also been suggested as a suitable medium to manage XML data [SYU99, RP02]. It provides a more expressive type system than that used in relational databases and represents a more efficient mapping for XML data. However, this option has not gained popularity in part because commercially available database support for XML is provided by dominant software vendors supporting relational technology [BKKM00, CX00, Rys01].

2.6 Discussion

The appearance of different storage paradigms has been an evolutionary process guided mainly by well-defined application requirements. Figure 2.3 positions particular data storage paradigms according to their intrinsic ability to manipulate programming abstractions against their data querying development. For simplicity, it ignores databases supporting the hierarchical and network data models; the figure illustrates these omissions with a double crossed arrow.

Before the development of the first file systems [DN65], programmers used to spend considerable programming time and effort managing persistent application data in the whole hierarchy of secondary storage. The file system API removed authentic programming hassles associated with this storage model. It isolates programmers from explicitly manipulating the secondary storage hierarchy and the physical addressing of persistent data in storage devices. The most important contributions of the file system API are to present a simple abstraction (i.e. a file) and to provide a standardised set of IO routines to access persistent data. It relieves programmers from working on special-purpose pieces of code for each application and from manipulating data directly on top of different storage devices.

Data-intensive application environments with a highly structured set of information that needs to be accessed by a large number of users with dynamic query requirements motivated the creation of database systems. This technology consolidated with the introduction of the relational data model [Cod70] whose manipulation was to be mainly done through a **high-level non-procedural language**. The level of isolation between internal data representations and programming abstractions that a high-level non-procedural database language enables has proved extremely effective in the relational database application domain.

The development of persistent programming languages shows how to augment languages in order to give programmers full command of applications' persistent

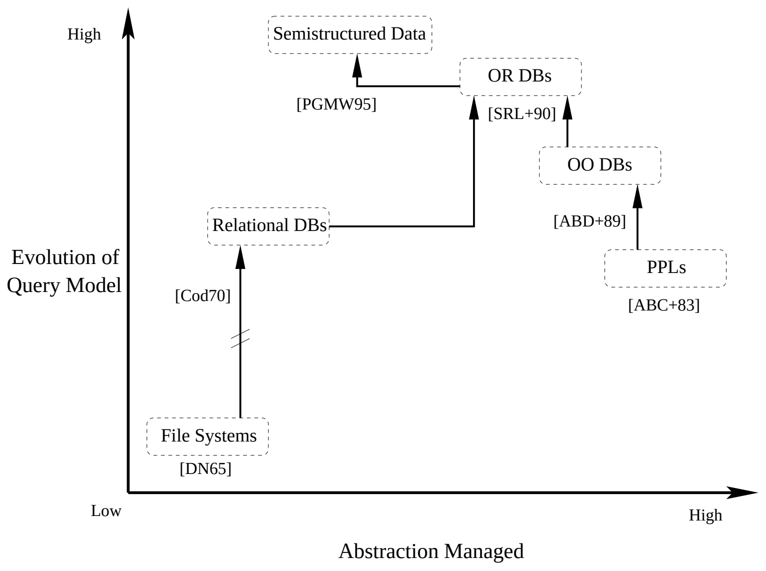


Figure 2.3: Development of the main data storage paradigms based on their intrinsic capacity to manage programming abstractions versus the evolution of their query models.

data abstractions through features in the host programming language. The most prominent idea of this model is that of fully replacing the mapping between persistent and volatile abstractions by coalescing them at run time and thus totally eliminating the **impedance mismatch** associated with data longevity.

The **data modeling power** available in the object oriented paradigm, together with concepts taken from PPLs, influenced the database community. Object oriented [ABD$^+$89] and object relational databases [SRL$^+$90] group features of database systems with the manipulation of more powerful persistent abstractions to attack a set of applications that the limited tabular data model fails to satisfy. Finally, **semistructured data** is proposed mainly as a common data exchange format and as a mechanism to query content and structure in banks of data in the absence of a known database schema [PGMW95].

In this evolution of data storage models, the constant addition of higher levels of abstraction in programmatic interfaces has been a driving force. Higher levels of abstraction in storage systems' APIs enable direct benefits not only for the operation of a system itself but for programmers who are provided with better tools to solve problems. In turn, more suitable data storage APIs assist in the creation of software of better quality with less effort.

The impact that data storage programmatic tools have on programmers' duties is evident if the typical file system API is compared with, for instance, the facilities offered by an orthogonally persistent object system. In the first case, programmers have to spend considerable amounts of effort in the tedious and error prone task of mapping application data to a flat file model while in the second case there is a transparent transition between persistent and volatile data abstractions. But more importantly, application data semantics are explicitly manipulated, exposed, and preserved regardless of its longevity. This finally, provides important advantages for programmers.

The persistence model presented by file systems provides simplicity and generality. It is fitted for applications looking for efficient streamed data manipulation; the APIs introduce minimal overheads but, in exchange, rely heavily on developers writing code for every new application they create.

Although in many situations it would be ideal to enrich file systems' data manipulation with higher levels of abstraction, richer storage systems' programmatic tools also result in the **addition of overheads**. For example, in the database context an intermediate language is used to manipulate persistent abstractions (e.g. SQL, ODL, OQL, XQuery) together with all the necessary infrastructure to make them work properly (e.g. query optimisation and execution, programming language bindings, and mappings).

Databases also provide data protection and data integrity services through transactional and access control frameworks, as well as mechanisms to secure data in case of media or system crashes. Management of rich data models also introduce additional overheads and complexity into storage systems' design.

The powerful capacity that persistent programming languages have to transparently manipulate applications' data independently of its longevity has many advantages for programmers. However, type support is confined to specific language compilers and restricts developers to particular programming languages.

Migrating file system applications to any of the available data storage models is not a good option because they have a completely different set of data access requirements in which the aforementioned overheads and compromises can be reduced or completely eliminated. Moreover, they present APIs that are inefficient, rely heavily on programmers properly managing persistent abstractions, or reproduce many of the problems that the impedance mismatch of data models introduce.

The **trade-off** between IO efficiency, a feature so valued in file system designs, and programmability of many applications that run on top of file systems may well be unbalanced according to the current state of affairs. Hence, the core motivation of this dissertation is to augment the abstraction exposed by APIs that deal with persistent data that exhibit file-like data access patterns.

Accordingly, Chapter 3 of this dissertation presents a practical study of file systems content. It analyses a set of techniques that may be used to extract internal file data organisations and structure in a systematic way using solely the fundamental file IO API. Subsequently, Chapters 4, 5, and 6 are devoted to the design, implementation, and evaluation of a storage system API that deliberately exposes data structures and type information specially targeted to assist file system applications whose persistent data is rich in abstractions.

Chapter 3

Identifying Portions of File Content

In this chapter I present a detailed study of internal file content [PP04]. A fundamental consideration is that redundancy of portions of data at a sub-file level might well suggest internal file data organisations and structure. Thus, I investigate to what extent it is possible to identify specific portions of data in considerably large data sets following a systematic approach.

In Section 3.1 of this chapter I discuss the motivation for the study of internal file content. After that, in Section 3.2, I introduce the techniques that have been employed in different works with the purpose of finding data redundancy. Next, I explain the methodology followed in this study and the experimental results obtained in Sections 3.3 and 3.4, respectively. Finally, I discuss the key findings of this analysis in the global context of this dissertation in Section 3.5.

3.1 Tracing Internal File Structure

Files represent the most common persistent abstraction used by applications to store data. The vast number of tools, programming languages, and applications using the services offered by file systems creates a great amount of diversity in file-system contents [DB99]. Practically, every programming language offers bindings to the basic file API. However, regardless of the capacity that data processing tools offer for creating **application abstractions** at run-time, when data has to be saved, abstractions are flattened in a file. This observation implies that at a file level there is an important amount of structure and data type that is hidden and unrecoverable unless application-specific libraries are used.

On the other hand, some applications may generate versions of a document stored as separate files, but whose content differs only slightly. For example, software development teams using a revision control system might open a number of source code files to change a group of methods and create an up-to-date software distribution; both the new source code and the compiled output files will share portions of **identical information** [MCM01]. These files will also present *localised* differences corresponding to the modified portions of data.

This situation repeats in several application contexts. In general, application programs open files to update specific portions of data embodied in run-time abstractions (e.g. objects, structures, or records) and then send the new version of the file to disk. Thus, computer systems frequently store and manipulate several copies of the same information. File synchronisers [BP98, Tri00, PV04], backup systems [QD02], reference data managers [DH03], and peer to peer storages systems [CMN02, GRR$^+$98, MPH02, SAS$^+$96] report considerable amounts of identical data.

In an attempt to exploit the advantages obtained by gaining access to the internal composition of file content, plenty of systems break information into discernible blocks of data. The recurring issue is that a block-level duplicate detection study provides hints about internal file organisation that can be exploited in different ways. For example, email data in real-world data sets has been studied by Denehy and Hsu [DH03]. They worked on diverse storage formats such as Lotus Notes NSF, mbox, and mh. In brief, they point out that email data has a recognisable structure made of header, body, and attachments which is constantly repeated over the data sets. Their experimental results show that increased levels of redundancy are found with those techniques that break file data into parts. Of particular interest is that many attachments are similar in terms of the data blocks they are made of, but not identical. This suggests that a certain degree of structure in file data can be characterised by analysing sub-file redundancy patterns.

In a different context, Broder proposed the concepts of resemblance and containment to determine whether two web documents are near-duplicate or contained [Bro97]. In this work, vectors of shingles of a large number of web pages are computed and used to measure the resemblance among them. A vector of shingles can be interpreted as the structural sketch and summary of a larger document. Ultimately, overlapping shingles of two documents correspond to overlapping sections of content in the web pages. This work has been extended and applied to different types of data as is indicated in Section 3.2.

Mogul et al. [MCK04] and Rhea et al. [RLB03] describe methods for computing checksums over real-world web resources (e.g. HTML pages) in order to eliminate retrieval of identical portions of data, even when these resources have been aliased or rotated. These examples indicate that, where a simple comparison of file name is impractical, more sophisticated data access strategies can be designed if a *deeper* representation of file content is used.

A detailed analysis of data redundancy patterns in practical data sets has broader implications and is also useful in other contexts. For example, efficient data management solutions may be created if system designers are aware of the amount of redundant data seen in diverse data sets. Although saving disk space can be useful, over the past few years there has been a constant reduction in the cost of raw disk storage [MT03] and some may argue that the disk space savings obtained by suppressing identical data are of minimal significance.

However, storage systems may exploit data duplication patterns to *optimise the use of storage space and bandwidth.* Single Instance Storage (SIS) [BCGD00] explores the content of whole files to implement links with semantics of copies instead of storing a file with the same content several times. Backup systems such as Venti [QD02] store duplicated copies of fixed size data blocks only once. LBFS [MCM01], Pasta [Mor02, MPH02], Pastiche [CMN02], and the Value-Based Web Caching algorithm (VBWC) [RLB03] find identical portions of data using Rabin fingerprints. In this method, data is divided into content-defined chunks in order to exploit cross-file data duplication. Additional details of the content-defined chunking method will be presented in Section 3.3.

File systems may obtain improved *caching performance* if they are aware of contents shared between files. In this way, it would be possible to provide better hit ratios for a given cache size. A potential size reduction of the main memory file cache may have important performance effects. In mobile environments, devices are often limited in storage and bandwidth. Furthermore, there are factors such as *energy consumption and network costs* associated with data transmission that can become critical [BA03, MCM01]. Under certain circumstances, it might be desirable to perform significant computation to reduce the number of bits transmitted over low-bandwidth or congested links.

The recurring issue of all the works previously mentioned is that it is feasible to unveil a degree of internal file structure through a data redundancy analysis. Therefore, instead of using application specific libraries to unveil this structure, they inspect file content in an attempt to identify portions of data that are repeatedly manipulated by applications.

The study presented in the rest of this chapter considers whether it is also feasible to disclose, in a systematic way, a degree of structure in file-system contents by taking into account two factors. Firstly, that files certainly retain applications' *data abstractions* which mirror update patterns at the application level, and secondly, that there is a considerable amount of *redundancy* in storage systems data. Hence, overlapping sections of file content might suggest internal file organisation and, probably, meaningful data types.

Therefore, this study reports the evaluation of three elementary methods used to find identical sections of data: whole file content hashing, fixed size blocking, and a chunking strategy that uses Rabin fingerprints [Rab81] to delimit content-defined portions of data. These techniques have been frequently employed in different systems and used as building blocks of more sophisticated strategies to spot data similarity as pointed out in Section 3.2.

The goal of this study is twofold. First, it presents a head-to-head evaluation of three elementary techniques used to disclose redundancy patterns of practical data sets, and expose their trade-offs; due to the lack of a practical comparative study, the typical performance of each method and their suitability for different data profiles were not clear before this study. Moreover, it assesses up to what extent these techniques are generally applicable to trace internal file data organisations based solely on the study of data redundancy patterns.

3.2 Related Work

A number of strategies to discover similar data in files have been explored in different systems. Unix tools such as `diff` and `patch` can be used to find differences between two files and to transform one file into the other. Rsync [Tri00] copies a directory tree over the network into another directory tree containing similar files. It saves bandwidth by finding similarities between files that are stored under the same name.

The Rabin fingerprinting algorithm [Rab81] has been employed with different purposes such as fingerprinting of binary trees and directed acyclic graphs [Bro93, KR81], or as a tool to discover repetitions in strings [Rab85]. However, this study examines how Rabin fingerprints can be used to identify identical portions of data in storage systems. In general, Rabin fingerprints have been used for this purpose in two ways: to sample files in order to discover near-duplicate documents in a large collection of files, or to create content-defined chunks of identical data.

Manber [Man94] employs Rabin fingerprints to sample data in order to find similar files. His technique computes fingerprints of all possible substrings of a certain length in a file and chooses a subset of these fingerprints based on their values; the selected fingerprints provide a compact representation of a file that is then used to compare against other fingerprinted files. Similarly, Broder applies resemblance detection [Bro97] to web pages [Bro00] in order to identify and filter near-duplicate documents. Rabin fingerprints of a sliding window are computed to efficiently create a vector of shingles of a given web page. Consequently, instead of comparing entire documents, shingle vectors are used to measure the resemblance of documents in a large collection of web pages. The techniques used by Manber and Broder have been adapted by Spring and Wetherall [SW00] to eliminate redundant network traffic. However, they used Rabin fingerprints as pointers into data streams to find regions of overlapping content before and after the fingerprinted regions.

Different systems use blocking strategies that employ the Rabin fingerprinting algorithm to create *content-defined* and *variable-sized* data chunks. Probably the first storage system that used Rabin fingerprints for this purpose was LBFS [MCM01], specially designed to transmit data over low-bandwidth networks. Using the Rabin fingerprinting algorithm, LBFS finds similarities between files or versions of the same file. It avoids retransmission of identical chunks of data by using valid data chunks contained in the client's cache and by transmitting to and from the data server only the chunks that have been modified. LBFS's chunking algorithm was tested on a data set of 354 MB reporting that around 20% of the data was contained in shared chunks.

Blocking in Pasta [Mor02, MPH02], an experimental peer to peer file system, also exploits the benefits of common information between files. Caching and replica placement are defined by data blocks' content. These blocks are built by computing Rabin fingerprints of the file data over a sliding window. Identical blocks are stored only once and referenced using a shared key. A similar technique is used in Pastiche [CMN02]. In Value-Based Web Caching [RLB03], web proxies index data according to their content and avoid retransmission of redundant data to clients connected over low-bandwidth links. Although all these systems show that improved block sharing levels can be obtained using content-defined chunking strategies, they do not present broad experimental results based on diverse data sets.

Data redundancy in storage systems has also been identified using *fixed size blocking strategies*. Sapuntzakis et al. aimed to reduce the amount of data sent over the network by identifying identical portions of data in memory [SCP+02]. They use a hash-based compression strategy of memory aligned pages (i.e. fixed size blocks of data) to accelerate data transfer over low-bandwidth links and improve memory performance.

Venti [QD02], a network storage system intended for archival data, aims to reduce the consumption of storage space. It stores duplicated copies of fixed size data blocks only once. Venti reports a reduction of around 30% in the size of the data sets employing this method. Future implementations of Venti may also incorporate a content-based blocking scheme based on Rabin fingerprints.

A different approach to eliminate data redundancy is used in SIS [BCGD00] for Windows 2000. It saves space on disk and in main memory cache but with a different approach; SIS explores the content of the *whole file* and implements links with the semantics of copies for identical files stored on a Windows 2000 NTFS volume. A user level service, called the *groveler*, is responsible for automatically finding identical files, tracking changes to the file system, and maintaining a database with the corresponding file indexes. When SIS was tested on a server with 20 different images of Windows NT the overall space saving was 58%.

Although many systems have proposed different techniques to manage duplicated data, it has been only lately that practical studies to assess their benefits and applicability have been performed. Building on Manber's observations, Douglis and Iyengar [DI03] explore duplication in empirical data sets using Delta-Encoding via Resemblance Detection (DERD) and quantify their potential benefits. Their technique generalises the applicability of delta-encoding by choosing an appropriate set of base versions in a large collection of files through resemblance detection.

As part of the design of a storage system specially crafted to manage reference data [DH03], a comparison on the effectiveness of three duplicate suppression techniques has been done; two of these techniques are similar to the methods examined in this document: fixed size blocking and the content-defined chunking algorithm. The third technique, called *sliding blocking*, uses rsync checksums and a block-sized sliding window to calculate the checksum of every overlapping block-sized segment of a file. The sliding blocking technique consistently detected greater amounts of redundant data than the other two strategies.

More recently, Redundancy Elimination at the Block Level (REBL) has been proposed as an efficient and scalable mechanism to suppress duplicated blocks in large collections of files [KDLT04]. REBL combines features of techniques such as content-defined data chunks, compression, delta-encoding, and resemblance detection. Empirical data sets were used to compare REBL with other techniques. REBL presented the smallest encoding size in 3 out of the 5 data sets analysed and consistently performed better than the other techniques.

Specifically, the effectiveness of this technique compared to the content-defined chunking strategy varied by factors of 1.03-6.67, whole file compression by 1.28-14.25, sliding blocks [KDLT04] by 1.18-2.56, object compression (tar.gz) by 0.59-2.46, and DERD by 0.88-2.91. All these studies [DH03, DI03, KDLT04] used Rabin fingerprints as a tool to eliminate data redundancy. Thus, their experimental results relate highly to this work.

3.3 Methodology

Three methods were used in this work in order to spot identical sections of data in large collections of files: whole file content, fixed size blocks, and Rabin fingerprints. In this section, a detailed description of the programs that implement each of these techniques is presented. When these programs are run on practical data, they gather information in order to characterise file system content and exhibit data duplication on different data profiles (see Section 3.3.1).

Whole file hashing

Similarity patterns at a whole file granularity are identified by calculating the SHA-1 digest [SHA95] of individual files as shown in Figure 3.1. The first 64 bits of the resulting digest are used as the key to a hash table; identical files are guaranteed to be indexed using the same hash table entry. This hash table is used to store statistical data of identical files such as number of occurrences and size of the original file. Although Henson [Hen03] warns about some of the dangers of comparing by hash digests, the highly unlikely event of false sharing of the key space is not a crucial concern in this study[1].

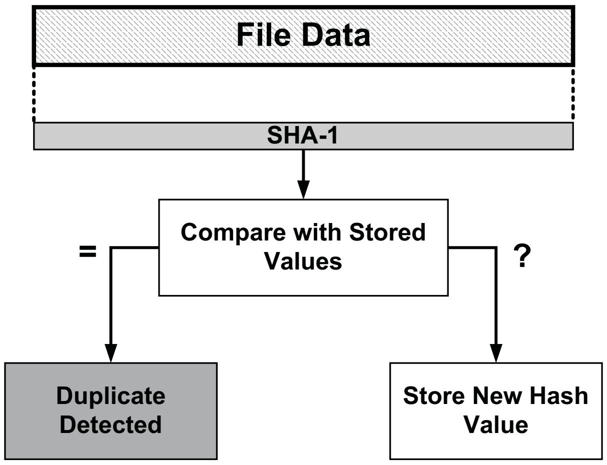


Figure 3.1: Whole file content analysis.

Fixed size data blocks

In order to find similarities using fixed size blocks of data an analogous procedure is employed, but instead of obtaining digests for whole files, they are calculated for contiguous non-overlapping fixed size portions of the files as can be seen in Figure 3.2. In this case, hash table entries correspond to unique data blocks.

Content-defined data chunks

The third method analysed employs Rabin fingerprints; it offers the advantage that the chunks generated are defined according to their contents. The mathematical principles of Rabin fingerprints are well documented [Rab81]. A Rabin fingerprint $f(A)$ is the polynomial representation of some data $A(t)$ modulus an irreducible polynomial $P(t)$. Such an irreducible polynomial is computed only once and used in all the experiments in order to find identical pieces of data. The algorithm for computing such a polynomial can be found in [CL01]. The

[1]The maximum number of blocks obtained for any single data set in all of the experiments was $n \approx 18\text{x}10^6$. These blocks were indexed using the first $b = 64$ bits of their SHA digests. The probability of having one or more collisions is given by $1 - (1 - 2^{-b})^n$. This small probability can be neglected in the experimental results.

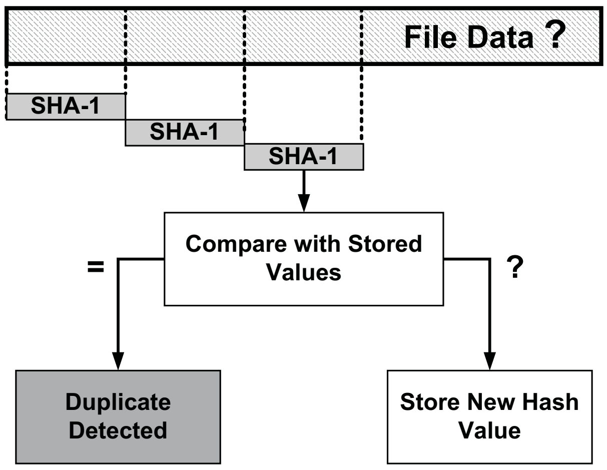


Figure 3.2: Analysis of fixed size blocks of data.

implementation of the Rabin fingerprinting algorithm used in this study follows the principles presented by Broder in [Bro93], which is an extension of the work done by Rabin. Broder uses precomputed tables to process more than 1 bit at a time, particularly, 32 bits are divided into four bytes and processed in one iteration. The value of k, which is the degree of the irreducible polynomial and consequently the length of the fingerprint, should be a multiple of 32.

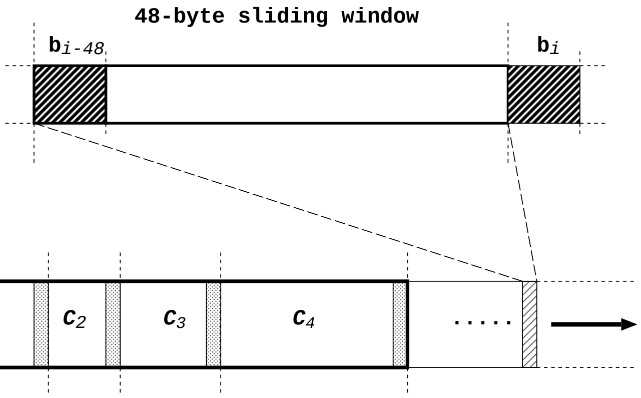
As shown in Figure 3.3, to divide a file into content-defined chunks of data, the program incrementally analyses a given file using a sliding window and marks boundaries according to the Rabin fingerprints obtained in this process. It inspects every w bytes of a sliding window that is shifted over the contents of the file. Although the value of w can be tuned, changing the size of the sliding window does not significantly impact the result. Experimental evidence in [Mor02, MCM01] shows that by setting $w = 48$ bytes it is possible to discover significant levels of data duplication.

Figure 3.3(a) shows that adding a new byte into the sliding window is accomplished in two parts. First, the value for the oldest byte in the window is subtracted from the fingerprint. Second, the terms of the new byte are added to the fingerprint. This is possible since the fingerprint is distributive over addition. When subtracting, precomputed tables are used in order to improve the implementation performance.

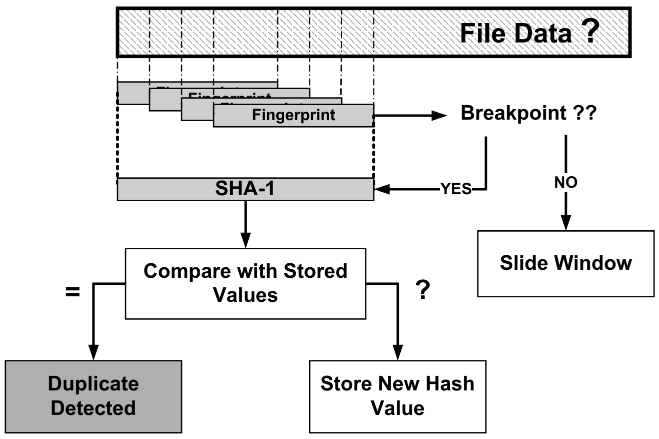
Rabin fingerprints for each window frame are calculated and if the value obtained matches the r least significant bits of a constant, a breakpoint is marked.

These breakpoints are used to indicate chunk boundaries. In order to avoid pathological cases (i.e. many small blocks or enormous blocks) the program forces a minimum and a maximum block size.

In Figure 3.3(a), shaded boxes represent the 48-byte regions that generated a boundary. The light striped rectangle corresponds to the current 48-byte window. At each step the byte in the oldest position of the sliding window (b_{i-48}) is subtracted from the fingerprint and the next byte in the file (b_i) is added to the fingerprint.



(a) Computing Rabin fingerprints over a sliding window.



(b) Analysis of content-defined chunks of file data.

Figure 3.3: Chunking file data using Rabin fingerprints.

As shown in Figure 3.3(b), once a boundary has been set, the SHA-1 digest corresponding to the chunk's content is calculated. Similar to the other two techniques, the first 64 bits of the SHA-1 digest are used as the key for accessing the hash table that stores statistical information about identical data chunks.

Results reported

Two kind of values are calculated and reported by the evaluation programs: percentage of identical data and storage savings. To calculate the **percentage of identical data** in shared blocks, the programs add for each duplicated block, the product of its size and its number of occurrences. They report this value as a percentage of the original data set size. To calculate **storage space savings**, the sizes of every unique block in the hash table are added; replicated blocks are counted only once. This value is presented as a percentage of the original data set size. Additionally, storage space savings are compared with the space used by simply **tarring and compressing** the whole collection of files in the different data sets. The standard `tar` and `gzip` utilities were used for this purpose.

3.3.1 The Data Sets

Different collections of real-world data sets were explored to determine how sensitive each of the methods are to diverse data profiles. These groups were:

- **Mirrored section of sunsite.org.uk**[2]. This data is a subset of an Internet archive and its size is over 35 GB. Compressed and packed data were common in this data set.
- **Users' personal files.** The data analysed in this collection of files is held in 44 home directories of different users in the Cambridge University Computer Laboratory. The size of this data set is approximately 2.9 GB.
- **Research groups' files.** This data set contains collections of files associated with different research projects of the Computer Laboratory. This is a data set with a potentially high level of data duplication because it stores software development projects, shared documents, and information accessed and manipulated by groups of people. The size of this data set was 21 GB.

[2]ftp://sunsite.org.uk

- **Scratch directories.** The 100 GB of information were taken from the Computer Laboratory scratch space and represents the largest and most diverse data set analysed. This collection of files is a good example of a data set where no obvious interrelationship is previously known.

- **Software distributions.** To explore the sharing patterns of highly correlated data in different states, five successive Linux kernel distributions in three different formats were studied: packed and compressed (.tar.gz), uncompressed but still tarred (.tar), and uncompressed and untarred.

3.4 Experimental Results

The experimental results presented in this section characterise file system content at different levels of granularity and in terms of the amount of redundancy that is possible to disclose following a systematic approach. These results are used to evaluate the benefits and performance of the three different duplicate detection methods in each of the aforementioned data sets.

3.4.1 Mirror of sunsite.org.uk

In order to find commonality in data that resembles a standard Internet archive, the programs were run on a 35 GB section of sunsite.org.uk. The total number of files in the data set was 79,551, with an average file size of 464 KB. Table 3.1 shows a partial characterisation of this data set. It shows the 15 most popular file-name extensions and their percentage of the total number of files. The 15 most popular file extensions account for over 82% of all files. Table 3.1 also indicates the 15 file extensions that use the most storage space and the percentage of the total space they consume; collectively, they cover almost 97% of the whole data set. Packed and compressed files (e.g. rpm, gz, bz2, zip, and Z) represent an important part of the data set (around 24.4 GB). A detailed analysis of compressed files is presented in section 3.4.1.1.

The content-defined chunking method was run over the data using different expected chunk sizes and sliding window lengths. Although in all the subsequent experiments the maximum chunk size permitted was set to 64 KB, the minimum chunk size was fixed to 1/4 of the expected chunk size; the expected chunk size is set as a parameter in the algorithm. By fixing the maximum chunk size to

Rank	**Popularity**		**Storage Space**	
	Ext.	% Occur.	Ext.	% Storage
1	.gz	32.50	.rpm	29.30
2	.rpm	10.60	.gz	20.95
3	.jpg	7.54	.iso	20.40
4	.html	4.83	.bz2	6.26
5	.gif	4.43	.tbz2	5.65
6	–	4.16	.raw	4.44
7	.lsm	3.74	.tgz	2.66
8	.tgz	2.90	.zip	2.53
9	.tbz2	2.35	.bin	2.00
10	.Z	2.12	.jpg	0.94
11	.asc	1.84	.Z	0.65
12	.zip	1.59	.gif	0.43
13	.rdf	1.39	.tif	0.31
14	.htm	1.21	.img	0.21
15	.o	1.06	.au	0.19
Total	——	82.26	——	96.92

Table 3.1: Data profile of a 35 GB mirrored section of sunsite.org.uk. Files without extension are denoted by the – symbol.

64 KB despite changes in the expected chunk size, it is possible to maximise the opportunities of finding identical portions of data in larger chunks. Minimum and maximum chunk lengths are enforced to avoid pathological cases such as very small or large chunks. Table 3.2 shows the amount of information in shared chunks with two different window sizes and three different expected chunk sizes. The last column of Table 3.2 illustrates the percentage of identical data that was found using fixed size blocks.

Table 3.2 suggests that the window size does not significantly influence the commonality levels; other studies [Mor02, MCM01] report similar findings. Therefore, in all subsequent experiments the window size was fixed to 48 bytes. It is also possible to observe that when the expected size of the chunks is shorter, the percentage of shared data is larger because the probability of finding similar chunks increases. However, the expected size of the blocks represents a trade-off between the size of the hash table needed to maintain a larger number of entries for each unique chunk and the potential storage space savings (see section 3.4.6).

The percentage of identical data in whole files was only 5%. Therefore, it was possible to find a considerable amount of similar data in partially modified files.

On the other hand, the percentages of identical data found using the content-defined chunking method presented only slight differences when compared with those obtained using fixed size blocks. These findings suggest that a storage system handling this kind of data could easily select a fixed size blocking scheme without losing significant storage space savings.

	% of Data in Identical Blocks		
Size	48 B window	24 B window	Fixed Size
8 KB	16.43	16.12	12.13
4 KB	20.27	19.72	17.25
2 KB	25.00	24.18	22.23

Table 3.2: Percentages of identical data in a 35 GB section of sunsite.org.uk.

Figure 3.4 shows the distribution of block sizes under the content-defined chunking method when the expected chunk size was set to 8 KB. In particular, an average block size of 9.2 KB was obtained. The algorithm generated 3,730,576 blocks of which 7.6% have at least one identical copy. It is also possible to appreciate the impact of pathological cases on the distribution: the chunks under the minimum block size (2 KB) correspond to files that are shorter than the minimum permitted or to final portions of files. Moreover, the peak at 64 KB corresponds to chunks inserted because of the maximum size allowed.

Special attention was given to the set of files that accounted for 97% of the total storage space; therefore, the levels of similarity of these 15 kinds of files were explored. The expected chunk size was fixed to a value of 4 KB. Table 3.3 shows the results and compares them against the percentage obtained with the whole file approach. Remarkable patterns were found in the results. Firstly, it is very difficult to exploit similarity in compressed files; practically all the duplicated chunks were contained in identical files. The behaviour of compressed data will be further investigated in Section 3.4.1.1.

Secondly, .iso and .img files presented the highest difference in percentage of similarity against the whole file column. These findings suggest that variations between files can be efficiently isolated using the content-defined chunking method, whereas under the whole file approach, even a small change in the file leads to storing a new almost-identical file. The negative effects of this situation are intensified in large files such as ISO image files in which the average size was 280 MB. All other files showed only slight increments if they are compared against the value obtained for the whole file scheme.

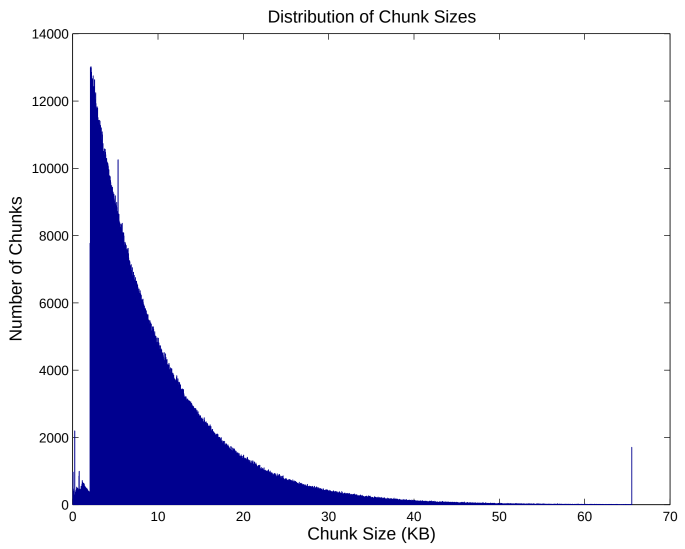


Figure 3.4: Distribution of chunk sizes obtained from sunsite.org.uk using an expected chunk size of 8 KB and a 48-byte sliding window size.

Keeping only one copy of the information saves storage space. Using the best scenario, which was the content-defined chunking method with an expected chunk size of 2 KB, the experimental results indicate that a file system would store only 30 GB of unique data instead of the original 35 GB, representing around 14% of storage savings. Duplicate suppression proved to be somewhat more efficient in saving storage space than the tar-compressed version of the data set; the tar.gz file for this data set claimed around 33.3 GB of disk space.

An explanation for these numbers may be found in the detailed analysis of the files that comprise this data set. As has been pointed out before, a large amount of data is already compressed (see table 3.1); approximately 24.4 GB correspond to rpm, gzip, bz2, zip, and Z files. Note that archives in rpm files are compressed using gzip's deflation method. To a certain extent, redundant data in these files has already been removed as part of the LZ77 [ZL77] compression technique. Compressing compressed data with the same algorithm normally results in more data, not less.

However, it may still be possible to argue that if the content-defined chunking method was able to remove duplicated chunks of data from the data set, the compressed version of the data would do it as well, resulting in a smaller tar.gz file. The gzip compression algorithm replaces repeated strings in a 32 KB sliding window with a pointer of the form (distance, length) to the previous and nearest

Format	% of Identical Data	
	Content-defined chunks	Whole file content
.rpm	9.08	7.07
.gz	6.71	5.29
.iso	31.26	0.54
.bz2	8.33	8.32
.tbz2	5.02	5.02
.raw	0.47	0.0
.tgz	13.55	13.55
.zip	2.79	1.43
.bin	2.57	0.0
.jpg	0.49	0.28
.Z	3.40	3.14
.gif	2.73	2.69
.tif	95.29	95.29
.img	33.84	8.69
.au	0.0	0.0
all ext.	20.27	5.03

Table 3.3: Detailed similarity pattern in our mirror of sunsite.org.uk. A 4 KB expected chunk size and a 48-byte sliding window size were used in the content-defined chunking method. The last row of the table shows the values obtained when all files in the data set were analysed.

identical string in the window. Distances are limited to the size of the sliding window (i.e. 32 KB) and lengths are limited to 258 bytes. As a consequence, redundancy elimination occurs within a relatively local scope; identical portions of data across files will be detected only if files are positioned close in the tar file. In contrast, the content-defined chunking method is able to find data redundancy across distant files in the data set and, in this particular case, to save more storage space than the compressed tar file.

3.4.1.1 Compressed Data

Compressed files (e.g. gz, bz2, zip, and Z) constitute an important segment of information within the sunsite data set: around 14 GB correspond to compressed files. The next experiment reports the potential storage space savings that might be obtained if, once data is decompressed, the content-defined chunking strategy is used to suppress duplicates, and then compression is applied again on the resulting set of unique chunks. The content-defined chunking method was selected to eliminate redundancy because it proved to be the most efficient strategy to find data similarity over the four main categories of compressed files. Tools such as `zcat`, `bzcat`, and `unzip` were used to decompress the files. The output stream generated by all these utilities was set as the input to the redundancy elimination program that used a 4 KB expected chunk size.

Figure 3.5 compares the storage space savings obtained in each of the four categories of compressed files using three different methods to eliminate duplication. The first method suppress duplication from the original collection of compressed files using the whole-file approach. The second method removes similar chunks of data from the original compressed files using the content-defined chunking method. Finally, in the third strategy the files are decompressed, redundancy is removed using the content-defined chunking method, and the resulting unique chunks are compressed again. Although the storage space savings are maximised using the third method, it barely outperforms the result obtained with the content-defined chunking strategy on the original files specially in the cases in which the original data set is of considerable size (gz and bz2 formats).

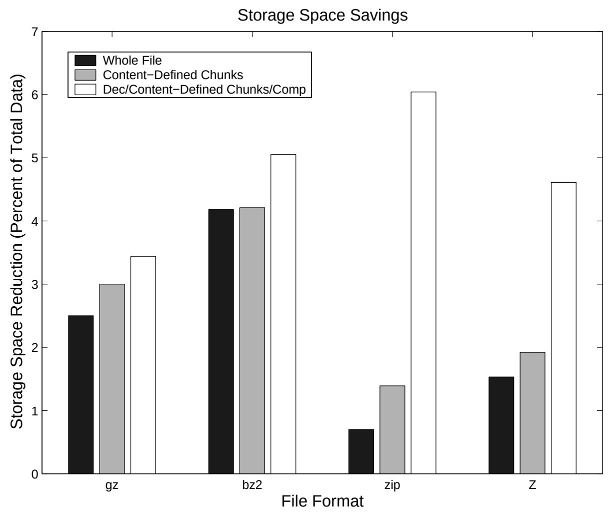


Figure 3.5: Storage space reductions using three different methods to eliminate duplication in compressed data. Original sizes of the data sets: gz=8.13 GB, bz2=2.4 GB, zip=1 GB, and Z=260 MB.

When duplication is removed from the uncompressed version of the files, the value obtained for the zip category is substantially different to those seen in the other two techniques. It also contrasts with the pattern observed in the other three data sets in which the differences between columns are fairly small. It seems that redundancy elimination specially helped zip files. In general, `zip` is used to deflate one file at a time to then include it into a single object; it limits any potential size reduction to intra-file compression. In contrast, the other compression tools also remove inter-file data duplication (e.g. from all the files in a tar) which finally reduces the benefits of redundancy suppression due to the content-defined chunking method. In conclusion, the improvement seen in zip files can be attributed to the ability of the content-defined chunking method to eliminate redundancy within a broader scope (i.e. inter-file redundancy); a gap that zip compression fails to address.

It seems that the comparatively slight storage savings obtained by decompressing files to eliminate redundancy, and compressing the result anew may not be enough to justify the computational overhead of the whole process. Douglis and Iyengar also analysed commonality patterns in compressed data [DI03]; they reached a similar conclusion.

3.4.2 Users' Personal Files

The data analysed in this section was held in 44 home directories of different users of the Computer Laboratory. Although the total amount of data processed was only 2.9 GB, this data set presented high diversity in the kind of files stored; it reported to have 1,756 different file-name extensions in 98,678 files with an average file size of 31 KB. However, the profile of the data set follows a clear pattern. Table 3.4 shows the most common file-name extensions in terms of popularity and storage space. Apart from the files without extension, home directories are mainly used to store files related to word processing and source code development. The 15 file-name extensions shown in Table 3.4 under the column related to storage space account for over 69% of the whole data set.

In this case the percentage of identical data in whole files was 12.80%. Table 3.5 shows the result of running the implementation of the content-defined chunking algorithm over the data set using different expected chunk sizes. It also shows the percentage of identical data found when the files were explored using only fixed size blocks. Although the performance of the content-defined chunking algorithm was better, significant storage space savings can also be obtained by using fixed size blocks.

The results indicate that using the content-defined chunking method and an expected chunk size of 2 KB, which is the best case, a storage utility would main-

Rank	**Popularity**		**Storage Space**	
	Ext.	% Occur.	Ext.	% Storage
1	–	19.47	.ps	17.24
2	.eps	4.83	–	11.73
3	.obj	3.98	.gz	10.66
4	.tex	3.82	.pdf	6.16
5	.c	3.43	.eps	4.58
6	.gz	2.85	.zip	4.13
7	.gif	2.45	.doc	3.13
8	.ps	2.28	.ppt	2.60
9	.dat	2.11	.obj	1.75
10	.html	1.81	.xls	1.53
11	.h	1.68	.tgz	1.29
12	.log	1.61	.tex	1.25
13	.aux	1.30	.c	1.24
14	.java	1.28	.so	1.19
15	.dvi	1.26	.txt	1.04
Total	——	54.16	——	69.52

Table 3.4: Profile of the data in 44 home directories of the Cambridge University Computer Laboratory. Files without extension are denoted by the – symbol.

Size	**% of Data in Identical Blocks**	
	Content-defined chunks	Fixed size blocks
8 KB	24.16	17.22
4 KB	26.76	18.05
2 KB	29.30	19.25

Table 3.5: Percentages of identical data obtained in 44 home directories of users of the Cambridge University Computer Laboratory. A 48-byte sliding window size was used in the content-defined chunking method.

tain only 2.3 GB of the original 2.9 GB. This represents a storage space reduction of 20.6%. However, the compressed tar version of the data set used only 1.4 GB of disk space; considerably outperforming the best duplicate suppression scenario. Similar storage saving ratios were obtained in the next two data sets (research groups' files and scratch directories) when their corresponding compressed tar files were generated.

	Popularity		Storage Space	
Rank	Ext.	% Occur.	Ext.	% Storage
1	.c	15.82	–	15.56
2	.h	14.51	.gz	10.20
3	–	13.90	.ps	8.01
4	.html	4.07	.c	6.66
5	.o	3.45	.a	3.29
6	.c,v	2.79	.pdf	3.15
7	.h,v	2.11	.o	2.97
8	.py	1.95	.eps	2.41
9	.gif	1.56	.tgz	2.18
10	.S	1.40	.h	1.93
11	.gz	1.23	.0	1.82
12	.if	1.14	.html	1.40
13	.m	1.01	.taz	1.29
14	.eps	0.97	.5	1.24
15	.s	0.85	.tar	1.23
Total	——	66.76	——	63.34

Table 3.6: Profile of the data stored by different research groups of the Cambridge University Computer Laboratory. Files without extension are denoted by the – symbol.

3.4.3 Research Groups' Files

This data set represents a collection of files stored by work groups; it contains the information of different research groups in the Computer Laboratory. This data set presents high degrees of similarity because it contains software projects, documents, and information shared among groups of people. This is an ideal environment to save storage space or to reduce the amount of data transmitted based on suppressing identical portions of data. The data set contained a total of 708,536 files in 21 GB of disk space and a 32 KB average file size. Table 3.6 illustrates the main sections of information arranged by file-name extension popularity and storage space used. Although 2,820 different file extensions were found, the 15 most popular extensions cover more than 66% of the files. Moreover, the percentage of data within the 15 extensions that use the most storage space accounts for over 63% of the total size of the data set.

The percentage of identical data in whole files was 25%. It clearly demonstrates an increment over the previous data sets. Table 3.7 shows the percentages of data in shared blocks for the other two methods: content-defined chunks and fixed size blocks for different expected block sizes. The rates of commonality

Size	% of Data in Identical Blocks	
	Content-defined chunks	Fixed size blocks
8 KB	37.01	28.77
4 KB	39.61	29.62
2 KB	44.59	32.94

Table 3.7: Percentages of identical data found in several research groups' directories of the Cambridge University Computer Laboratory. A 48-byte sliding window size was used in the content-defined chunking method.

obtained under the fixed size approach also present high levels of commonality although they are substantially behind the content-defined chunks' percentages.

Under this ideal scenario, not only due to the high amount of identical data contained in whole files but also due to the potential relationships between the files analysed, the use of Rabin fingerprints proved its efficiency. By using an expected chunk size of 2 KB, as indicated by the experimental results, a storage system would hold only 14 GB in unique blocks in contrast with the 21 GB of the original data set. This means a reduction in storage space of around 33%. Once more, the compressed collection of files (i.e. tar.gz format) used less storage space than the duplicate suppression techniques; it claimed only 8.9 GB of disk space.

3.4.4 Data Stored in Scratch Directories

The 100 GB of data explored in this section represents the largest data set studied. It contained a total of 1,959,883 files with an average file size of 55 KB. Table 3.8 gives a partial characterisation of the data set. Once more, it presents the information organised in two main columns according to file-name extension popularity and storage space used. This time the top 15 files, in terms of storage space, account for almost 70% of the total size. It is notable that a large portion of the data set is contained in files without extension or with the .log extension; they represent more than 57% of the whole data set. Apart from this fact, the information was evenly distributed over the whole set of files.

This large data set was explored with an 8 KB expected chunk size which enabled a reduction of the potential number of chunks generated. All the percentages of similarity dropped. Although the percentage of similar data in fully identical files was 14%, the value obtained using the content-defined chunking

Rank	**Popularity**		**Storage Space**	
	Ext.	% Occur.	Ext.	% Storage
1	.c	16.47	.log	33.41
2	.h	15.05	–	24.04
3	–	13.14	.gz	2.76
4	.o	6.55	.c	1.52
5	.0	4.00	.txt	1.46
6	.d	3.99	.o	1.24
7	.gz	1.94	.ps	0.81
8	.3	1.89	.a	0.70
9	.S	1.47	.pdf	0.69
10	.py	1.26	.ul2	0.60
11	.s	1.25	.xls	0.57
12	.ih	1.09	.dl1	0.56
13	.al	0.92	.il1	0.55
14	.1	0.84	.prof	0.55
15	.ast	0.81	.frag	0.53
Total	——	70.67	——	69.99

Table 3.8: Profile of the data stored in scratch directories in machines of the Cambridge University Computer Laboratory. Files without extension are denoted by the – symbol.

strategy was only slightly over 20%. The advantages of using the content-defined chunking method are minimal considering that the fixed size blocking scheme offered a value of 17.52%. According to the experiments, storage space used in unique chunks for this data set would be 88.26 GB and 91.59 GB, using the content-defined chunking method and fixed size blocks, respectively. The compressed tar version of this data set claimed 49 GB of disk space.

The two main categories of files in the data set in terms of storage space used were investigated separately: .log files and files without extension. Using the content-defined chunking strategy, these files showed values of around 0.3% and 30% of identical data, respectively. Files without extension represent a considerably large amount of data that is difficult to characterise. However, they presented better levels of correlation compared to those obtained for the whole data set (20%). When the fixed size blocking strategy was used on files without extension, the percentage of identical data reached a value of 24.5%.

On the other hand, files with the .log extension offered extremely low levels of similarity and none of the .log files analysed were identical. It negatively

Version	8 KB		4 KB		File
	tar	raw	tar	raw	raw
2.5.34	1.49	2.39	2.26	3.18	1.5
+2.5.35	43.42	94.46	57.41	95.43	93.8
+2.5.36	44.17	96.33	58.24	96.94	95.12
+2.5.37	44.62	97.09	58.85	97.71	95.31
+2.5.38	44.99	98.33	59.25	98.70	96.86

Table 3.9: Sharing pattern percentages in a succession of five Linux kernel distributions. A 48-byte sliding window size was used in this experiment.

influenced the sharing levels of the whole data set. Pathological cases such as these may be difficult to foresee and handle given the limited information that file names in these two categories provide about their contents; no relationship can be inferred *a priori* just by looking at the file names.

3.4.5 Software Distributions

In this experiment, five successive Linux kernel source distributions were explored. Initially, the content-defined chunking method was run over one of the kernels (i.e. 2.5.34) and then successive kernel versions were added one by one; the results were recorded at each step. Furthermore, the three possible states of the distributions (i.e. tar.gz, .tar, and raw files) were analysed.

When the files were in the tar.gz format the percentage of data in shared chunks was 0% in all cases. Table 3.9 presents the values obtained in the other formats (tar and raw files) with two different expected chunk sizes. The amount of information shared was substantially greater in tar files when the expected chunk size was smaller. Once more, this result suggests that a smaller expected chunk size increases the likelihood of content overlap.

Furthermore, the last column of Table 3.9 shows the values obtained when the whole file content of the original files was explored. Comparing these values with those obtained in the 4 KB expected chunk size over raw files, which was the best scenario, in none of the cases the difference is greater than 2%. This leads to the conclusion that, although the content-defined chunking method efficiently found identical portions of data, their main source was in wholly identical files. As would be expected, no similarity was found among the files in their tar.gz and tar formats when the whole file technique was used.

Figure 3.6 shows the discrete cumulative distribution function of chunk occurrences that was obtained considering the five kernels in their raw state and using an expected chunk size of 4 KB. These results indicate that the kernel distributions are very similar. When an ordinary file system holds different versions of Linux kernels in its primal state, it is storing the same information almost as many times as versions it holds. Storing a Linux kernel in its primal state adds around 145 MB of data but at least 95% of this information is already contained in chunks of preceding versions of the kernel. Therefore, a hypothetical storage utility would add only 7 MB of new data if it reuses the chunks already stored. However, distributing patches of the kernels in their compressed format continues being the most efficient method of propagation. For example, the largest patch in the set of kernels accounts for only 977 KB. Even if uncompressed, the size of this patch file is smaller than the size of non identical chunks that were found using the content-defined chunking method. Using the content-defined chunking method a hypothetical storage utility would transmit 7 MB in new chunks while the size of the uncompressed patch is only about 3.7 MB.

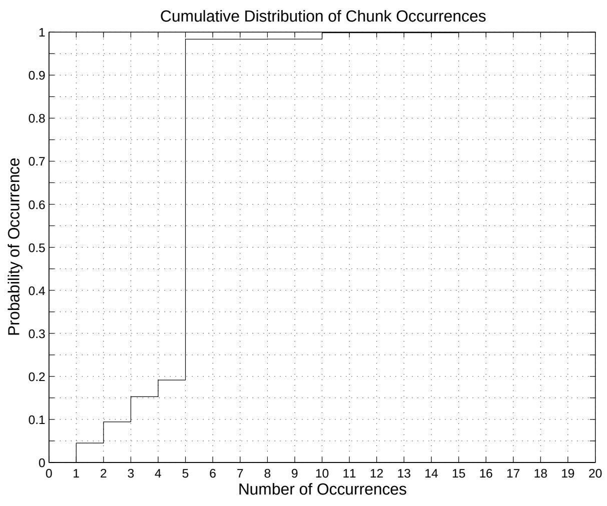


Figure 3.6: Discrete cumulative distribution function of chunk occurrences of five uncompressed and untarred kernels.

However, if these five kernels are decompressed and untarred, then processed with the content-defined chunking method in order to eliminate repetitions, and finally compressed again, the resulting size is only 38 MB. This value is considerably smaller than the original 171 MB used by five tar.gz kernel files, and only slightly over the 34 MB of an individual compressed kernel.

3.4.6 Associated Overheads

System designers considering employing any of these techniques to reduce data duplication must be aware of computational and storage overheads. Computational overheads are due to the calculation of SHA-1 digests and Rabin fingerprints. Additional CPU time and memory is spent in maintaining the data structures that keep track of the chunks generated (a hash table in this case) and their reference counters.

To compare the computational overhead in the three methods analysed, a 300 MB file containing random chunks of data taken from the data set corresponding to the Research Groups' files was created. The three methods were run on this synthetically created file. All the experiments were performed using a network-isolated machine with an Intel Pentium III 500 MHz processor. The content-based chunking method involves the generation of Rabin fingerprints and SHA-1 digests of the chunks. It took around 340 CPU seconds to process all the file. Around 76% of the total execution time was spent in tasks related to the computation of fingerprints over a sliding window. The fixed size blocking method took approximately 71 CPU seconds to compute all the necessary SHA-1 digests. Finally, the whole file approach used a total of 62 CPU seconds to calculate the digest. However, the reader should notice that the goal was to analyse data sharing patterns and potential storage space savings in diverse data sets; the prototypes were not implemented having performance as a compelling feature.

SHA-1 computations have been made extremely efficient. Commercially available hardware and operating systems' cryptographic services can be used to compute SHA-1 digests at very high speeds. Furthermore, Rabin fingerprints of a sliding window can be computed efficiently in software due to their algebraic properties, specially if the internal loop of the Rabin fingerprinting method is coded in assembly language [Bro93]. An efficient implementation and computation of Rabin fingerprints on real-life data sets has already been reported [Bro00].

Storage overhead is related to the number of unique blocks produced and the data structure used to keep track of them. As mentioned before, a hash table was used with this purpose. Figure 3.7 shows the storage space needed to store the hash table. The overhead has been computed for a 4 KB expected chunk size and fixed block size. It compares these values with the overhead introduced by the whole-file strategy in which only one digest per file is needed. In all the cases, the amount of extra space required is small compared with the total size of the data set, but would pose a significant burden were it to be stored in memory.

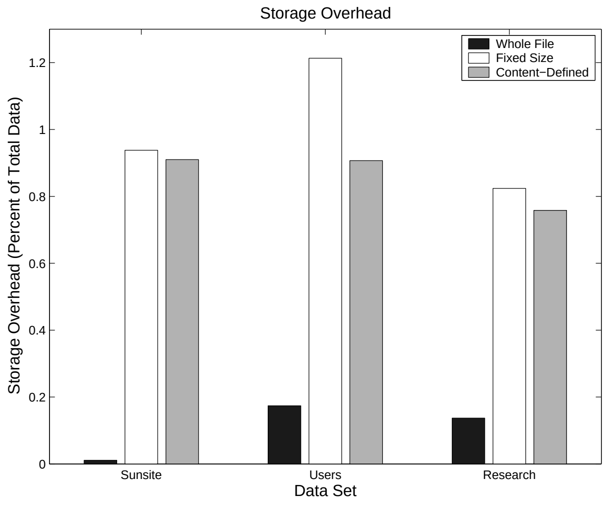


Figure 3.7: Storage overhead in three different data sets using a 4 KB expected chunk or fixed block size. Original data sets sizes: Sunsite=35 GB, Users=2.9 GB, and Research=21 GB

3.5 Summary and Discussion

Remarkable **data similarity patterns** were found in the experimental results. The content-defined chunking algorithm was the best strategy to discover redundancy in the data sets studied. It consistently reported the largest amounts of data duplication. However, the fixed size blocking strategy also revealed useful levels of similarity. As may be expected, the whole-file approach was always at the bottom of the ranking (see Figure 3.8).

The experimental evidence presented through this study indicates that a new file will often be created by making a number of changes to an original file; increased levels of duplication were found in data sets where this kind of update is prevalent. Specially in these type of scenarios, a block or chunk-level characterisation of storage systems' content is worth exploring.

For instance, the experimental results proved that the content-defined chunking strategy is particularly efficient with potentially correlated data where the aforementioned update pattern is dominant. Hence, high levels of similarity were obtained when this method was run on data held by research groups (44.59%), and expanded source code distributions and software projects (98.7%). When this method was used on more diverse data sets, such as the sunsite.org.uk mir-

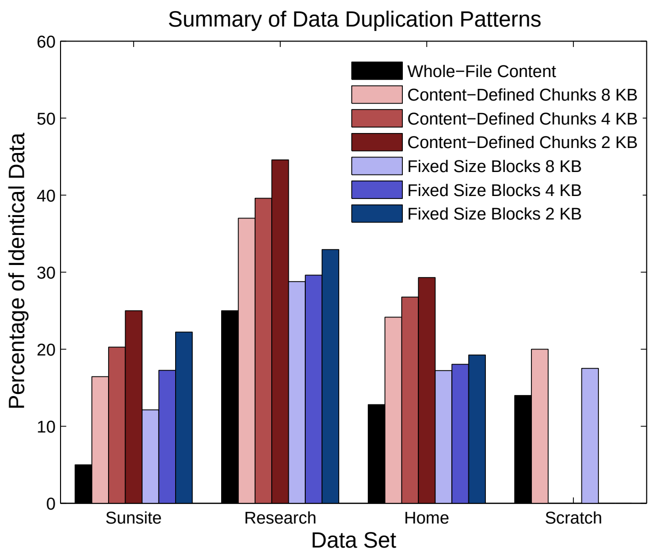


Figure 3.8: Summary of data duplication patterns in the four main data sets.

ror and scratch directories, the similarity levels dropped (25% and 20%, respectively), and were noticeably closer to sharing levels found using a fixed size block approach (22.23% and 17.53%, respectively). The whole file content approach reported modest levels of similarity with exceptionally high values in research groups' data (25%) and expanded source code distributions (96.86%). In data sets with a not-so-evident correlation and a more passive update pattern such as the sunsite.org.uk mirror the similarity levels plummeted (5%).

In terms of **storage space savings**, the tar.gz version of the data sets consistently outperformed the other techniques. However, for systems that need to access and update separate files, probably in a distributed environment, compression is not easy to implement effectively. The main disadvantage of this technique is that data has a strong correlation that is naturally embraced into one element (i.e. the compressed object) in order to be properly manipulated, and thus losing the potential benefits enabled by fine-grained data access strategies; chunk or block-based strategies such as those explored in this study might be a better option in this domain.

In general, packed (i.e. rpm) and compressed data presented low levels of similarity; compression algorithms have already removed a degree of redundancy in the data sets. Apparently, the storage space savings that can be obtained by decompressing a large number of arbitrarily selected files in order to remove data duplication from their expanded versions and finally compressing only non-identical chunks (see figure 3.5) does not justify the extra computational effort involved in this process.

Storage overhead should not be a serious impediment in the construction of data access technologies that fully exploit the benefits of data decomposition. The data structures needed to keep track of the extra information (SHA-1 block digests and reference counts) introduced a very small amount of storage overhead. The amount of extra storage required depends on the number of unique blocks managed by the hash table. As expected, the whole file approach created only a very small number of unique entries in the hash table (i.e. one per file in the data set). The fixed size and content-defined methods produced comparable amounts of storage overhead, considerably greater than those exhibited by the whole file approach (see figure 3.7). A practical study of **computational overheads** incurred in each of the methods remains to be done. This analysis, together with other experimental results [DH03], simply point out that there is an important amount of extra computation that has to be considered when using the content-based chunking method.

File access patterns should also be taken into consideration. Whole file content and fixed size blocking strategies present the disadvantage that file updates may lead to the recomputation of SHA-1 digests for large amounts of data. File updates under the whole file technique create the need to recompute the SHA-1 digest for the whole file. Using the fixed size blocking approach, any update that causes a shift at any position of the file will invalidate the SHA-1 digests for the rest of the blocks. As a consequence, reference counters of these blocks have to be decremented and SHA-1 digests have to be computed for the new blocks. However, a fixed size blocking scheme may offer the advantage that blocks can be page-aligned and consequently improve memory performance as is pointed out by Sapuntzakis et al. [SCP+02]. On the contrary, updates in the content-defined chunking scheme are self-contained into the blocks where they occurred, thus SHA-1 digests are recomputed only for the modified blocks.

In light of all these results, it is possible to say there is no one method able to perform satisfactorily on all data sets. The extra processing and storage space required for each of the techniques, together with usage patterns and typical workloads of specific data sets can be decisive factors when deciding to employ these techniques. The fixed size blocking method offers high processing rates, which make it a good candidate for interactive contexts. Despite the fact that the recomputation of SHA-1 digests could represent an inconvenience, specially under workloads in which file updates are common, the experimental results showed that the similarity patterns seen in passive data sets were relatively close to those obtained using the content-defined chunking strategy.

The question seems to be whether in practice the majority of the common blocks would remain valid after file updates and how often these updates occur. File access patterns [RLA00, Vog99] indicate that file updates present significant locality: only a very small set of files is responsible for most of the block overwrites. Files tend to have a bimodal access, they are either read-mostly or write-mostly. Finally, an important percentage of files, even under different workloads, are accessed only to be read [RLA00]. Considering this experimental evidence one feels persuaded to believe that, in the general case, an important number of blocks will remain valid for their lifetime.

Overall, the levels of data redundancy that can be identified using the fixed size blocking strategy are respectably high and sometimes close to those obtained using the content-defined chunking method. With file access patterns in consideration, the fixed size blocking strategy seems to be a sensible option for the general case; it is simple, acceptably effective, and quite efficient. The content-defined chunking method is justified only in contexts in which potential data repetition is high and the costs of not identifying redundant portions of data due to scattered updates throughout the file are elevated. Suppressing duplication at the file level still seems to be a good option specially where the amount of duplicated data is high and enclosed in a group of well connected machines [BCGD00]. System designers should take a decision based on the practical trade-offs between saving storage space, bandwidth consumption, and the computational and storage overheads necessary to support each of these methods; the results presented in this work can assist them in such a decision.

Chapter 4

Datom: An Abstract View of File Content

In this chapter I introduce Datom, a storage API that departs from the view of file content as a monolithic object by unveiling data types and internal data organisations [PP05]. Its main goal is to assist programmers to create advanced application-specific data access strategies. The Datom API represents an alternative storage interface to be of use for applications that exhibit file-like data access patterns whose application data is rich in structure and type. Consequently, it avoids the addition of functionality that may add unnecessary overheads and compromises for its target application domain.

In Section 4.1, I contrast the techniques analysed in Chapter 3 with the motivation of disclosing file data organisations through high-level abstractions. In Section 4.2, I present the rationale behind the creation of the Datom API. After that, in Section 4.3 I discuss the functional requirements of applications running on top of the file API in order to identify the type of functionality that the Datom API aims to provide. In Section 4.4, I introduce the data model supported by the Datom API. Finally, I compare the Datom API with other storage paradigms in Section 4.5.

4.1 Towards Meaningful Data Abstractions

In an attempt to disclose internal file data organisations through the file API, Chapter 3 analysed a set of techniques that make use of the generic file API to characterise file data according to their content. The experimental results demonstrate to what extent it is feasible to reveal particular fragments of file data using the generic services of the file API. They confirm that there are important amounts of repetitive data that might, ultimately, suggest the presence of internal file data layouts. This redundancy in storage system contents can be exploited to design improved data access strategies, as has been shown by diverse storage technologies [PP04]. However, from a programmer's point of view, there are some observations to this approach:

- **The understanding about file content is limited.** The level of abstraction that can be manipulated by developers based on the results of a redundancy analysis hides existing relationships between portions of data. Furthermore, it proves to be a deficient approach in terms of the semantic abstraction that is provided to developers since they keep manipulating opaque chunks of file content.
- **Data identification is only approximate.** The structural composition in storage systems contents that is unveiled with a redundancy study can only be considered a rough hint to real programming abstractions. The data chunks created do not necessarily match applications' abstractions mapped into file content.

If the primary goal is to disclose internal file data organisations with benefits for mainstream programmers, the previous two observations indicate that there are limitations in the use of these techniques. Furthermore, from the point of view of the implementation of these strategies, there are two more issues that should be considered:

- **Redundancy detection becomes subject to overheads.** Redundancy analysis techniques incur computational overheads to process file content and compute fingerprints of the analysed data chunks. Furthermore, storage overheads are also involved since these techniques need to hold block identifiers to track data redundancy.

- **Addition of management layers.** Ultimately, aside from the degree of structural composition in file systems contents that could be disclosed through a redundancy analysis, it is necessary to adopt a management layer to properly manipulate the generated data blocks. Depending on the implementation, this might involve the use of a database or hash table, and the modification of the original semantics of the storage system [MCM01, MPH02, CMN02].

Taking into account these observations, one of the core motivations for the creation of the Datom API is to move away from the services provided by the file API and to provide an API that enables a more fine grained and intelligent manipulation of application data. The advantages of the file API should be preserved only for those application domains whose functional requirements include streamed data manipulation as a must.

4.2 The Rationale for an Alternative API

The persistence model presented by file systems provides simplicity and generality. To enable these key features, it does not make any assumptions on the applications' data model or on their type of file contents. Traditional file systems technologies are agnostic of any applications' data structure or type, which finally results in developers having to write large amounts of code in order to access and manage persistent data. Developers find themselves repeatedly implementing the same IO services over an API that makes this task a tedious process.

Besides the historical reason of files being the most common persistent data abstraction in computing systems, file systems are a recurrent option in many application contexts for an important reason: *efficiency.* They represent a mature technology which introduces minimal overheads into applications' execution context. However, the data manipulation paradigm enabled by the current file system model was coined many years ago when the computational resources were more limited and applications imposed simpler requirements on both storage and developers.

By analysing the set of persistent data and applications that use file systems as their primary data repository, it is possible to observe that there is a vast amount of data rich in structure and type (e.g. PDF, HTML, XML, ELF, MPEG, JAR files, etc.). An important amount of data stored in files is amenable to structural decomposition. Even binary data such as bitmaps or sound files can be included in a multimedia document where they can be explicitly distinguished from each other under certain document organisation.

The rest of this section explores the rationale behind Datom's design. It first points out the most significant drawbacks that programmers experience when manipulating persistent data through the services provided by the file system API. These shortcomings are used to identify the areas of major impact in the interaction between the programmer and the storage system interface. This analysis is later used to define the goals of the Datom API.

4.2.1 Flaws in Persistent Data Access

The file API handles applications' requests in a generic and untyped way, which is mainly concerned with storing application data as blocks of bytes and with managing a reduced set of associated attributes in the form of metadata [DN65, Kle86, San86, Nag97]. From a developer point of view, the most crucial disadvantage of the file API is in its inability to recognise and manipulate persistent abstractions used by applications. The file API disregards particular applications' data access semantics and consequently promotes the creation of file formats that do not map well to the persistent abstractions manipulated by programs. As a result, writing code to manipulate persistent data with the file API exhibits the following shortcomings:

- **No support for data abstraction.** Any degree of structure, type, or semantic knowledge about applications' persistent abstractions that is present at runtime is lost within the file. As a consequence, developers spend significant amounts of time creating and debugging application-specific IO libraries in order to create meaningful abstractions in the host language. Changes in the format of persistent elements or file layouts automatically render the IO libraries of an application invalid.

- **Limited support for fine-grained data access.** Data manipulation through the file API greatly assumes access to a file as a whole, or in the best case to opaque regions of data within the file [CPS95]. It is a programmer concern to identify which sections of the file correspond to the elements requested by applications, to keep track of their location in the file, or to manage sharing of file content.

- **Lack of type-safe data access.** Not having the type-safe data access guarantees provided by the run-time abstractions that hold applications' persistent data is a potential source of programming mistakes and data corruption problems. Once more, it is the responsibility of programmers to ensure that data is fetched and restored from the appropriate programming data types.

- **Dismissal of application-specific update requirements.** Poor guarantees over the actual moment and extent in which persistent data is pushed to stable storage gives place to data corruption problems. Developers have to rely on system-wide mechanisms (e.g. `bdflush` kernel daemon) to transfer updates to disk. Under this approach, there is not a straightforward way to support application-specific update and consistency requirements.

These major deficiencies in the file API make effortless manipulation of persistent data elements unfeasible; they heavily impact the way in which file content is queried, updated, and managed by programmers. The ultimate consequence is that developers find themselves writing code to solve the same set of generic problems on a per-application basis. Using the file API to manipulate persistent data as arrays of bytes is a tedious task prone to errors and data corruption. Recovering persistent abstractions from file formats makes code abundant and wastes programmers' effort in tasks that should be automated or at least simpler.

Furthermore, the code written with these disadvantages violates, to an important degree, the properties that may lead to the creation of software of improved quality [GSAW98]. The manipulation of persistent data through plain arrays of bytes negatively affects the *understandability* of persistent code (i.e. code concerned with loading/storing persistent data) and forces programmers to maintain a mental mapping of two completely different data formats (i.e. runtime abstractions and their file mappings).

Reusability of persistent code is usually a painstaking process as it forces programmers to have a deep understanding of the on-disk data formats and layouts as well as of their management routines. *Maintainability* effort increases due to the explicit manipulation of low-level abstractions, the augmented code size, and the management of persistent data in two different states. To sum up, from a programmer perspective, development of persistent code with the file API is far from ideal.

4.2.2 The Goals of the Datom API

Whether the trade-off between efficiency and programmability is balanced or not has been brought to attention in recent projects [GSAW98, GBHC00, MMN^{+}04, Rec05]. Key data access capabilities seen in other storage paradigms in terms of support for data types and structure [ABS00, ABD^{+}89, SRL^{+}90], querying capabilities (i.e. SQL, OQL, XQuery), or effective concurrency and transactional support [GR93] have not successfully permeated into mainstream file-based applications. As a consequence, the file API presents evident drawbacks when compared with other technologies in terms of the ease with which programmers can access and manage persistent data.

The main goal of the Datom API is to provide a data access layer that departs from the flat-file paradigm to a more abstract and richer model in which applications' persistent abstractions are deliberately exposed. Therefore, the Datom API was created as an alternative way to represent, access, and manage file content.

The design of the Datom API has been influenced by two issues. First, it attempts to *fill the gap* created by the persistent data manipulation drawbacks pointed out in Section 4.2.1. Second, the design of the Datom API aims to *maintain the balance* between two attributes. The first is related to the functional requirements of its target application domain. The second is related to the overheads and compromises that more advanced facilities provided by other storage system APIs generate. Accordingly, its design pursues the following goals:

- **Expose persistent data access semantics.** The Datom API aims to augment the abstraction level that programmers are able to manipulate through the storage API. Thus, it will unveil and manipulate a judicious degree of the structure and data types that constitute the persistent abstractions employed by applications.

- **Improve programmers' interaction with persistent data.** The Datom API aims to improve the management of persistent code. Therefore, it will simplify the management of persistent data elements and assist programmers to create application-specific data access strategies with minimal effort.

- **Reduce impedance mismatch.** The Datom API aims to decrease the acute difference between file data and programming abstractions in order to simplify the development of persistent code. Therefore, from a programmer's perspective, it will present the persistent data space as a natural extension of the volatile space.

- **Observe file-type workload requirements.** The Datom API aims to service applications that exhibit the usage patterns observed in file-like workloads (see Section 4.3). Therefore, it will incorporate advantageous features seen in other storage paradigms while these are not contrary to the functional requirements of Datom's application domain.

Although these issues are pinpointed through Sections 4.3 and 4.4, respectively, they are analysed in detail in Section 4.5.

4.3 Characterising the Target Applications

The Datom API has been conceived as an alternative programming device to assist developers in the creation of persistent code for the application domain of file systems. The characterisation of the persistence requirements of this group of applications has one main goal: to provide hints to the type of functional features that the design of the Datom API should include. In addition, this analysis also exposes the type of data access facilities that may represent unnecessary overheads for this application domain. The most salient attributes of applications running on top of the file API are discussed in the next paragraphs.

Moderate complexity in persistent data sets

Many factors place file-based persistent data within data sets of moderate complexity, specially when compared with data in other application domains (e.g. data bases or PPLs). First, the *size* of the data set that has to be processed in a typical file-based application is still normally small [Sat89, BHK^{+}91], although the observed average file size has experienced steady growth over recent years [DB99].

Additionally, persistent data abstractions are organised on-disk in structured data layouts that are, or can be, generally mapped into programming abstractions resembling *collections* that aggregate them in an explicit manner. These collections of persistent objects have predetermined data access semantics and are used by applications at run time to manage file data. Finally, *file formats* are not expected to change during the useful lifetime of an application.

All these properties contrast with the data sets manipulated by applications in the database systems' context. In these environments, applications' data sets tend to be large. Furthermore, the persistent data model required by applications using object oriented database recreates fairly complex abstractions and relationships of persistent data at run-time. The changing nature of the data types in databases also contrasts with the more static nature of file formats; databases' schemas and data types are constantly modified as new logic and requirements are demanded from applications.

Remarkable data access patterns

Data access patterns in file systems applications show remarkable characteristics, presumably, as a consequence of today's file APIs and the data access requirements of applications using them. First, most files are opened to be read, and they will be processed in their entirety and in a sequential way. Write operations occur in a sequential form either covering the whole file or only portions of it [Vog99]. If a work session with a file involves both read and write operations, random access is the dominant pattern [RLA00]; although this use case accounts for a very small fraction of the file system operations.

An interesting observation is that larger files, which are becoming more common in recent file system workloads, tend to be read randomly. Therefore, current prefetching strategies, which simply prefetch blocks of files that are being accessed sequentially, seem to provide little benefit to both large and small files [RLA00]. If large files tend to be accessed randomly this kind of prefetching strategy may be ineffective; in the case of small files there will not be many blocks to fetch. One way to predict file access patterns more precisely would be to make explicit the data access semantics of applications through proper abstractions in the storage system API, similar to those employed by the Datom API.

The dominant whole-file and sequential data access pattern observed in file system workloads also indicates that applications running of top of the file API do not query data, instead they process it. This type of data access eliminates the need of APIs based on query languages such as those employed in database systems, letting aside the overheads that they introduce in applications.

Separation of concerns

The file API represents a general and data-centric approach that is agnostic to any run-time application abstractions and which is focused entirely on primitive data manipulation. In general, an application program sequentially reads an entire file into its address space and then performs non-sequential processing on the in-memory data abstractions. There is an explicit and sharp separation of concerns between the persistent and the volatile data spaces.

Persistent programming languages, specially those that support orthogonal persistence, eliminate the explicit separation of persistent and volatile data and tightly integrate data manipulation of these two spaces. From a programmer perspective, they represent the most advanced approach in the manipulation of persistent abstractions, but in exchange they force programmers to make crucial compromises such as the adoption of a determined programming language and consequently, the loss of generality. Therefore, a storage API for file-like data access should take into account the separation of concerns in the current computing model of mainstream platforms.

4.4 Data Model

The goal of this section is to present the data model supported by the Datom API. It introduces the collection of data structure types and the set of operations which can be applied to any valid instance of these types. Ultimately, the data model shows how to retrieve and derive data from any of the Datom types in any permissible way.

The Datom API follows two principles to improve the interaction with persistent data: expose application-specific **data organisations** and bring to light persistent data types through **information-hiding** modules. These principles are embodied in the two main types of persistent items of its data model: Composite Entities and Elements. The Datom API manages applications' data as a rooted graph made of these two types of persistent items.

Composite Entities are a set of managed abstract data types whose goal is to aggregate persistent items. They present a well-known interfaces to programmers with the semantics of stack, queue, list, map, and matrix elements; they are discussed in detail in Section 4.4.1. *Elements* are terminal data containers that hold application-specific programming abstractions; as such, their definition is provided by applications; they are discussed in detail in Section 4.4.2.

Formally, a persistent graph in the Datom data model is a rooted directed cyclic graph denoted by:

$$G = (V, E, r) \tag{4.1}$$

where V is the set of nodes in the graph, E is the set of edges in the graph, and r is the root of the graph. The set of nodes V is partitioned into Composite Entity and Element nodes:

$$V = V_{CE} \uplus V_E \tag{4.2}$$

where V_{CE} are nodes of type Composite Entity, and V_E are nodes of type Element. Composite Entities' types are members of the set:

$$T_{CE} = \{stack, queue, list, map, matrix\} \tag{4.3}$$

where *stack*, *queue*, *list*, *map*, and *matrix* represent the corresponding abstract data type. Elements' types are the set:

$$T_E = A \tag{4.4}$$

where A is the universe of application-defined data types. The edges E represent the *composition* relationship between nodes in V. A Composite Entity node V_{CE1} is said to be composed of other nodes in V, if these nodes are referenced by the V_{CE1} node. Element nodes V_E are terminal and, as such, do not reference other nodes in V. Finally,

$$r \in V_{CE} \tag{4.5}$$

By way of example, Figure 4.1 shows a graph that conforms to the data model supported by the Datom API. It illustrates the use of each of the six abstractions of the API and the composition relationship between them. The graph is rooted in a Composite Entity of *list* type from which all the other persistent items in the graph can be reached. The rest of the graph is composed of other Composite Entities and Elements. The composition relationship in the graph can be identified by the arrows that connect Composite Entities and by the immediate placement of Elements inside Composite Entities; in the latter case arrows were omitted for aesthetic purposes of the illustration.

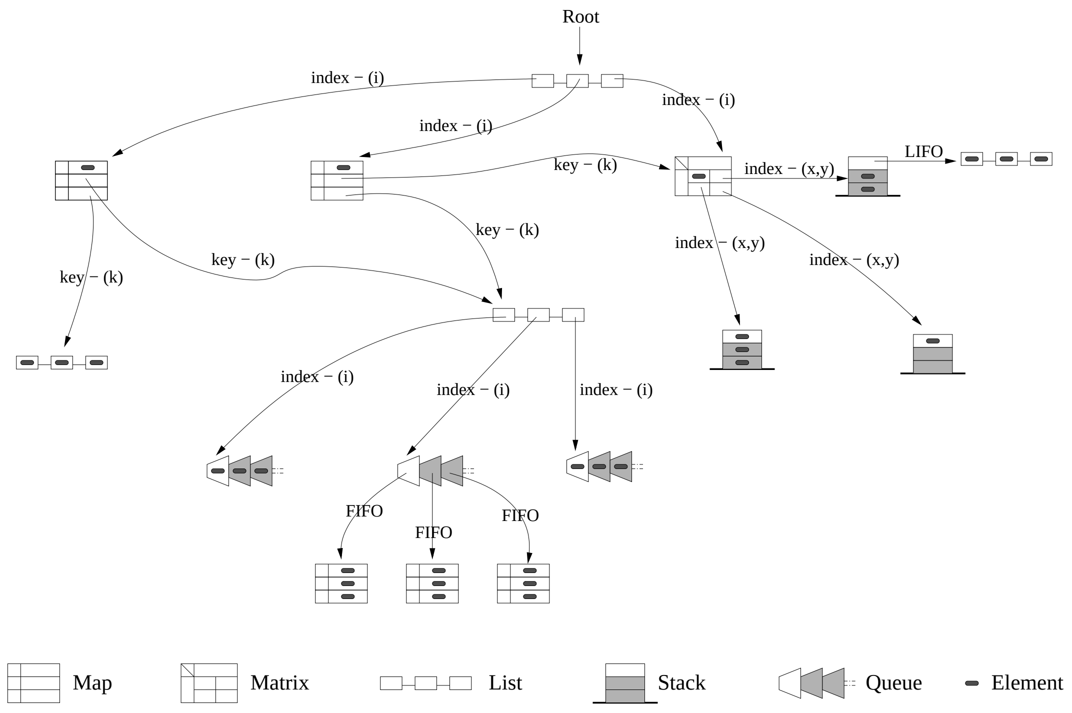


Figure 4.1: Graphical representation of Datom data.

Figure 4.1 also exemplifies the data manipulation approach enabled by the data model. Programmers manage persistent data using a set of well-defined abstractions with explicit access semantics. To be able to access a persistent item a program has to *traverse* the graph of persistence making use of the operations available at each node. These operations are defined in the abstract data types of the corresponding Composite Entity. For example, the list root in the illustration provides access to three Composite Entities, two of *map* type and to one of *matrix* type, through their corresponding indices in the list. To access these maps an application program has to use the list operation that gives access to its elements (e.g. `List.get(int)`); a similar strategy has to be employed to access persistent items inside other types of Composite Entities.

To manipulate and update the morphology of the persistent graph, an application program has to employ the operations supported by the Composite Entity type. For instance, to remove one of the maps contained in the list root in Figure 4.1, an application program would have to employ the `List.delete(index)` operation.

4.4.1 Composite Entities

Composite Entities are persistence-capable data structures which exhibit access semantics of a group of commonly used abstract data types. These data structures correspond to the nodes in the graph of persistence of the Datom data model. There are five types of Composite Entities, each of them supports different data access semantics: stack, queue, list, map, and matrix.

The decision to employ this set of abstractions as the primary interface to a storage layer follows one empirical observation: they are the most common data structures used by application programs to manage persistent data objects at runtime. Because these abstract data types are commonplace in programs, implementations of them abound in modern programming languages as native types or as additions into their standard libraries [AGH00, NET05, SGI05]. In the context of databases, these types have also proven useful. They have been used as an interface to the underlying database data model [BDB04], as a support tool to manage databases' query results [CBB^{+}00], or as fundamental storage elements [OBJ00].

A storage interface based on Composite Entities is a solution to some of the persistent data manipulation shortcomings that were identified previously in this chapter. Accordingly, this group of persistent data abstractions represents a practical solution that not only meets the goals set for the Datom API but also fulfils the persistence requirements of its target application domain (see Section 4.2.2 and 4.3). As such, they ease the programmability of persistent code. Furthermore, they can be of use to the underlying implementation of the API, which could employ the behavioural properties of the Composite Entities' types to investigate alternative data management solutions (e.g. better prefetching and caching strategies). Building the Datom API on top of these abstractions represents a minimalist approach to the issue of getting access to the abstract composition of file data.

Composite Entities help to *reduce* the impedance mismatch by providing a set of high-level abstractions that are frequently used by developers to manage persistent data once it is in the volatile data space. Therefore, they afford the possibility of modeling application data more accurately. They systematically *expose* persistent data semantics and applications' intent.

The combination of the different types of Composite Entities in a single graph of persistence aggregates application data items under high-level abstractions in which only a judicious degree of data relationships are exposed. Programmers are able to write code to navigate a graph made of basic building blocks that exhibit well-known interfaces. All these characteristics *improve* programmers interactions with persistent code and enable the management of on-disk persistent data layouts through high-level abstractions.

The data model enabled by the use of Composite Entities *observes* the functional requirements of a broad number of file-based applications. The Datom API favours navigation over advanced querying capabilities. This is specially practical in file-based applications since file data tends to be fully processed instead of queried, which indicates that data access requirements for these applications are mostly navigational. Furthermore, the group of high-level abstractions used by the Datom API are constantly employed to manage persistent data at runtime and, as such, they can be used to map applications' data models and data access semantics into persistent data layouts. Experience with abstract data structures [OBS99, GBHC00, MMN^{+}04] shows that data manipulation through a combination of constructor and inspection operations in a basic set of types can be used to design powerful data access strategies. Although not as general as SQL, this approach is rich enough to successfully build sophisticated data access services.

Providing high-level abstractions as an access layer to recover and restore persistent data may gain in importance over the coming years. The tendency in file-size growth observed in recent file system studies is accompanied with a more recurrent use of random access in larger files [RLA00], meaning that programmers will have to program more complex data access strategies on top the file API. Finally, a data-centric approach built on top of a reduced set of high-level abstractions maintains to a minimum the overheads of the augmented functionality without losing generality; this issue is related to the implementation of the Datom API. Thus, it will be discussed in detail in Chapter 5.

The rest of this section presents in detail the building blocks that make up the Datom data model, i.e. the stack, queue, map, list, and matrix abstractions. The interface of each of these elements is clearly defined using an algebraic specification of its operators. For example, Table 4.1 presents the specification of the

Datom stack; the interfaces of the other persistent items' types are analysed and presented in the same way.

The specification has four parts. The first part contains the name of the stack specification in capital letters. The next part introduces the types used in the rest of the specification. The *sort* value represents the name for a set of objects; it takes the name of the specification in lowercase as a convention. The *enrich* statement indicates that the stack type is also of PersistentItem type. The *imports* statement lists the names of the specifications that define sorts that are used in the stack definition.

The INTEGER and BOOLEAN specifications are not included in this document as their definitions are out of the scope of this dissertation. In a similar way, the algebraic specification of the PersistentItem sort has been omitted since it does not add any functionality to the public interface of a Datom stack; it is used only with the purpose of indicating that the Stack sort is a persistence-capable item.

The third part in the specification defines the signature of the interface to persistent items of the Stack type. It describes the name of the operations, the number and type of their parameters, and the type of their results. Finally, the last part presents the semantics of the operations that make up the Stack interface. They are defined using a set of axioms which characterise the behaviour of the data type. The axioms relate the operations used to construct and modify the Stack with operations used to inspect its values. They represent the semantics of the data type and the inferencing rules that can be applied to any valid instance of the data model to retrieve and derive persistent items from the graph of persistence.

Stack

A Datom stack is a persistence-capable collection of items in which only the most recently added item can be accessed; also known as "last-in, first-out" or LIFO. This abstract data type is interesting not only because it enforces LIFO semantics but also because it is explicit about the point of interest of the application among all the items within the collection.

The need for stack structures arises naturally in many processes in which traffic control is generated. Whenever a sequence of data items is to be processed, STACK

sort Stack **enrich** PersistentItem
imports INTEGER, BOOLEAN

New → Stack
PopOff(Stack) → Stack
Pop(Stack) → PersistentItem
Push(PersistentItem, Stack) → Stack
Clear(Stack) → Stack
SeeTop(Stack) → PersistentItem
IsEmpty(Stack) → Boolean
Size(Stack) → Integer

New = an empty Stack
SeeTop(New) = Fail **exception** (empty stack)
SeeTop(Push(pi, S)) = pi
IsEmpty(New) = True
IsEmpty(Push(pi, S)) = False
Size(New) = 0
Size(Push(pi, S)) = 1 + Size(S)
PopOff(New) = Fail **exception** (empty stack)
PopOff(Push(pi, S)) = S
Pop(S) = pi (Obtained by a combination of SeeTop(S) and PopOff(S))
Clear(S) = New

exceptions
  pi = Null ⇒ Fail Push **exception** (null object)

Table 4.1: Algebraic specification of a Datom Stack.

and some item is encountered for which the processing is to be deferred, stacks can be used for holding the item until processing can be done. Some common examples in the use of stacks are related to solving recursion problems, syntactical analysis, or control of program execution. They may be used to support a wide variety of tasks in applications such as a persistent page-visited history in a Web browser, a persistent undo sequence in a word processor, as well as an auxiliary data structure for algorithms that process persistent data using LIFO semantics.

Table 4.1 contains a formal specification of the operations of the persistence-capable Stack data type. The stack type, as well as the other types of Composite Entities, has two main types of operations in its interface: operations that create or modify the sort (i.e. constructor operations) and operations that evaluate or query its attributes (i.e. inspection operations). Thus, the Stack interface is defined informally as follows.

- **Constructor Operations.** The constructor operations of the Stack type are New, which creates an empty stack; Pop, which evaluates and removes the top item in the stack; Push, which piles an item into the stack; and Clear, which removes all the items from the stack. The PopOff operation is a hidden constructor operation which is introduced to simplify the specification; it removes the top item from the stack.

- **Inspection Operations.** The inspection operations of the Stack type are SeeTop, which evaluates the top item of the stack; IsEmpty, which tests if the stack contains any items; and Size, which evaluates the number of items in the stack.

Queue

A Datom queue is a persistence-capable collection of items in which only the earliest added item can be accessed; also known as "first-in, first-out" or FIFO. Similar to the stack data type, the queue type presents an interface with a single point of interest to applications. They are also used in programming tasks in which processing control is required. Queues are used in situations in which persistent data has to be processed in the order in which it was introduced to the persistent collection.

A persistent queue may have practical applications in many programming scenarios. For example, virtually every resource manager (e.g. printer, disk, CPU) employs a queue or a waiting list. Simulators also use event queues to store events to be processed in the order they occurred. When Web-based business applications communicate with each other, producer applications enqueue messages and consumer applications dequeue messages to queues on different machines. In addition, queues can be used as a supporting data structure for algorithms that access persistent data using FIFO semantics.

Table 4.2 contains a formal specification of the operations of the persistence-capable Queue data type. Each of these operations is defined as follows.

- **Constructor Operations.** The constructor operations of the Queue type are New, which creates an empty queue; Add, which adds an item into the queue; Dequeue, which evaluates and removes the first item in the queue; and Clear, which removes all the items in the queue. The Remove operation is a hidden constructor operation which is introduced to simplify the specification; it removes the first item from the queue.
- **Inspection Operations.** The inspection operations of the Queue type are SeeFront, which evaluates the first item of the queue; IsEmpty, which tests if the queue contains any items; and Size, which evaluates the number of items in the queue.

Map

A Datom map is a persistence-capable collection of items which associates keys to persistent values. Accordingly, a key value has to be provided to store and retrieve an item from a Map. The semantics of this data structure imitate those of a dictionary, which are useful in situations where persistent data can be associated with a key value. A map data type is one of the most important structures in computer science and it is frequently used in application programs. They are commonly employed to organise record-oriented data and to provide a simple mechanism for single-keyed databases. In the context of the Datom API, maps can also be used as building blocks of more complex data structures.

Table 4.3 presents the algebraic specification of the operations supported by the Map type. The ".=." notation indicates an infix operator with operands of type Key, which can be objects of any type that supports the equality operation. The precise notion of equality depends on the sort of objects to which the operator is applied. Since the equality operator is necessary for the correct manipulation of a Map item, the specification is explicit about its use. The constructor and inspection operators for the Datom Map are informally defined as follows.

- **Constructor Operations.** The constructor operations of the Map type are New, which brings an empty map into existence; Put, which inserts an item in the map; Delete, which eliminates the specified item from the map; and Clear, which removes all the items from the map.
- **Inspection Operations.** The inspection operations of the Map type are Get, which evaluates the specified item; IsEmpty, which tests whether the

QUEUE

sort Queue **enrich** PersistentItem
imports INTEGER, BOOLEAN

New → Queue
Remove(Queue) → Queue
Add(PersistentItem, Queue) → Queue
Dequeue(Queue) → PersistentItem
Clear(Queue) → Queue
SeeFront(Queue) → PersistentItem
IsEmpty(Queue) → Boolean
Size(Queue) → Integer

New = an empty Queue
SeeFront(New) = Fail **exception** (empty queue)
SeeFront(Add(pi, New)) = pi
SeeFront(Add(pi_1, Add(pi_2, Q))) = SeeFront(Add(pi_2, Q))
IsEmpty(New) = True
IsEmpty(Add(pi, Q)) = False
Size(New) = 0
Size(Add(pi, Q)) = 1 + Size(Q)
Remove(New) = Fail **exception** (empty queue)
Remove(Add(pi, New)) = New
Remove(Add(pi_1, Add(pi_2, Q))) = Add(pi_1, Remove(Add(pi_2, Q)))
Dequeue(Q) = pi (Obtained by a combination of SeeFront(Q) and Remove(Q))
Clear(Q) = New

exceptions
  pi = Null ⇒ Fail Add **exception** (null object)

Table 4.2: Algebraic specification of a Datom Queue.

map is empty; HasKey, which tests if the specified key refers to an item in the map; and Size, which evaluates the number of items stored in the map.

List

A Datom list is a persistence-capable collection of items referred by their position within the list. This abstract data type is interesting not only because it enforces an order among the items in the list but also because it defines a movable point

```
MAP ( Key: [ .=. → Boolean] )

sort Map enrich PersistentItem
imports INTEGER, BOOLEAN

New → Map
Put(Key, PersistentItem, Map) → Map
Delete(Key, Map) → Map
Clear(Map) → Map
Get(Key, Map) → PersistentItem
IsEmpty(Map) → Boolean
HasKey(Key, Map) → Boolean
Size(Map) → Integer

New = an empty Map
Get(k, New) = Null
Get(k, Put(j, pi, M)) = if k = j then pi else Get(k, M)
IsEmpty(New) = True
IsEmpty(Put(pi, M)) = False
HasKey(k, New) = False
HasKey(k, Put(j, pi, New)) = if k = j then True else False
Size(New) = 0
Size(Put(k, pi, M)) = if HasKey(k, M) = True then Size(M) else 1 + Size(M)
Delete(k, New) = New
Delete(k, Put(j, pi, M)) = if k = j then Delete(k, M) else Put(j, pi, Delete(k, M))
Clear(M) = New

exceptions
  k = Null ⇒ Fail Put, Delete, Get, HasKey exception (null object)
  pi = Null ⇒ Fail Put exception (null object)
```

Table 4.3: Algebraic specification of a Datom Map.

of interest in the sequence of items. As a consequence, it is possible to inspect all the elements in the list without changing its contents.

List structures are required in applications that require to manage collections of items in which the relative ordering between list items may need to be observed. Programming scenarios for the use of lists abound; they find their application either as primary programming objects or as building blocks of more complex data structures. Table 4.4 shows the algebraic specification of the persistence-capable List data type. Similar to the other Composite Entities, it has two types

```
LIST
```

sort List **enrich** PersistentItem
imports INTEGER, BOOLEAN

New → List
Append(PersistentItem, List) → List
Add(Integer, PersistentItem, List) → List
Delete(Integer, List) → List
Clear(List) → List
Get(Integer, List) → PersistentItem
IsEmpty(List) → Boolean
Size(List) → Integer

```
New = an empty List
Get(i, New) = Fail exception (out of bounds)
Get(i, Append(pi, L)) = if i = Size(L) then pi else Get(i, L)
Get(i, Add(j, pi, L)) =
   if i = j then pi
   elseif i < j then Get(i, L)
   elseif i > j then Get(i-1, L)
Get(i, Delete(j, L)) = if i < j then Get(i, L) else Get(i+1, L)
IsEmpty(New) = True
IsEmpty(Append(pi, L)) = False
IsEmpty(Add(i, pi, L)) = False
Size(New) = 0
Size(Add(i, pi, L)) = 1 + Size(L)
Clear(L) = New
Add(i, pi, L) = if i = Size(L) then Append(pi, L)
Delete(i, Add(j, pi, L)) =
   if i = j then L
   elseif i < j then Add(j-1, pi, Delete(i, L))
   elseif i > j then Add(j, pi, Delete(i-1, L))

exceptions
   i < 0 or i > Size(L) ⇒ Fail Add, Delete, Get exception (out of bounds)
   pi = Null ⇒ Fail Add, Append exception (null object)
```

Table 4.4: Algebraic specification of a Datom List.

of operations: constructor and inspection operations. They are informally defined as follows.

- **Constructor Operations.** The constructor operations of the List type are New, which creates an empty list; Add, which incorporates an item to the list at a specified position; Append, which adds an item at the end of the list; Delete, which removes from the list an item at a specified position; and Clear, which removes all the items from the list.

- **Inspection Operations.** The inspection operations of the List type are Get, which evaluates the value of the specified element; IsEmpty, which tests if the list contains any items; and Size, which evaluates the number of items in the list.

Matrix

A Datom matrix is a persistence-capable data structure that describes a two-dimensional arrangement of items. Thus, every position in the matrix is associated with a tuple of the form (i, j) where i and j denote the row and column indices of a given cell within the matrix, respectively. The Matrix data type has many applications. For example, they may be used in applications that associate coordinates to persistent data, or that manipulate information using tabular access semantics such as worksheets, embedded tables in word processing documents, or board games. Table 4.5 contains a formal specification of the operations of the persistence-capable Matrix data type. Each of these operations is informally defined in the following way.

- **Constructor Operations.** The constructor operations of the Matrix type are New, which creates an empty matrix and initialises the items of the matrix to Null; Set, which creates a matrix where a specified element has been assigned an item; DeleteRow, which removes the specified row from the matrix; DeleteColumn, which removes the specified column from the matrix; Resize, which modifies the dimension of the matrix to the specified values setting new cells to Null, if any were created as the result of the resizing; and Clear, which sets to Null all the elements of the matrix.

- **Inspection Operations.** The inspection operations of the Matrix type are Get, which evaluates the item at a specified position; IsEmpty, which tests if the matrix is empty; SizeRows, which returns the number of rows in the matrix; SizeColumns, which returns the number of columns of the matrix; and Size, which returns the number of cells in the matrix.

MATRIX
sort Matrix **enrich** PersistentItem **imports** INTEGER, BOOLEAN
New(Integer, Integer) → Matrix Set(Integer, Integer, PersistentItem, Matrix) → Matrix DeleteRow(Integer, Matrix) → Matrix DeleteColumn(Integer, Matrix) → Matrix Clear(Matrix) → Matrix Resize(Integer, Integer, Matrix) → Matrix IsEmpty(Matrix) → Boolean SizeRows(Matrix) → Integer SizeColumns(Matrix) → Integer Size(Matrix) → Integer Get(Integer, Integer, Matrix) → PersistentItem

New(i, j) = an empty Matrix
IsEmpty(New(i, j)) = True
IsEmpty(Set(i, j, pi, M)) = **if** pi ≠ Null **then** False **else** True
SizeRows(New(i, j)) = i
SizeRows(Set(i, j, pi, M)) = SizeRows(M)
SizeRows(DeleteRow(i, M)) = SizeRows(M) - 1
SizeRows(DeleteColumn(j, M)) = SizeRows(M)
SizeRows(Clear(M)) = SizeRows(M)
SizeRows(Resize(i, j, M)) = SizeRows(New(i, j))
SizeColumns(New(i, j)) = j
SizeColumns(Set(i, j, pi, M)) = SizeColumns(M)
SizeColumns(DeleteColumn(j, M)) = SizeColumns(M) - 1
SizeColumns(DeleteRow(i, M)) = SizeColumns(M)
SizeColumns(Clear(M)) = SizeColumns(M)
SizeColumns(Resize(i, j, M)) = SizeColumns(New(i, j))
continued on next page...

...continued from previous page

```
Size(New(i, j)) = i*j
Size(Set(i, j, pi, M)) = Size(M)
Size(DeleteRow(i, M)) = Size(M) - SizeColumns(M)
Size(DeleteColumn(j, M)) = Size(M) - SizeRows(M)
Size(Clear(M)) = Size(M)
Size(Resize(i, j, M)) = Size(New(i, j))
Get(i, j, New(k, l)) = Null
Get(i, j, Set(k, l, pi, M)) = if i = k and j = l then pi else Get(k, l, M)
Get(i, j, DeleteRow(k, M)) = if i < k then Get(i, j, M) else Get(i+1, j, M)
Get(i, j, DeleteColumn(l, M)) = if j < l then Get(i, j, M) else Get(i, j+1, M)
Get(i, j, Resize(k, l, M)) = if i ≤ k and j ≤ l then Get(i, j, M) else Null
Clear(M) = New(SizeRows(M), ColumnRows(M))

exceptions
  i < 0 or i ≥ SizeRows(M) or j < 0 or j ≥ ColumnRows(M)
    ⇒ Fail Set, Get exception (out of bounds)
  i < 0 or j < 0 ⇒ Fail New, Resize exception (out of bounds)
  i < 0 or i ≥ SizeRows(M) ⇒ Fail DeleteRow exception (out of bounds)
  j < 0 or j ≥ SizeColumns(M) ⇒ Fail DeleteColumn exception (out of bounds)
```

Table 4.5: Algebraic specification of a Datom Matrix.

4.4.2 Elements

Elements are the fundamental unit of storage for application data in the Datom API. Datom Elements can be regarded as docking points at which application programs load or store typed data. They represent application-specific programming abstractions with rich semantics and defined access routines. Accordingly, the definition and specification of Elements is done by developers based on applications' programming abstractions. This creates an API perceived by developers as extensible since they are able to model the interface to persistent data and to extend it, if necessary.

The main objective of Datom Elements is to use application-specific data semantics to manipulate persistent data. The fundamental motivation behind Datom Elements relies on the observations made by Keedy and Richards which highlight that file data should not be manipulated as a free-standing data structure [KR82]. Instead, files should be modeled as information-hiding modules that encapsulate the low level details of the manipulation of file data; such as the design and implementation of the major data structures and algorithms that process file content.

However, in the Datom API this principle is applied at sub-file granularities to provide programmers with a tool to discern the composition of file contents, breaking the vision of file data as a monolithic data structure. In parallel, this approach is closely related to the object model that associates operations to data abstractions, which is of common use in modern object oriented programming languages; Elements can be thought of as a compound of these fine-grained programming objects into one single item.

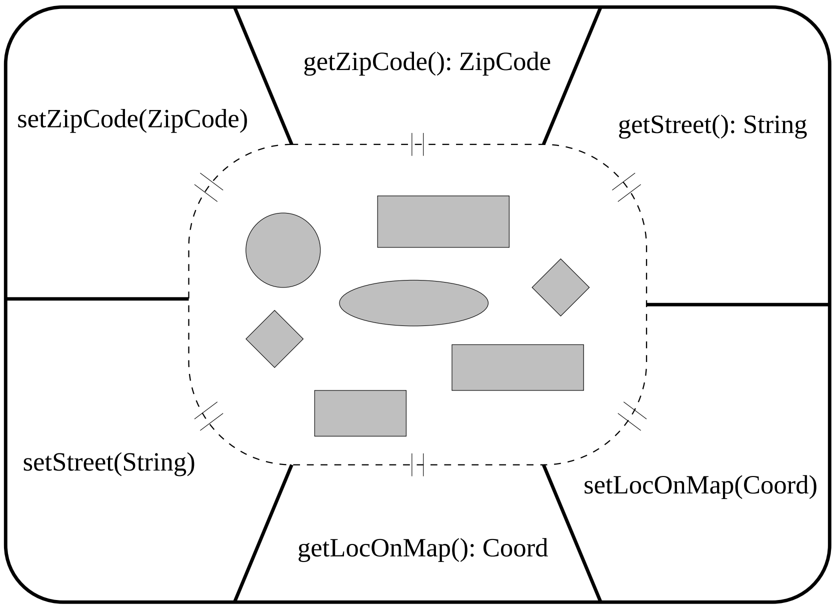


Figure 4.2: An abstract view of a hypothetical Datom Element.

Elements are type managers which aim to provide a light-weight mechanism to store application data. As far as the API is concerned, the data items stored inside a given Element lack identity and are managed through the holding Element. Elements' data is thought to be clustered together on disk since they clearly reflect the data access semantics of the application, and as a result, a powerful hint to spatial locality of reference. For example, Figure 4.2 shows a hypothetical Element in charge of managing persistent data corresponding to an address. The internal data abstractions of the Element such as the data structures or the code to operate on them are hidden and are only accessible through the Element's interface, which guarantees type safety in the manipulation of persistent data.

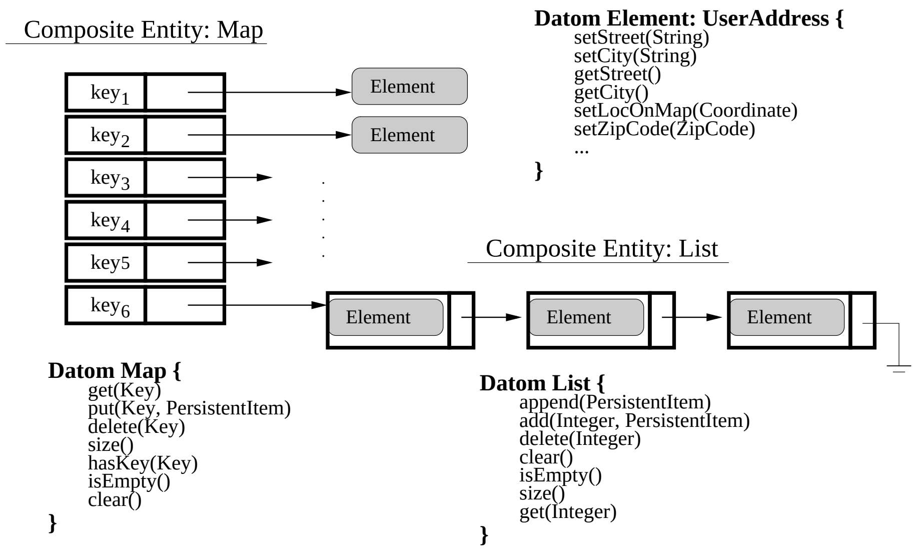


Figure 4.3: Example of the relationships and interfaces of persistent items in the Datom data model.

In this way, instead of accessing persistent data using the generic read and write operations on plain arrays of bytes as supported by the file API, developers create data access routines that directly employ application data access semantics. Figure 4.3 exemplifies the interactions between the main types of persistent items of the Datom API. As shown, Elements are final data containers that do not reference other persistent items in the graph of persistence. In contrast, Composite Entities are in charge of keeping the relationships between the different items of the graph and of providing access routines to these elements.

Figure 4.3 also shows that at each persistent item in the graph of persistence there is a well-defined set of operations to manipulate persistent data abstractions; the actual definition of Elements is discussed in the next Chapter. An application manipulating the graph of persistence shown in the figure using the Datom API would be able to include statements of the form:

$$\texttt{Map.get(key}_6\texttt{).get(1).getStreet();}$$

to obtain the desired information from within an Element, or to create sophisticated statements to access data in the graph of persistence such as:

$$\texttt{Map.get(key}_6\texttt{).append(UserAddress);}$$

4.5 The Datom API in the World of Persistence

The Datom API represents a data-centric and language neutral option to systematically and incrementally disclose files' structure and typed contents. This model assists programmers with more powerful persistent data abstractions enabling not only the provision of new services to applications, but also the manipulations of persistent data in a more intelligent way. Figure 4.4 shows the gap that the Datom API attempts to fill in the world of the main storage paradigms. The API's tenet is to positively impact both the ability of programmers to manage persistent data and the morphology of persistent code. The *structure* of data exposed by the Datom API eliminates code used to parse and serialise data from a flat file model; the programmer no longer manually manages internal data layouts.

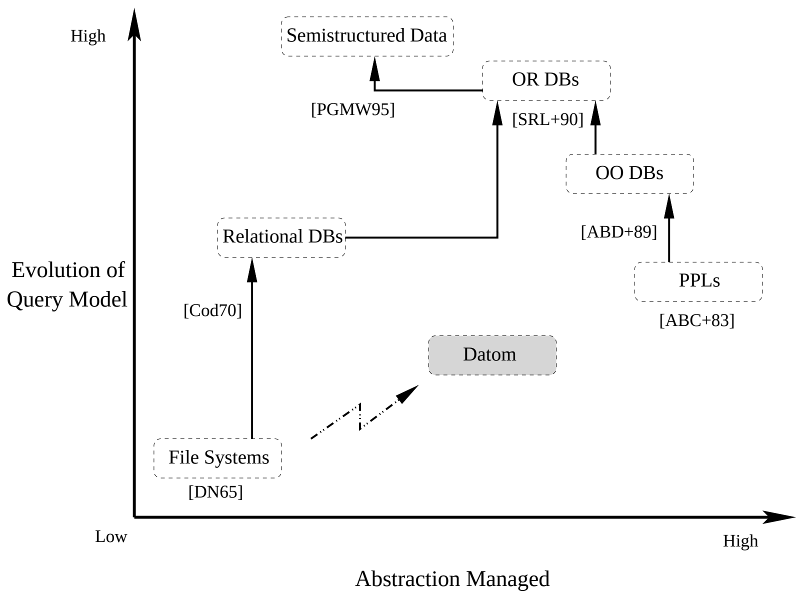


Figure 4.4: Datom API in context with the main storage system paradigms.

The Datom API aims to enable the creation of tightly integrated navigational querying capabilities based on data types and structure instead of relying on whole file analysis and regular expressions to extract specific data elements. Improved data access strategies to persistent data should be provided relying on different mechanisms such as key-based access, navigation and positioning, data types, or content. Additionally, according to the computing environment or user requirements, applications can create different policies to load data from disk or across networks such as whole-prefetching or incremental access [PCV03].

If the storage system API also includes meaningful *information-hiding elements* as its fundamental unit of access, code is made more understandable and naturally assists in providing data integrity by eliminating a whole range of errors due to wrongly accessing and updating data. Furthermore, it may also enable data sharing through an open data interface that can be manipulated using different programming languages. By accessing application data according to its semantics, programmers have explicit knowledge on what and how data persists.

Although other storage technologies offer solutions to each of the issues mentioned above, the Datom API has been specially crafted for data access requirements as typically seen in file-based applications. It stresses a balance between the trade-offs faced by programmers aiming to migrate from the file paradigm to a more evolved data model in the search of the benefits brought by higher levels of abstraction in the manipulation of persistent data. Next, a detailed analysis of the major persistence paradigms in comparison with the rationale of the Datom API is presented.

Databases

Databases, whether relational or object oriented, are designed to cope with data-intensive workloads, dynamic querying capabilities, and advanced transactional frameworks with ACID properties. A considerable amount of persistent data of applications running on top of file systems is amenable to structural decomposition; however, its access patterns do not map properly to database functionality. Applications that use databases accept as necessary the overheads introduced by a high-level query language (e.g. SQL, OQL, XQuery) such as parsing, query optimisations, access path and plan selection, and query execution in exchange of the ability to dynamically query the content of the database.

However, the majority of applications using the file API have well-defined access patterns that can be predicted in advance. This offers the advantage that programmers might be able to create ad-hoc, and consequently, more efficient data access strategies directly on top of applications' abstractions, since the overheads associated to query languages are removed. Furthermore, the kind of access pattern enabled by query languages, in which very small portions of data are selected dynamically, highly differs with the consistent access patterns exhibited in file system workloads; data in this context is mainly accessed navigationally and processed following a more coarse-grained access pattern.

Strict ACID properties incur in important overheads by coordinating the actions of a potentially large number of processes against the same information, probably stored in a database table. High data contention and no tolerance to inconsistencies in application data have shaped databases' design. If applications do not require transactional semantics for up-to-the-last operation, as is the case of Datom's application domain, the overhead added to support ACID properties has little value and a negative impact on performance [WD92].

Object relational [SRL+90] and object oriented databases [ABD+89], aim to provide all the advantages of the object oriented programming model to the persistent world. They are aimed to support data-intensive applications with complex data models that require the services of robust and advanced transactional frameworks (e.g. long and nested transactions). Thus, they add persistence to objects, including their associations and methods, and support major features such as complex objects, object identity, type hierarchies, classes, extensibility, and computational completeness. This approach introduces additional overheads to the traditional database paradigm which, as has already been discussed, represents a mismatch with the requirements of applications using the file API.

Persistent Programming Languages

Persistent programming languages [ABC+83, ADJ+96, Car86, DCBM89] allow programmers to manipulate persistent data directly from the host language without using any form of an intermediate data-manipulation language (e.g. SQL). The impedance mismatch problem, as typically seen in other storage paradigms, is solved by providing orthogonal persistence [AM95] to programming language abstractions. PPLs are aimed for applications that require an intimate composition of long-lived data and programs (i.e. Persistent Applications Systems).

This strong cohesion between data and logic opposes the sharp separation of concerns observed in file-based applications and environments. Data-centric environments dictate that only a set of abstractions should be taken into account with persistence purposes and the rest disregarded. Consequently, PPLs present problems of integration with mainstream computing environments. This situation is evident in the sort of difficulties to properly handle external state reported in recent implementations [JA98] of persistent programming languages (e.g. graphic libraries, network parameters and utilities, system managed objects such as files).

Furthermore, confining type support to a specific language compiler and tightly integrating the store format with a specific programming language restricts developers into that programming language. A possible workaround to this limitation would be to adopt a form of intermediate language supported by a common language runtime as done in the .NET Framework [NET05]. However, no implementation of such a multi-language persistent platform has yet been reported. Additionally, saving data in a format which tightly binds persistent data to a programming language might become a limitation since normally data outlives the programming language used to create it [MT03].

In contrast with the orthogonal persistence feature supported by PPLs, in which every single object is a potential candidate for persistence, the Datom API constitutes a data-centric strategy that stress a judicious manipulation of structure and data types with explicit control of persistent abstractions through a richer and neutral API. The Datom API preserves the prevalent separation of concerns seen in file-based applications while aiming to reduce its negative impact on persistence related programming tasks.

Semistructured Data

Plenty of research has been done in the area of XML [BPS+04] to handle semistructured data. Among other important features, XML data has undoubtedly proved its value in enforcing syntax-level interoperability [Bra03]. However, current manipulation of XML file formats has limitations mainly due to the programmatic APIs used to process XML data; tools depend heavily on stream-based data manipulation to disclose file structure and data types. This reproduces many of the limitations observed in traditional file processing such as substantial programming effort or whole-file processing.

Pull based APIs such as SAX [SAX05] represent a light-weight solution to XML processing but the programmer has to create his own object model and write code to handle SAX events. Furthermore, since data is processed sequentially, backward data navigation is not possible. This is a major disadvantage to developed advanced data access strategies. The only feasible workaround involves storing processed XML and writing extra code; that results in memory overheads and defeats the purpose of using SAX.

Tree-based APIs such as DOM [HHW+04] create whole-file in-memory representations and thus are able to provide tree-like navigation. However, this model puts great strains on system resources, specially if the XML document is large. Furthermore, applications will normally need to build their own more diverse data structures with particular access semantics. It is extremely inefficient to build a tree-like data structure from a file format only to map its elements into different data structures, that will be processed with completely different semantics (e.g. lists), and then discard the original data structure.

When compared with XML APIs, the Datom API aims to explicitly expose data type and structure without relying on internal file formats by employing a rich set of programming abstractions equipped with fully navigational and incremental loading capabilities (discussed in Chapter 5). These differences enable the Datom API to work better for diverse data sets and to create more sophisticated strategies to manipulate persistent data.

Other Works

Attempts to move away from the traditional file API to higher levels of abstraction have been investigated in different works. Compound files in the OLE Structured Storage Model [Bro94] represent a solution to the problem of internal file structure. In a compound file, *storage* and *stream* objects have a strong parallel with directories and files in a conventional file system; they are organised hierarchically and their respective APIs show strong similitudes.

By using compound files programmers avoid the burden of directly dealing with internal data placement leaving this task to the file system infrastructure itself; they also improve application performance due to features such as incremental access in which only the required portions of the compound file are loaded into memory. However, the compound files API reproduces most of the limitations seen in the traditional file API. Data objects are stored as untyped byte arrays in closed application data formats; programmers are still left with the onerous task of developing persistent data access code using an untyped API.

Gribble et al. [GBHC00] explore the utility of distributed data structures and higher-level abstractions (e.g. hash tables, b-trees) as a storage infrastructure replacing the traditional flat view of data as a persistent data management layer. They highlight the advantages of these data structures such as a more structured interface with higher-level abstractions; operations are applied to the data structure itself instead of to a range of bytes. Under the same line, the Boxwood project [MMN$^+$04] explores the utility of data structures as storage infrastructure. Higher-level abstractions and structural information inherent to the data abstractions can enable the system to perform better load-balancing, data prefetching, or informed caching.

To a certain extent the Datom API shares the ideology of these works [Bro94, GBHC00, MMN$^+$04]. However, one of the core goals of Datom is to show how application data can be modelled and effectively manipulated through a richer API that discloses file data organisations, relationships, and data types. Moreover, it aims to expose particular data access semantics through the combination of practical ADTs. Furthermore, the Datom API stresses programmability features and the impact of improving the persistent substrate from a programmer point of view.

Berkeley DB [OBS99, BDB04] supports efficient storage of record oriented data using a simple (key, value) access mechanism. Different to the Berkeley DB approach, which is completely unaware of applications data types, the Datom API aims to fully exploit the notion of type and the judicious employment of pointers present in specific system-managed ADTs. Thus, the data model supported by the Datom API enables the creation of richer data relationships based on nesting and aggregation of fundamental data storage abstractions. Additionally, Datom's persistent Elements allow access to data through a well-defined interface created based on application-specific data types.

Microsoft is currently implementing WinFS [Rec05], a layer over an existing file system (NTFS [Nag97]). It provides a richer data storage model than traditional file systems and attempts to unify access to file systems, relational databases, and object oriented databases. With its incorporated schematised data model, WinFS aims to systematically expose persistent data types, structure, and relations. WinFS programming model enables access to the persistent store via three different APIs (i.e. managed WinFS, ADO.NET, and Win32), which give programmers the ability to manipulate persistent data according to their needs (i.e. relational, object-oriented, XML based, or Win32 files).

Another approach proposes type-system and language extensions to natively support different data models (e.g. relational or semistructured) within a statically typed object oriented environment [MS03]. This approach allows objects, tables, and semistructured data to be constructed, updated, and queried in a unified and type-safe manner. A noteworthy implementation of this proposal is the Language Integrated Query (LINQ) project [HB05], which adds query capabilities to the Common Language Runtime (CLR) and languages that target it. The query facility builds on lambda expressions and expression trees to allow predicates, projections, and key extraction expressions to be used as opaque executable code or as transparent in-memory data suitable for downstream processing or translation. The standard query operators defined by LINQ are integrated with ADO.NET and `System.Xml` to allow relational and XML data to gain the benefits of language integrated query.

The key difference between the Datom API and these proposals (i.e. WinFS and LINQ) is that it does not aim to unite current persistent data models or facilitate a whole programming platform. Instead it aims to offer an alternative storage API based on a different set of ADTs whose main goal is to provide an option to file-based storage; an area where other storage APIs seem to fall short.

4.6 Summary

In this chapter I proposed to deliberately uncover file data organisations in order to ease programming effort and better model applications' persistent abstractions. Thus, I introduced the Datom API that deliberately departs from the view of file content as a monolithic and flat object by unveiling data types and internal data organisations. The Datom API represents a data-centric solution that aims to bring to light application data types using information-hiding elements, which in turn are organised under high-level programming abstractions.

The Datom API aims to exploit the benefits of a more abstract API in terms of software productivity, code understandability, and maintenance costs. When compared with the mechanisms used by other storage system APIs, its design contrasts with the introduction of minimal overheads and compromises.

Chapter 5

From Analysis to Implementation

In this chapter I present the implementation of the Datom API. The prototype combines techniques that ease the management of the data model supported by the Datom API. Its novel persistence model aims to balance the trade-offs between functionality, overheads, and generality. Distinctive features in the implementation of the API are automatic data movement between the volatile and persistent data spaces, mindful use of resources based on the application's access patterns, automatic memory management, full control of update granularity, and atomic updates. The strength of the prototype relies on the improvement in persistent data manipulation enabled by the combination of its data model and the mechanisms that support its persistence model effectively.

In Section 5.1, I introduce the persistent model supported by the implementation of the Datom API. Then, in Section 5.2, I present the high-level architecture of the prototype of the Datom API. The rest of this chapter presents the implementation of the prototype following a top-down approach. Thus, in Section 5.3 I discuss the Interface Subsystem, which presents the functionality of the Datom API to developers. In Section 5.4, I introduce the Storage Management layer. Then, in Section 5.5, I present the Persistent Data Composer, which provides the translation services of the API. Finally, in Section 5.6, I discuss the Physical Storage Subsystem.

5.1 Persistence Model

The persistence model supported by the implementation of Datom is a feature that affects the prototype's design at different levels. It defines how to write code in order to manipulate both transient and persistent items. Thus, it specifies the way in which persistent data is created and deleted, as well as how the internal mechanisms of the system are able to modify the lifetime of persistent items. The persistence model of the Datom API represents a novel strategy called *selective reachability*, a hybrid model derived from the combination of two original persistence models: persistence by reachability [AM95] and persistence by type [Sol92].

Selective reachability aims to integrate the strengths of both of these approaches. Persistence by reachability frees the programmer from the burden of writing code to control the movement of data among the hierarchy of storage devices and from coding translations between long-term and short-term data representations, since all these tasks are performed automatically by the system. However, one of the main criticisms of this approach is the potential risk of unintentionally pushing to disk unwanted data. To be able to precisely determine which objects persist, programmers need not only to have a skillful command of a whole programming language, but also to keep a mental representation of the graph of persistence. This puts the strain of managing a large conceptual chunk on programmers, i.e. they need to keep track of a large amount of information to properly manipulate persistent data. This, ultimately, may restrain them from using this type of persistence model.

In contrast, the persistence by type approach makes explicit the distinction between the two groups of data, i.e. volatile and persistent. This differentiation can be used as a programming tool that not only defines the boundaries of the graph of persistence but also facilitates the manipulation of the conceptual chunk for developers. In addition, it may be employed to optimise access to objects which are known not to be persistent [RCS93] and to incur overheads only for those objects that are managed as persistent.

In the implementation of the Datom API, the identification of persistent items is done by the system in an **automatic** way by determining their reachability from a root of persistence. However, membership of the graph is restricted by **type**; only items that have been explicitly declared as persistence-capable can be included as valid nodes. Membership is enforced by the type system of the host programming language. This approach does not suffer from the problem of dangling pointers, or invalid references, in persistent data that points to non-persistent objects, since all references take place among persistence-capable items.

Selective reachability represents a persistence model that is suitable for the data manipulation approach enabled by the Datom API which defines as persistent only a well-defined set of objects: Composite Entities and Elements. It supports programmers by relieving them from writing code to control the movement of data and from coding translations between the volatile and the persistent space. Furthermore, by explicitly defining the boundaries of the persistent graph, developers are provided with a programming device in which the conceptual chunk can be more easily processed and managed.

For the most part, the techniques used to implement Datom's persistence model are influenced by the adoption of Java as the programming language of choice. The main limitation in this approach is the reduced ability to define memory management strategies on top of operating system primitives. On the other hand, the introduction of a level of isolation between the implementation of the Datom API and particular hardware configurations augments the portability and generality of the library.

The rest of this chapter presents the implementation of the Datom API and shows how selective reachability is accomplished in the prototype. Unless otherwise stated, the rest of this chapter uses the term *Datom* to denote the implementation of the Datom API.

5.2 The Big Picture

This section provides an architectural overview of Datom. The prototype developed is a fully-functional application library which enforces both the data model specified in Chapter 4 and data persistence through selective reachability. The prototype has been programmed in the Java programming language using the Sun's Java 2 Standard Edition v1.4.2 virtual machine implementation. Datom is available as a JAR package that is included as a Java library in applications' code[1].

[1]Available at `http://www.cl.cam.ac.uk/users/cbp25/datom/datom.jar`

Figure 5.1 breaks down the prototype by component. The implementation has been organised using a layered architecture made of interchangeable software modules with well-defined responsibilities and interfaces. The isolation of functions enabled by a layered architecture facilitates fine-tuning or even the replacement of specific functionality within layers without impacting the rest of the prototype; a characteristic that eases successive refinements in the implementations of the Datom API.

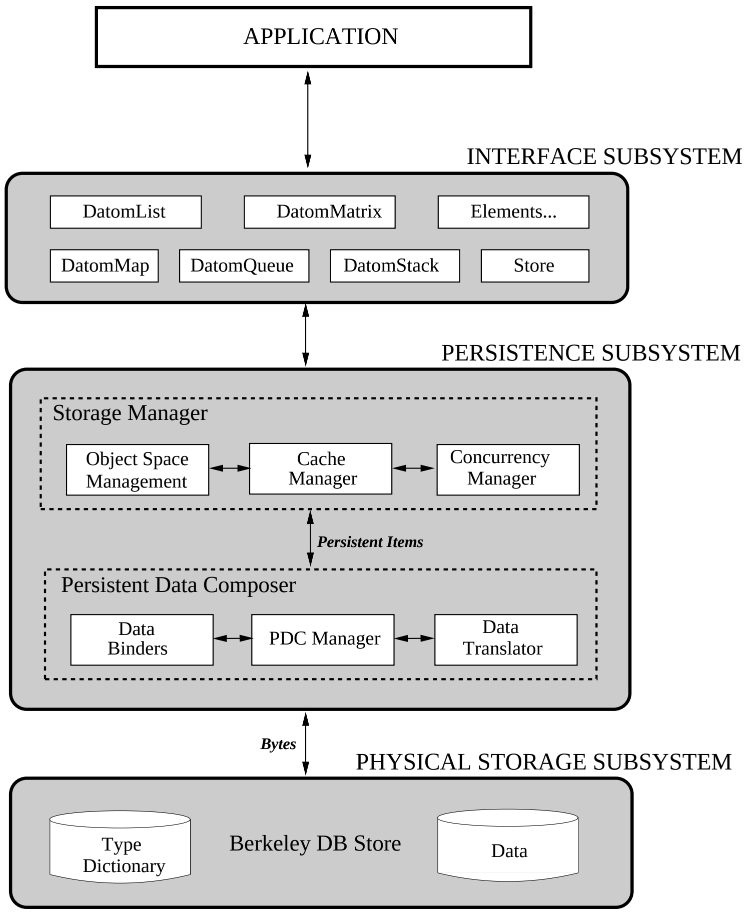


Figure 5.1: High-level view of the layered architecture of the implementation of the Datom API.

The responsibilities and modules of each of the subsystems that make up Datom are defined as follows.

- **Interface Subsystem.** It exposes to programmers the abstractions of the Datom API and the persistence model that defines the API implementation. It also validates applications' operations on persistent items.

- **Persistence Subsystem.** It creates, destroys, and moves persistent data between the volatile and the persistent data spaces. It is composed of two sublayers: *Storage Management* and *Persistent Data Composer*. The former validates items' identity, performs memory management, and executes concurrency control while the latter conducts the transformation of persistent data between the physical storage format and the run-time representation of persistent data.

- **Physical Storage Subsystem.** At the bottom layer, the *Physical Storage Subsystem* is in charge of securely storing and fetching persistent items from secondary storage as requested by the Persistence Subsystem. The current implementation of the Datom API uses the Berkeley DB Java Edition, an off-the-shelf technology that is a good match for the requirements of a Physical Storage Subsystem for the prototype of the Datom API.

The typical interactions between each of the layers of the prototype are shown in Figure 5.2, which illustrates two important issues in the architecture of the Datom API. First, interactions between layers are performed through a well-defined set of calls where each layer uses the capabilities of the layer below and adds new capabilities while abstracting the underlying complexity. Second, it shows the transformations of persistent data at the different layers in the prototype.

At the Interface Subsystem layer persistent items are full-blown and semantically rich programming objects. Applications manipulate persistent data in this form through object handlers that provide access to the operations offered by the persistent item. Below the Interface Subsystem, the Storage Manager manipulates persistent items in their primitive form. It uses a logical life-long identification number (PID) that is associated with every persistent item.

At the Persistent Data Composer (PDC) layer items are in the process of being recovered from disk or being serialised to be sent to physical storage, so they take the form of flat arrays of bytes. For storage and retrieval purposes, the PDC employs a simple interface to the Physical Storage layer which associates the item's PID with its flat representation; accordingly, they are fetched or stored by their corresponding PIDs. Finally, the serialised version of a persistent item represents its form on disk. The next sections discuss in-depth the main functions and implementation of each of the layers that constitute the prototype of the Datom API. It follows a top-down approach from application interactions to the description of the Physical Storage Layer.

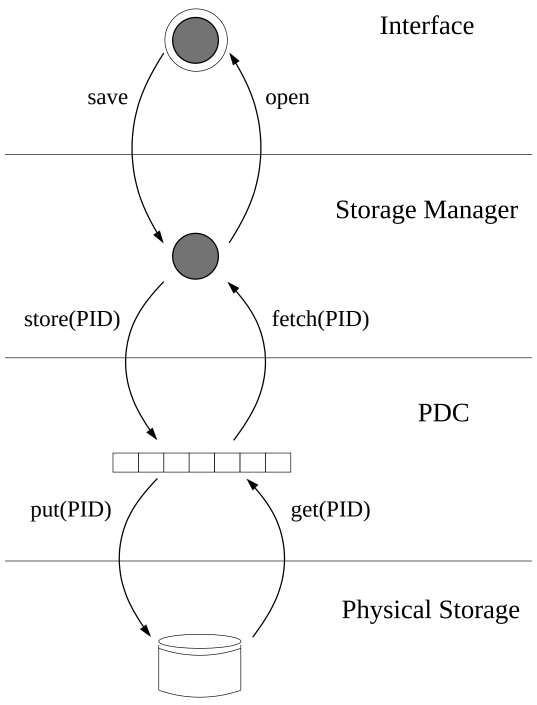


Figure 5.2: Persistent data manipulation in the layered architecture of the Datom API prototype.

5.3 Interface Subsystem

The fundamental role of the Interface Subsystem layer is to expose the necessary functionality to operate on persistent items. In addition, it aims to hide from programmers the details of interactions with the physical storage medium.

The group of public classes of the Datom API validates the data model at the Interface Subsystem layer in various ways. First, they enforce membership of the graph of persistence by using the type checking mechanisms of the host programming language; in this way only valid networks of persistent items can be created. Furthermore, the typed interface offered by data Elements warrants type safety since all the persistent items that are entered or retrieved through this interface are type checked at execution time. Abnormal manipulation of persistent items is managed by Datom through a set of exception classes that report detailed information about the nature of the error.

Another important function in the public interface of Datom is related to control over updates. The process to *stabilise* a graph of persistence (i.e. to push uncommitted updates to disk) can be triggered in two ways: implicitly or explicitly. An implicit invocation occurs when a root item is inserted in a `Store` object. If this operation is successful then the root of persistence has been validated. An explicit invocation takes place when the `save()` method is executed from a validated root item. The stabilisation of a graph of persistence involves running the reachability algorithm that identifies the mutated members of the graph together with the new items that should be *promoted* to disk. However, the internal mechanisms of this process are fully hidden from the programmer's view.

5.3.1 Components

To be able to persist data, programs need to interact with the `Store` object, which represents the initial point of access to the persistent infrastructure of Datom. It provides methods to create an entry for a specified graph of persistence in the store, recover it, or delete it. The `Store` maintains a directory of valid entries which associates names to instances of items acting as roots of persistence. Programmers fetch stored graphs of persistence by name from the `Store`.

Figure 5.3 shows the different data types provided by the implementation of the Datom API and their relationship[2]. Selective reachability enforces membership in the graph of persistence by type. Concretely, all data items in the graph of persistence inherit their persistent capabilities from the `PersistentItem` base class. This class includes the basic functionality to identify, manage, and persist data items.

The full functionality of the API is exposed to programmers through the programming abstractions that can be identified in the figure as the class definitions in bold. Programmers manage the graph of persistence through this set of classes that, ultimately, instantiate the abstract data types that constitute the data model supported by the Datom API. They are defined as follows.

- **DatomList.** This class represents a persistence-capable abstraction that holds a collection of persistence-capable items ordered by their position in the list.

[2]A full reference of the public classes of the Datom API is available at `http://www.cl.cam.ac.uk/users/cbp25/datom/apidocs/index.html`

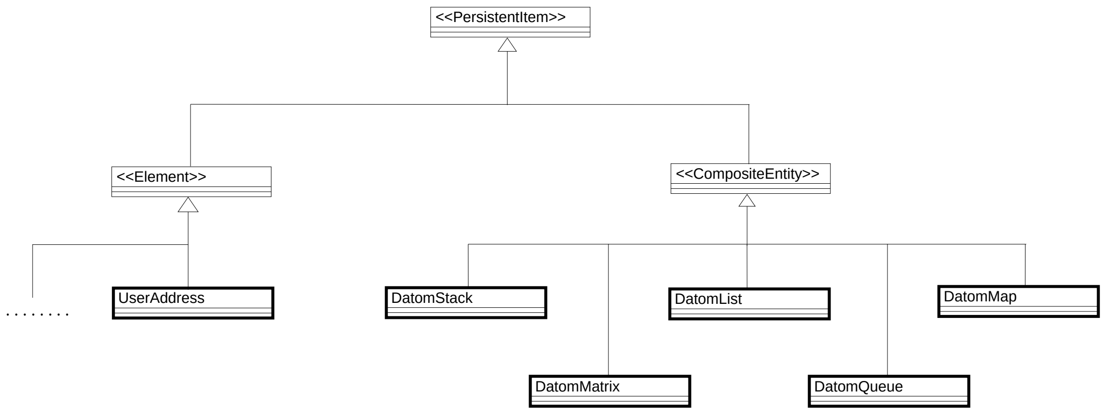


Figure 5.3: Overview of the classes that expose the functionality of the Datom API.

- **DatomMap.** This class provides a persistence-capable abstraction which stores associations between keys and their corresponding persistence-capable items.

- **DatomMatrix.** This class represents a persistence-capable abstraction that holds a collection of persistence-capable items organised in a two-dimensional arrangement.

- **DatomStack.** This class provides a persistence-capable abstraction that stores persistence-capable items using LIFO semantics.

- **DatomQueue.** This class provides a persistence-capable abstraction that stores persistence-capable items using FIFO semantics.

Applications never instantiate a `PersistentItem` class since this is only an abstract class that enables persistence in concrete programming abstractions. Similarly, `CompositeEntity` and `Element` types are abstract type definitions which enrich the persistent item concept and provide specialised capabilities to the programming abstractions that enforce the data model of the Datom API. The implementation of these three classes is discussed in detail in Section 5.4.1.

Actual application data is defined by inheriting persistence capabilities from the `Element` type. An example of a persistent data Element defined by the programmer is shown in the class diagram of Figure 5.3 with the name `UserAddress`; this class realises the application type introduced previously in Figure 4.2.

5.3.2 Simplifying the Creation of Datom Elements

In contrast with the fixed interface provided by Composite Entities, the public interface of data Elements is fully defined by the programmer. Programmers can perform this task from scratch or, to simplify the creation of persistent Elements, can use the ad-hoc automatic code generator tool provided with the distribution of Datom. This tool automatically creates a persistence-capable class that exposes an interface to the data types specified in a definition file. The format of the definition file is simple and is comprised of a header line and a list of the fields for which calls in the class interface should be created. Thus, the format of the definition can be described as:

```
fully-qualified-element-name [toString] | [equals] | [hashCode]
field-Type field-Name  "validation-java-snippet"
```

Figure 5.4 shows the file definition for the interface of the UserAddress Element. The fully qualified Element name provides its visibility in the context of the application, i.e. its package name. The `UserAddress` name is followed by a list of options that override the default implementation of the specified methods in the Java object model. This is done to automate the definition of methods whose default behaviours either do not match a data-centric approach or represent useful utilities. Accordingly, `toString` returns a string representation of the Element based on the contents of its fields that is useful for debugging purposes; `equals` evaluates the equality of state of UserAddress (opposed to equality of identity); and `hashCode` defines the hash code for the Element based on the content of its fields. Each field definition includes a validation snippet which aims to provide the necessary data-integrity conditions over the values passed through the Element's interface.

```
datom.data.examples.UserAddress        toString equals hashCode

Street       String                    "X!=null  &&  X.trim().length()>0"
Number       int                       "X>0"
ZipCode      com.application.ZipCode   "X!=null"
LocOnMap     com.application.Coord     "X!=null"
```

Figure 5.4: Example of a data Element definition file.

The result of running the code generator on a specification file is the class definition of a persistence-capable Element with a public interface to query and update its fields. To create this class definition, the code generator uses a predefined template file containing code formats for building Elements. The code formats are managed using a property list in which property names correspond to specific portions of the Element's class. Properties' values are generic strings formatted to accommodate the parameters taken from the definition file. Properties' values are defined using the patterns supported by the `java.text.MessageFormat`, which enables these parameters to be set dynamically at runtime. Programmers may enrich the resulting class definition to fine tune Elements' behaviour, if needed.

Aside from allowing to write code quickly, the automatic code generator used in Datom offers various advantages [Her03]. It aims to augment the quality of the code since Elements are derived from a high-quality and debugged template. The naming of the output class is completely consistent, which makes Elements' interface easy to use and understand. The method used to designate the names of the interface members follows the *getter* and *setter* naming conventions; a common pattern used in object oriented programming. Furthermore, it is possible to rebuild the code base rapidly. Finally, a type definition file can be used by multiple programming languages to derive Datom Elements from the same source.

5.4 Storage Management Sublayer

The Storage Management Sublayer involves creating, destroying, and moving persistent data between the persistent and volatile spaces. Accordingly, the Storage Management Sublayer makes use of the translation services provided by the Persistent Data Composer (PDC). The life cycle of persistent data items in Datom is controlled by the Storage Manager which automatically moves data from memory to disk and vice versa.

The possible states of a given item in Datom are depicted in Figure 5.5. New items are persistence-capable only and stay in the application-managed memory space until they are promoted to persistence (i.e. a copy of these items is created on disk for the first time). Once they have been promoted, they have a corresponding copy on disk and may have an entry on Datom's cache. Items reach the disk because they have either been promoted to persistence or evicted from Datom's cache. They can be recovered from disk as requested by the Storage Manager. For identity purposes items are assigned a PID for their lifetime.

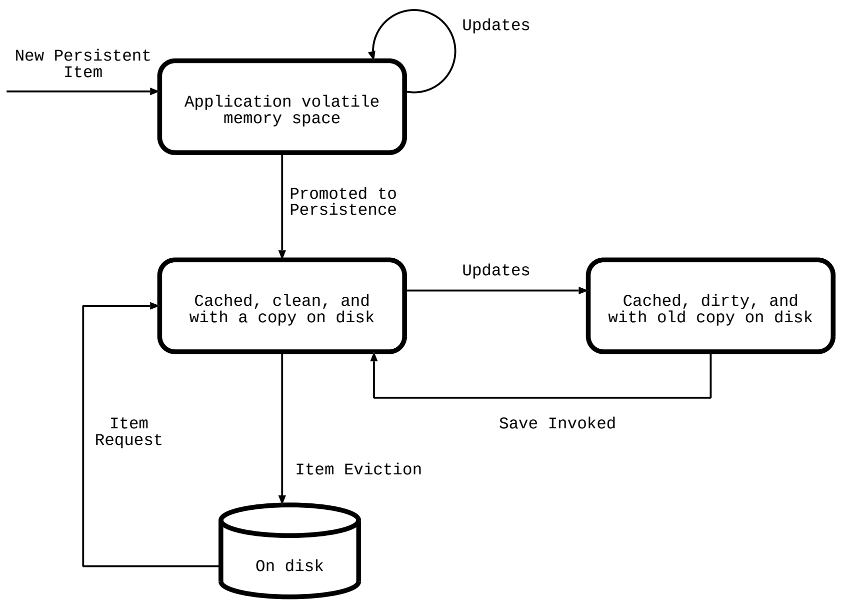

Updates
New Persistent Item
Application volatile memory space
Promoted to Persistence
Updates
Cached, clean, and with a copy on disk
Cached, dirty, and with old copy on disk
Save Invoked
Item Request
Item Eviction
On disk

Figure 5.5: Life cycle of persistent items.

The persistence model supported by Datom relies on selective reachability, which controls the identification of persistent data and their automatic promotion to disk. This method identifies persistent data by computing the transitive closure of all items reachable from a given root of persistence. Two mechanisms are fundamental to support this operation: incremental data loading and promotion to persistence. These mechanisms are described in detail later in this Section.

5.4.1 Components

Figure 5.6 shows the main software components of Datom's Persistence Subsystem, which is divided in two sublayers: *Storage Management* and *Persistent Data Composer (PDC)*. This section elaborates on the former; a detailed discussion of the latter is presented in Section 5.5. The major software components involved in storage management are defined as follows.

- **Storage Manager.** This component is responsible for coordinating the operation of the Datom prototype. It controls persistent items' life cycle by interacting and synchronising the operation of the rest of the software elements in the prototype.

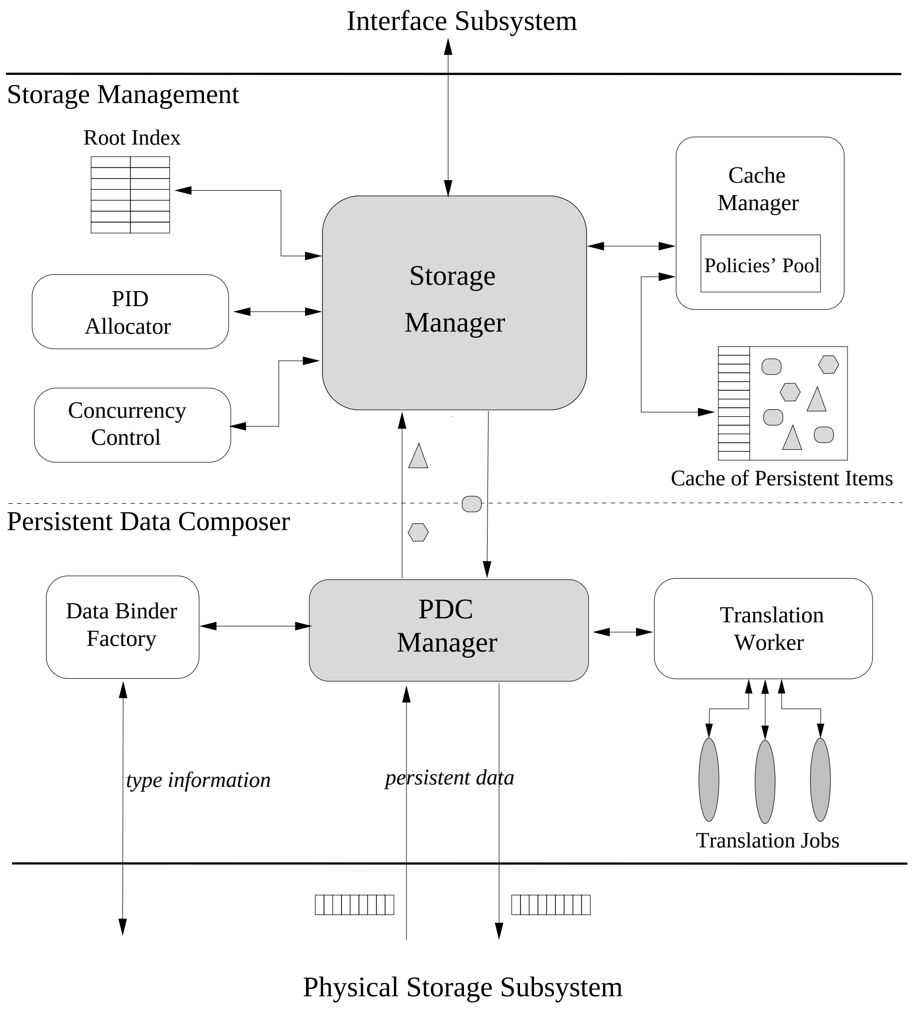


Figure 5.6: Main software components of the Persistence Subsystem layer.

- **Root Index.** The Root Index is a persistent data structure that stores the associations between root names and their corresponding PIDs. Datom identifies persistent items using a unique identification number (PID) which is assigned to objects at *promotion* time; PIDs are associated to objects for their whole lifetime. Entries to the index are added or deleted according to applications' operations.

- **PID Allocator.** The PID Allocator is in charge of creating PID numbers and dispatching them as requested by the Storage Manager. Since PID allocation involves disk access to guarantee that numbers are assigned uniquely and securely even in the event of system crashes, they are produced atomically. To avoid performance penalisation, the PID number is incremented in steps of 300 each time. In this way, the Storage Manager has control of a pool of PIDs without suffering expensive disk accesses every time it needs to assign a new PID. The step size can be modified in the setup of Datom.

- **Cache Manager.** The Cache Manager handles a cache of persistent items according to the eviction policy indicated in the configuration of Datom; although at the moment Datom enforces only one eviction policy. The cache is used to prevent items from being removed from memory by the garbage collector of the Java Virtual Machine (JVM) and to avoid translation penalisation for items that are constantly requested by the application.
- **Concurrency Control.** The Concurrency Control module manages concurrent access to persistent items. Datom uses the Java `synchronized` mechanism to enforce concurrent access to persistent items.

Persistent Items' Internals

The provision of the fundamental persistence capabilities of the Datom's abstractions are provided by the **PersistentItem** abstract class (see Figure 5.3). It preserves the identity of persistent items by ensuring that a given persistent item is assigned a PID only once in its lifetime. This class also manages a channel of communication with the Storage Manager. This conduit is used to notify it about any relevant application activity on items, mainly those that affect their state such as updates, and reference swizzling.

By default, a newly created item has its PID set to a null value. It is only when it is promoted to persistence that it is assigned a meaningful PID. In a similar way, the connection with the Storage Manager is initially set to null and updated on promotion. Finally, the `PersistentItem` class defines the implementation-wide mechanism for marshalling and unmarshalling persistent items by extending the functionality of the `serializable` interface of the Java programming language to their child classes.

The other classes that support the persistence functionality in Datom, i.e. `Element` and `CompositeEntity`, enrich the capabilities of the `PersistentItem` class. The **CompositeEntity** class is an abstract definition of the generic functionality that has to be implemented by the classes that are part of the set of Composite Entities, i.e. list, map, matrix, queue, and stack. This interface is used by the Storage Manager to control items in a uniform way, independently of their specialised type.

Thus, the `CompositeEntity` class defines abstract methods for swizzling and deswizzling a persistent item, clearing item's references, freeing memory resources, and querying an item about its references. The implementation of these abstract methods is provided by the `CompositeEntity`'s derived classes (see Figure 5.3). The provision of this functionality enables the Storage Manager to correctly manipulate persistent data. From a programmer's point of view, this functionality is hidden.

`CompositeEntity`'s specialisation classes are declared as `final`. This prevents programmers from introducing spurious data to the graph of persistence by extending these classes and allows the compiler to perform optimisations via inlining expansion of the methods in these classes. Inlining presents the trade-off between code size and performance gains. In the context of the Java programming language, it is implied that the compiler is able to detect this trade-off and choose wisely whether to inline a final method.

Furthermore, a Composite Entity aggregates all of the item's core information. Thus, with the exception of the inspection operations that involve traversing nodes in the graph of persistence, Datom is able to answer all the inspection operations promptly without involving additional disk accesses. Internally, Composite Entities employ standard objects of the Java Collections library as their back-end data structures. The advantage of this approach is the facility to fine tune the internal behaviour of Composite Entities easily due to the different implementation options offered by the Java Collections classes.

For example, the *DatomList* Composite Entity employs the `Vector` class as its back-end data structure. The implementation of this class may perform either aggressive or reserved memory allocation strategies according to configuration parameters. This type of configuration parameters may be available for applications in future releases of Datom to provide applications with the opportunity to manage resources in a clever way.

The **Element** abstract class is a type definition for a particular group of persistent items in the API, those defined by applications. The definition of this class ensures that their specialisation classes cannot redefine the functionality employed by the Storage Manager to control Composite Entities in a generic way.

Persistent items notify the Storage Manager about **updates** using the method `update` defined in the `PersistentItem` class. Update notifications cause the Storage Manager to include persistent items in the set of mutated objects. Update notification is supported implicitly for Composite Entities. By convention, Datom defines that *setter* members in an Element's interface update its contents. Accordingly, the code generator tool automatically includes update notifications in all the interface members that use this naming convention.

However, for cases in which programmers decide to build Elements from scratch, or design interfaces with more elaborate patterns, it is a developer responsibility to include update notifications in the Element's definition where appropriate. The advantage in this approach is the flexibility to create efficient update notification strategies by fine-tuning the placement of notifications in the code. Thus, execution penalties due to update notification are incurred only where programmers consider necessary.

There are other possible strategies for identifying dirty items [AM95]. One option can be the use of a precompiler to generate code during compilation in order to include data in the set of mutated items. Similar to the strategy used by Datom, this mechanism has the disadvantage that incurs in execution penalties on subsequent updates to the item. A different possibility would be to use virtual memory support to detect updates dynamically, e.g. through the use of the Address Translation Unit (ATU) protection system. This method has the disadvantages of being approximate (i.e. multiple objects in the same protected region) and of interacting with other uses of the ATU protection system. Furthermore, it involves modifying the operation of the JVM. These issues increase the complexity of implementation.

If compared with Datom's ideology, which aims to provide programmers with complete control on the manipulation of persistent data, both strategies present the drawback of limiting the programmer's ability to decide when items should be pushed to disk. On the other hand, complete control over updates comes at the cost of inserting update notifications in the code manually. The prototype aims to simplify this task by making available the `update` method in all the classes derived from the `PersistentItem` class.

5.4.2 Main Processes

Incremental Data Loading

The incremental data loading strategy used in Datom frees programmers from manually moving data from disk to memory. Thus, the movement of persistent data between volatile memory and disk is performed in an automatic and incremental manner according to the access pattern of the application. This process is illustrated in Figure 5.7. All the references held by Composite Entities are resolved lazily. This implies that, when an application first loads a Composite Entity into memory, all the references to its members remain unresolved. It is only when there is an explicit request from an application to access any of its members that the Storage Manager evaluates if the requested item is already in memory or if it has to be fetched from disk.

The incremental loading mechanism detects light-weight *surrogates* in place of persistent items. A surrogate is a conventional object composed of an attribute to store a PID and of two methods to set and to get the PID value. The Storage Manager discovers whether an item is resident in memory by checking if the item that the surrogate represents has an entry in the cache of persistent items. If so, then the reference to the original surrogate is modified to point to the cached persistent item. If not, the object is requested from disk and the cache is updated to include the new persistent item. Datom makes use of inlined method calls to check the residency of items to reduce performance penalisation. The disadvantage of executing code even when the item is loaded in memory is counterbalanced by the fact that references take place exclusively in a selected group of persistent items (i.e. Composite Entities).

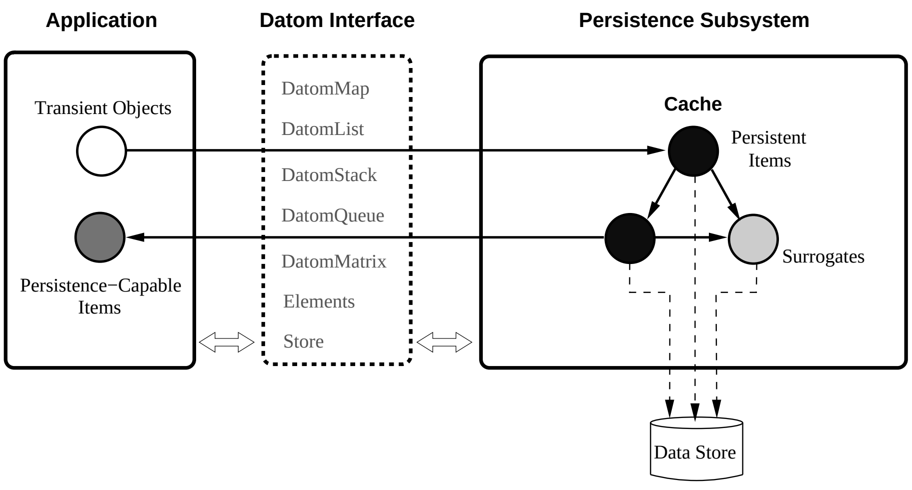


Figure 5.7: Management of persistent items in Datom.

Promotion to Persistence

Promotion to persistence and flushing of uncommitted updates is a process which is also accomplished automatically by Datom. It relieves programmers from manually moving data from memory to disk. When application programs request a stabilisation of a graph of persistence, any updated items should be pushed to disk. In addition, a copy on disk has to be created for all the new items that are reachable from the already-persistent items. Thus, a non persistent item is said to be promoted to persistence when, as part of the stabilisation of the graph of persistence, a copy of the item is created on disk for the first time.

The promotion algorithm involves, as a first step, discovering all the new items that are reachable from the set of mutated items. After that, the graph is traversed to find additional items reachable from the original group of new items. Finally, the group of items that is transferred to disk consists of all the mutated items together with the items to be promoted. It is not necessary to consider unmodified persistent items since they do not contain references to new items. Furthermore, unmodified items in the graph of persistence have an up-to-date copy on disk, and thus, they do not need to be stored again.

Two operations take place before transferring persistent items to disk. First, the Storage Manager assigns a PID and a valid connection to the items in the promotions set. Second, Composite Entities are *deswizzled* before being sent to disk. Deswizzling requires replacing persistent items' references to other persistent items with surrogates. Deswizzled Composite Entities and Elements are now sent to the Persistent Data Composer, which serialises and sends them to disk. To preserve the integrity of the graph of persistence, the stabilisation of the store takes place as an **atomic operation** supported by the facilities of the Physical Storage Subsystem. Finally, Composite Entities' references are swizzled to continue with the normal operation of the application. The Storage Manager requests active references from the cache of persistent items with this purpose.

Accordingly, a graph of persistence in an intermediate state may be composed of *three kind of objects*, as illustrated in Figure 5.7. Full-fledged persistent items are objects in memory that have a corresponding copy on disk. This implies that they have been promoted to persistence at some point as part of a previous stabilisation of the graph. Due to the incremental data loading strategy used by the Storage Manager, it is possible that some of the items in the graph can have the form of surrogates. The real identity of these objects will not be resolved until the application attempts to use them, at which point the surrogate is replaced with a persistent item fetched from disk.

The third type of objects in a graph of persistence corresponds to the items that are reachable from any of the full-fledged persistent items in the graph but for which a copy on disk is not yet available. These objects are persistence-capable items, which will be promoted to disk as part of the next stabilisation of the graph.

Cache Management

The Cache Manager controls a cache of persistent items. The main goal of this cache is to eliminate the negative impact on performance due to upward data translations, i.e. the conversion between on-disk and in-memory data representations. In contrast, the reduction of expensive disk accesses is better addressed by a cache of persistent items in its flat form at the Physical Storage Subsystem. Every time an application attempts to access a persistent item through a surrogate, the Persistent Manager requests this item to the Cache Manager. The Cache Manager checks if it holds a reference to the persistent item associated with the corresponding surrogate. If a persistent item reference is found for the requested surrogate, the Cache Manager passes this reference to the Storage Manager, which removes the reference to the surrogate in the Composite Entity for the newly acquired reference.

The eviction strategy enforced by the Policy Manager uses the facilities provided by the reference-object classes of the Java programming language, which support a limited degree of interaction with the garbage collector of the Java Virtual Machine (JVM). Persistent items stored in the cache are encapsulated in *soft reference* objects. A soft reference object maintains a reference to a persistent item in such a way that the persistent item may still be reclaimed by the garbage collector. An object is softly reachable if it is not strongly reachable but can be reached by traversing a soft reference. Accordingly, the cache of persistent items uses a hash table that associates PIDs to soft reference objects which in turn hold persistent items as referents.

A reference object is not cleared while its referent is strongly reachable from any other object in the application program. This guarantees that persistent items in the graph of persistence will be cached if they are referenced in an active graph of persistence. If a persistent item is removed from the graph of persistence and it is not reachable other than through its soft reference in the items' cache, this object may be garbage collected by the JVM and its entry safely removed from the cache.

Eviction occurs at the discretion of the garbage collector in a memory-sensitive way. The garbage collector might or might not reclaim the memory of this persistent item depending on how recently the persistent item was created or accessed. However, all the memory space held by soft references will be reclaimed before the JVM throws an `OutOfMemoryError`. Other eviction policies may be designed on top of reference objects according to the levels of reachability of persistent items.

Concurrency Control

It is possible that in many situations concurrently running threads in the application share access to the graph of persistence. Accordingly, Datom protects the integrity of the data in the graph by synchronising the simultaneous activities of different threads. The implementation of the Datom API supports concurrency control at the level of the operations offered in the public Datom interface.

Within a program, the code segments that access the same persistent item from separate, concurrent threads are considered to be critical sections. In Datom, a critical section corresponds to the members of the public interface of persistent items. Accordingly, they are protected with the *synchronized* Java primitive in their definition. The JVM associates a lock with every persistent item. Locks are acquired upon entering an item's critical section. The acquisition and release of a lock on a persistent item is done automatically and atomically by the Java run-time system. This ensures that race conditions cannot occur in the underlying implementation of the threads, thus ensuring data integrity.

All the methods in the Composite Entities' public interfaces use the synchronisation primitives available in the Java programming language to ensure that the graph is accessed in a controlled manner. Concurrency control on access to methods in Elements is normally related to the logic of the application. Therefore, it is a programmer's duty to define the synchronisation of Elements requiring concurrency control. However, Datom supports concurrency control on all public method access to avoid programming mistakes. Thus, all the operations in the interface defined using the automatic Element generator are protected with synchronised statements.

5.5 Persistent Data Composer Sublayer

The Persistent Data Composer is the second sublayer of the Persistence Subsystem. Its services are related with data translation of persistent items from its on disk and in-memory data formats. Persistent items are moved from memory to disk and vice versa upon request of the Storage Manager. The Storage Manager passes individual Composite Entities (always in their swizzled form) and Elements to the PDC Manager, which flattens the items in order to transfer them to disk. Translation is performed on one item at a time.

The costs associated to object storage techniques based on data translation contrast with storage strategies in which the on-disk format of the data matches exactly the in-memory format. However, data translation has the benefits of increasing data longevity, since its disk format is independent of the hardware architecture used to store the data. Figure 5.6 illustrates the main software modules involved in the data translation process. They are as follows:

- **PDC Manager.** The Persistent Data Composer (PDC) Manager receives requests from the Storage Manager to translate an object to its flat representation. This type of request occurs only when objects need to be sent to disk. In addition, the PDC Manager fetches persistent items from disk and translates them into their in-memory representation.
- **Data Binder Factory.** The Data Binder Factory is in charge of managing bindings for the different types of persistent items. A binding normally includes functionality for item marshalling and unmarshalling.
- **Translation Worker.** The Translation Worker is in charge of executing the transformation requests of the PDC Manager.

5.5.1 Persistent Items' Translation

Although simple Java serialisation techniques could be used for item marshalling, Datom uses the serialisation and binding services provided by the BIND API of the Berkeley DB JE [Lam05]. The utilities in this library enable applications to separate an object's class information from its actual contents. The benefits of this approach in comparison with the standard Java serialisation technique are various. First, there is a reduction in the size of the serialised items; this enables the Physical Storage Subsystem to make more efficient use not only of disk space but also of the memory employed to cache persistent items. Furthermore, the items' type information needs to be stored only once and can then be used for all the persistent items of the same type.

The translation process takes place only after an explicit request from the Storage Manager to recover or store persistent items. PIDs are used to identify the requested persistent items between the Storage Manager and the PDC Manager. In Datom, two different stores are part of the Physical Storage Subsystem. The Type Dictionary stores the class information of persistent items; it constitutes a mapping between class names and type information. The Data Store is used to save items' contents, which are retrieved by the PDC Manager through their corresponding PIDs.

The process of serialising a persistent item is as follows. The PDC Manager informs the Data Binder Factory when a new item needs to be serialised. The Data Binder Factory checks if the type information of the new persistent item already exists in the Type Dictionary. If it does, the Data Binder Factory fetches the type information and builds the binding for the corresponding type. If it does not exist, it creates an entry in the Type Dictionary for the new type information. The binding is then used by the PDC Manager to fire a new translation job through the Translation Worker.

The persistent items' bindings include functionality to convert objects to bytes and vice versa. When data needs to be recovered from disk the Data Binder Factory fetches type information from the Type Dictionary to construct a binding that is used to execute a new translation job.

To preserve items' integrity, the stabilisation of the store takes place as an atomic operation. This implies that all the persistent items that constitute the mutated and promotion sets have to be successfully transferred to disk or the stabilisation operation fails, leaving the data on disk in the state of the last successful stabilisation. The boundaries of the stabilisation operation for a particular graph of persistence are delimited by the Storage Manager and indicated to the PDC Manager. The latter element requests a transactional cursor to the Physical Storage Subsystem to safely transfer the group of updates to disk.

5.6 Physical Storage Subsystem

The main objective of the Datom's physical storage substrate is the provision of an efficient and secure mechanism for retrieval and storage of persistent items. The implementation of this functionality was considered to be out of the scope of the implementation of a Datom prototype. Thus, instead of building the physical storage layer from scratch, Datom employs Berkley DB Java Edition v2.0.54; a high-performance data management tool written in Java. An introduction to the implementation of Berkeley DB JE is presented in the following lines. A detailed analysis of this tool can be obtained from different sources [OBS99, Lam05].

The aims behind Berkeley DB JE are well aligned with the physical storage requirements of Datom. First, it fully eliminates the overheads introduced by a query language. Furthermore, it offers a programmatic record-oriented storage interface that stores data in an application's native format. And finally, it supports atomic updates on groups of disk transfers.

As mentioned before, Datom uses a strategy that separates the type information of items from their actual contents for persistence purposes. Accordingly, the Physical Storage Subsystem provides two different types of stores; they are defined in the following manner.

- **Type Dictionary.** This data repository is used to securely store all the type information of all the persistent items that are promoted to persistence. This store is managed by the Data Binder Factory.
- **Data Store.** The Data Store is in charge of storing the actual data of persistent items and is managed directly by the PDC Manager.

5.6.1 On-Disk Storage of Persistent Items

The Type Dictionary as well as the Data Store are built on top of the services provided by Berkeley DB JE. Figure 5.8 illustrates the relationships between the three main software modules that constitute the implementation of the Berkeley DB JE data management tool. They are defined as follows.

- **Log-based storage.** On-disk data management is performed using a log-based storage approach in order to maximise write performance.

- **Map.** Berkeley DB JE uses an in-memory concurrent B+tree structure acting as a map to the data on disk.
- **Cache Manager.** Finally, to deal with memory limitations the Berkeley DB JE enforces a least-recently eviction policy on the items in the map.

The communication between the PDC and the Physical Storage Subsystem is built around two calls of the programmatic interface of the Berkeley DB JE. The `get(PID)` call is used to obtain a persistent item from disk using its corresponding identification number. The `put(PID, byteArray)` call is used to store an item in its serialised form on disk that can later be retrieved using its PID.

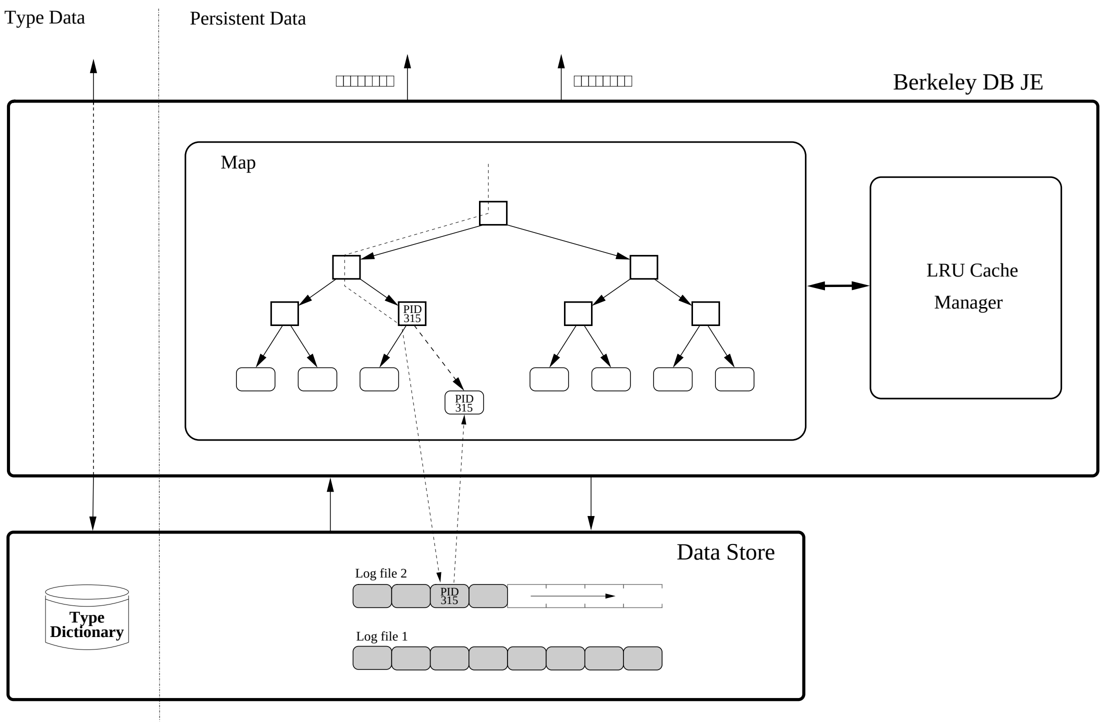


Figure 5.8: Main software modules of the Physical Storage Subsystem.

The Type Dictionary is built on top of the fundamental byte-based API of the Berkeley DB JE. Thus, the Type Dictionary is just a regular store whose records, containing class information, are managed and stored using the same structures employed for persistent items' data. For this reason, the figure simplifies the representation of the Data Dictionary. The software processes of the store for persistent items' data, shown on the right hand side of Figure 5.8, are explained in the following lines; they also apply to the operation of the Data Dictionary.

The Berkeley DB JE implementation uses a log-based storage approach for on-disk data management in order to maximise write performance. Write operations append serialised persistent items to a log file until it reaches the maximum allowed size (10 MB by default); new empty log files will be subsequently created as write operations fill the log file in use. Updates of old persistent items are a two-step procedure in which the updated data is appended to the current log and the mapping to the old record is changed to point to the new one.

Deleting an old persistent item only involves updating the map to indicate that there is no data for that PID. A copying garbage collector running as a background daemon is activated periodically by the system, looking for log files that predominantly contain stale data. This cleaning process copies all non stale records to the current log file, and then deletes the old log file from the file system. The employment of a log-structured storage approach is based on the hypothesis that the ever-increasing memory sizes on modern computers lead to disk IO becoming write-heavy because disk reads can be almost always satisfied from memory cache. This view matches the persistent data access pattern that the Datom API favours, i.e. to keep persistent items in memory until they need to be saved.

To manage the in-memory cache of records, the Berkeley DB JE implementation adds concurrency mechanisms to a B+tree based structure. It holds only PIDs at internal nodes and data at leaf nodes. Figure 5.8 also illustrates the process of loading persistent items from the log files. When searching for a persistent item that exists in the store but that has not yet an entry in the tree, the traversal procedure will eventually reach a dead end node which contains a pointer to the data on disk. The persistent item record is loaded from disk and included in the active map. If the record does not exist for a given PID, no mapping through the tree will be found for that PID.

The tree structure acts as a cache of serialised persistent items that saves expensive disk accesses for frequently used data. The cache manager in Berkeley DB JE approximates the usage information of each node in the tree by keeping track of how recently each node was used and enforces a least-recently used (LRU) eviction policy. It makes use of a LRU list of the nodes in the map whose LRU time stamp is updated every time a node is traversed. This list is traversed with the support of a sliding window and a priority queue that discriminates items based on the LRU time stamp of the node [Lam05].

5.7 Summary

This chapter presented the implementation of the Datom API. It began by defining the concept of selective reachability, the persistence model supported by the prototype. Then, it discussed the high-level architecture of Datom. Its layered design favours a separation of concerns that, ultimately, simplify the modification of the internal mechanisms of each layer. Finally, this chapter elaborated on the infrastructure that supports not only the data model of the API but also its persistence model.

Chapter 6

Evaluation

In this chapter I present a comprehensive evaluation of the Datom API. In Section 6.1, I report on the experience of migrating two file-based applications to the persistent abstractions offered by the Datom API. Then, in Section 6.2, I assess operational aspects of the implementation of the API including performance, scalability, and latency. In Section 6.3, I evaluate usability aspects of the Datom API from a programmer point of view; the results of this study clearly indicate to which type of developer the API may be suitable and why. Finally, in Section 6.4, I present a summary of the key contributions of this chapter.

6.1 Practical Experiences with the Datom API

This section reports on the impact of migrating applications to the Datom abstractions. The goal of this study is twofold. First, to establish a baseline for the typical transformations and form of application code that employs the Datom API. Second, to evaluate the feasibility and advantages of using the Datom API instead of other common storage facilities to manage structure and type in file data. With this purpose, two applications were ported: the first application uses the basic services of the file API, while the second employs the XML DOM API to manage structured data. The results obtained from this study suggest that the changes in the application agree with good software engineering practices and lead to the construction of software of improved quality.

6.1.1 Source Code Measurements in Context

Measuring the change incurred in applications' source code caused by the migration process is important for various reasons. First, they are correlated with the amount of effort needed to convert applications to Datom abstractions. Second, they are a clear indication of the consequences of adopting a more abstract API as an underlying storage infrastructure. Finally, by analysing the final form of the source code and by comparing it with its original version, it is possible to evaluate whether the storage system API may represent a better software tool for programmers in the given context. However, it is necessary to keep in mind that applications created from the start using the Datom API may exhibit a neater integration with its facilities.

The source code metrics collected in this study aim to reveal if the adoption of the Datom API improves the way in which programmers deal with persistence-related tasks. Additionally, they are also useful to indicate to what extent the new API abstractions modify the interaction of programmers with persistent data.

The relationship between internal software attributes collected by source code metrics and external attributes such as *understandability*, *usability*, or *maintainability* have been suggested by a number of studies. In general, it is recognised that reducing the number of lines of code has a positive impact on the understandability and maintainability of the software [GSAW98]. Maintaining large applications requires a lot of effort, and big code bases are vulnerable to error. The Datom API would likely augment the quality of the application if it is able to reduce the size of applications, as long as the same functionality is provided.

However, reducing the lines of code is not fully indicative of an improved software product. It is also necessary to analyse how the code is transformed and what is the impact on the morphology of the application caused by the introduction of a different technology. This involves analysing how pervasive the Datom API is and what the compromises in its employment are.

The use of the Datom API is expected to reduce the *size* of the source code and to hide low-level interactions with the file system. It should augment the degree of *abstraction* that developers are able to manipulate through the storage system API in order to improve code understandability and maintainability. Additionally, it should greatly *simplify* data retrieving tasks by making code clear and simple. With the Datom API, manual parsing and serialisation of application abstractions should be fully eliminated.

Source Code Measurements Explained

Instead of building a tool from scratch or counting source code modifications manually, the Persistent Code Measurement Tool (PCMT) [Gri97] has been used to gather code metrics. PCMT is a general purpose measurement tool that can be easily adapted to fulfil the requirements of a counting strategy suitable for the Datom API. As mentioned before, one of the main goals of collecting source code metrics is to attempt to establish the relationship between internal and external software attributes.

In this respect, the metrics produced by PCMT provide a rich and fine-grained description of the effect of adopting the Datom API at the source code level from which the relationship between internal and external software attributes can be investigated. PCMT not only quantifies the proportion of lines and classes that explicitly manipulate the persistent abstractions provided by the Datom API but also collects measurements on the distribution and clustering of persistent lines of code in the application. Accordingly, it measures the persistence impact ratio and produces statistics at class level for the whole application. Moreover, the PCMT metrics enable one to track and quantify the presence of the graph of persistence in the application, an issue that is directly related to the mental effort needed by the programmer to manipulate persistent data through the abstractions offered by the Datom API.

This detailed analysis is accomplished by using a keyword file that can be easily replaced to characterise the usage of the Datom API in different applications. Persistent lines of code are determined by a set of keywords which act as identifiers of persistent code. PCMT parses Java source code and tracks the use of persistent keywords to identify instances of objects referring to persistent code. Given that the persistence model of the Datom API is based on selective reachability, i.e. all the persistent elements in an application can be distinguished by their type, this method of identifying persistent code produces a complete characterisation of the use of the Datom API.

Furthermore, the PCMT metrics provide a detailed characterisation that indicates where and how an application is transformed by the adoption of the Datom API. The PCMT's counting strategy is based on production rules rather than textual lines of code; it is a strategy that produces an accurate measure since it eliminates the effect of different programming styles. This counting strategy is needed to properly compare the original versions of the applications with their corresponding portings since they have been developed by different programmers.

The source code metrics reported by the PCMT tool are shown in Table 6.1.

Metric	Description
LOC	Lines of code
PLOC	Lines of code referring to persistent elements
PIROL	Persistence impact ratio on lines of code (PIROL=PLOC/LOC)
NOC	Number of classes
NOPC	Number of persistence affected classes
PIROC	Persistence impact ratio on classes (PIROC=NOPC/NOC)
ADPLOC	Average distance between persistence affected lines of code
PIBPLOC	Length of the interval between the first and last PLOC

Table 6.1: Summary of source code metrics reported.

- **LOC.** PCMT counts lines of code (LOC) as productions rather than simply textual lines; this helps to minimise the impact of personal programming styles. Accordingly, a line of code is defined as a compiler directive, a declaration, or an executable ending with "`;`". Class and methods headers are counted as lines as well. Comments in the source code are ignored.

- **PLOC.** Lines of code referring to persistent objects are considered those in which at least one class or interface from the keyword file is used. Nested classes' LOCs or PLOCs are counted as being part of the outer class and not as part of the inner classes. As a way of example, the following line of code is counted as a PLOC considering that the `DatomMap` class has been included in the keyword file:

  ```
  DatomMap dm = new DatomMap();
  ```

  Additionally, objects introduced by persistent declarations are also tracked and counted as a PLOC every time they appear in a line of code. Following the example above, the next statement will also be counted as a PLOC since the `dm` variable is of type `DatomMap`, which happened to be a persistent keyword:

  ```
  int dmSize = dm.size();
  ```

- **NOC.** The number of classes or interfaces in the application are reported by the NOC metric. Internal classes are considered to belong to the outer class and are not counted individually.

- **NOPC.** Persistence affected classes are considered those that contain at least one PLOC. Accordingly, the number of persistence affected classes (NOPC) is the sum of all the classes that contain persistent lines of code.
- **PIROL.** The persistence impact ratio on lines of code per class (or interface) is calculated by dividing the PLOC metric by the corresponding LOC metric.
- **PIROC.** Similar to the PIROL metric but it quantifies impact ratio at a class level. It is calculated by dividing the number of classes (NOC) by the number of persistence affected classes (NOPC).
- **ADPLOC.** PCMT captures the average distance between the persistence affected lines of code in a given class in the ADPLOC metric.
- **PIBPLOC.** The interval between the first and last persistence affected line of code is accounted in the PIBPLOC metric which is reported as the proportion of the total number of lines. Both the PIBPLOC and the ADPLOC metrics estimate the distribution of persistent code within a class.

6.1.2 The Applications Ported

Two applications were chosen to be ported to the Datom API: Bibkeeper and Gradebook. The results of both portings are discussed in turn in Sections 6.1.3 and 6.1.4. Next, the features of both applications are presented.

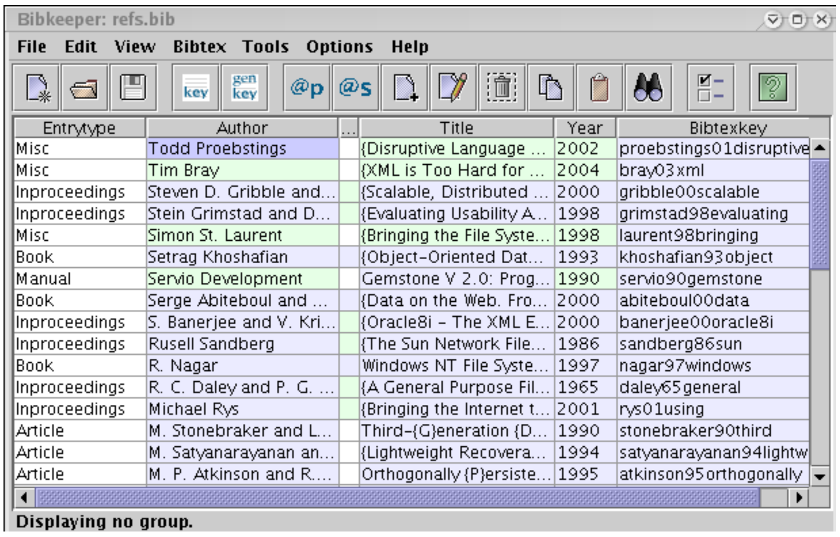
Bibkeeper

The first application that was migrated to run on top of Datom abstractions is called *Bibkeeper* [BIB05]; it is written in Java and its main function is to manage bibliographical references written in BibTeX format[1]. Bibkeeper presents a user-friendly graphical interface as shown in Figure 6.1. This application was chosen as a reasonable candidate for migration after considering important features intrinsic to the group of applications for which the Datom API was designed.

Bibkeeper's persistent data is fairly rich in structure and type. Its source code is a representative example of the type of functionality needed in applications running on top of the flat file paradigm. In this kind of application, all the

[1]`http://en.wikipedia.org/wiki/BibTeX`

high-level abstractions manipulated at run time are lost inside file formats. Consequently, its data layout on disk follows a predetermined organisation, although not directly accessible through the storage system API. It makes extensive use of parsing libraries to build the data layout on disk into programming abstractions and vice versa. Finally, it represents a potential enabler of data sharing and collaborative work in which concurrent units of work can be directly associated to individual entries of the database.



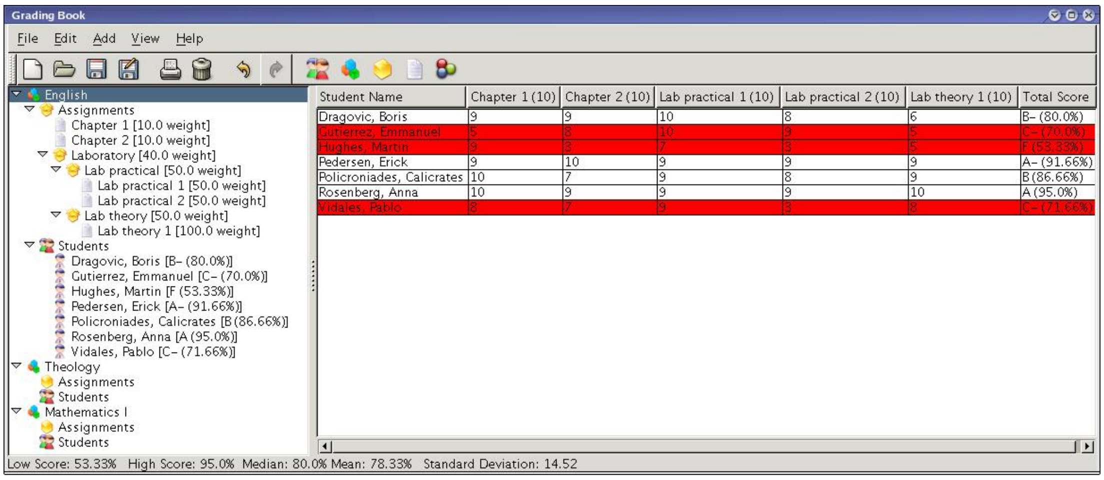
Entrytype	Author	...	Title	Year	Bibtexkey
Misc	Todd Proebstings		{Disruptive Language ...	2002	proebstings01disruptive
Misc	Tim Bray		{XML is Too Hard for ...	2004	bray03xml
Inproceedings	Steven D. Gribble and...		{Scalable, Distributed ...	2000	gribble00scalable
Inproceedings	Stein Grimstad and D...		{Evaluating Usability A...	1998	grimstad98evaluating
Misc	Simon St. Laurent		{Bringing the File Syste...	1998	laurent98bringing
Book	Setrag Khoshafian		{Object-Oriented Dat...	1993	khoshafian93object
Manual	Servio Development		Gemstone V 2.0: Prog...	1990	servio90gemstone
Book	Serge Abiteboul and ...		{Data on the Web. Fro...	2000	abiteboul00data
Inproceedings	S. Banerjee and V. Kri...		{Oracle8i - The XML E...	2000	banerjee00oracle8i
Inproceedings	Rusell Sandberg		{The Sun Network File...	1986	sandberg86sun
Book	R. Nagar		Windows NT File Syste...	1997	nagar97windows
Inproceedings	R. C. Daley and P. G. ...		{A General Purpose Fil...	1965	daley65general
Inproceedings	Michael Rys		{Bringing the Internet t...	2001	rys01using
Article	M. Stonebraker and L...		Third-{G}eneration {D...	1990	stonebraker90third
Article	M. Satyanarayanan an...		{Lightweight Recovera...	1994	satyanarayanan94lightw
Article	M. P. Atkinson and R....		Orthogonally {P}ersiste...	1995	atkinson95orthogonally

Figure 6.1: Screen shot of Bibkeeper managing a group of bibliographical references.

Gradebook

The second application ported to the Datom API is called *Gradebook* [GRA05]. Its core functionality is the management of multiple courses, students, and assignments. It supports the creation of arbitrarily nested and weighted assignments, different types of summaries, statistics, and scatterplots, as well as different data views. Gradebook presents a graphical interface to users as that shown in Figure 6.2. However, the main interest on porting this application is that it uses a persistent layer based on an XML file format that is recovered and updated with the XML DOM API [HHW$^+$04].

The programming patterns that are observed in the majority of applications using the XML DOM API can be also identified in Gradebook. The XML DOM API recovers structure in XML files and creates a tree-like representation of this structure in memory. Programs operate on this in-memory tree to recover



```
English
  Assignments
    Chapter 1 [10.0 weight]
    Chapter 2 [10.0 weight]
    Laboratory [40.0 weight]
      Lab practical [50.0 weight]
        Lab practical 1 [50.0 weight]
        Lab practical 2 [50.0 weight]
      Lab theory [50.0 weight]
        Lab theory 1 [100.0 weight]
  Students
    Dragovic, Boris [B– (80.0%)]
    Gutierrez, Emmanuel [C– (70.0%)]
    Hughes, Martin [F (53.33%)]
    Pedersen, Erick [A– (91.66%)]
    Policroniades, Calicrates [B (86.66%)]
    Rosenberg, Anna [A (95.0%)]
    Vidales, Pablo [C– (71.66%)]
Theology
  Assignments
  Students
Mathematics I
  Assignments
  Students
```

Student Name	Chapter 1 (10)	Chapter 2 (10)	Lab practical 1 (10)	Lab practical 2 (10)	Lab theory 1 (10)	Total Score
Dragovic, Boris	9	9	10	8	6	B– (80.0%)
Gutierrez, Emmanuel	5	8	10	9	5	C– (70.0%)
Hughes, Martin	9	3	7	3	5	F (53.33%)
Pedersen, Erick	9	10	9	9	9	A– (91.66%)
Policroniades, Calicrates	10	7	9	8	9	B (86.66%)
Rosenberg, Anna	10	9	9	9	10	A (95.0%)
Vidales, Pablo	8	7	9	3	8	C– (71.66%)

```
Low Score: 53.33%   High Score: 95.0%  Median: 80.0% Mean: 78.33%   Standard Deviation: 14.52
```

Figure 6.2: Screen shot of Gradebook managing a group of courses.

application data and to place it in application-specific data abstractions. In the same way, when data has to be saved, the application program needs to provide a set of routines to recreate XML data out of the application's run-time abstractions.

One of the problems with this approach is that applications do not operate directly on persistent abstractions. Instead, an intermediate data representation has to be employed which exhibits completely different data access semantics to those used by the application. As a consequence, the persistence mismatch problem remains and programmers are forced to maintain an explicit mapping between persistent and run-time data abstractions.

6.1.3 Removing the File API

Addition of Datom Types in Bibkeeper

The work to migrate Bibkeeper to the Datom API consisted of different tasks. As a first step, it was necessary to understand the logic and structure of the application in order to identify the points where persistent code was being used. After that, Java IO calls and volatile versions of objects in charge of managing persistent data were replaced with the persistent abstractions provided by the Datom API where appropriate. And finally, the ported code was debugged.

Porting Bibkeeper to the Datom API affected the morphology of the application in a number of ways. In the file-based Bibkeeper, once persistent data has been parsed, it is grouped into different objects representing mainly *lists* and *maps*; these objects are spread in various classes of the application but principally managed in the `BibtexDatabase` class.

The final layout of the whole graph of persistence for a Bibkeeper database is illustrated in Figure 6.3. In order to port the application to the Datom API, transient versions of the original map and list objects were replaced with `DatomMap` and `DatomList` abstractions which provided a natural and straightforward mapping for these objects. Then, they were grouped together in a `DatomMapRoot` object managed by the corresponding `BibtexDatabase` object. Thus, the graph of persistence for a specific database became rooted in its corresponding `DatomMapRoot`. The procedural behaviour of the application was left unchanged.

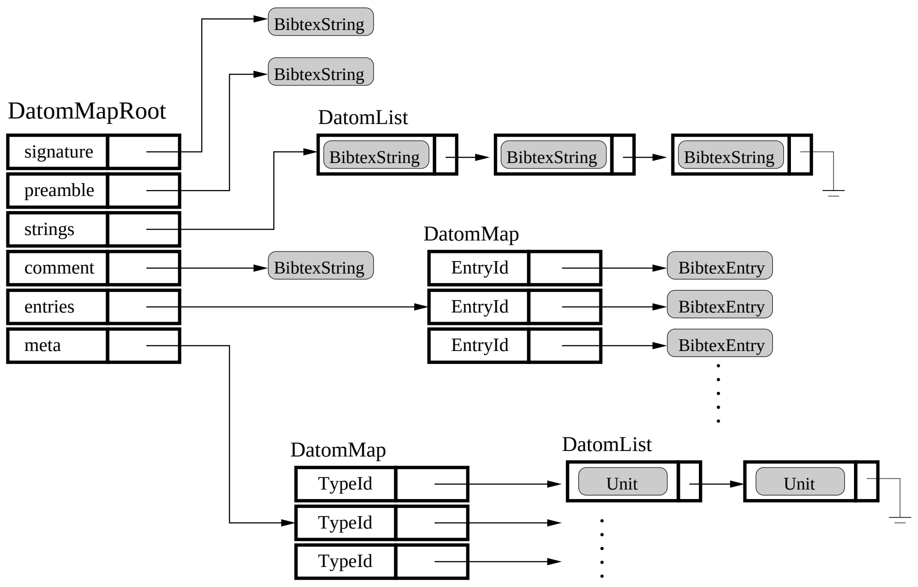


Figure 6.3: Layout of the graph of persistence in the ported version of Bibkeeper.

Metrics Obtained

Bibkeeper is a medium-sized application comprising 6002 lines of code of which 208 make explicit use of the persistence facilities provided by the Java IO libraries as shown in Table 6.2. The values in this table were obtained using the keyword files shown in Appendices A.1 and A.2 for the Datom-based and file-based versions of the application, respectively. The keyword file for the Datom-based version corresponds to all the classes and interfaces that are part of the Datom API plus the persistent data types of the application. The keyword file used for the file-based version includes the classes and interfaces that are part of the Java IO API, which is the persistent technology used by the original Bibkeeper.

Application Name	**LOC**	**PLOC**	**NOC**	**NOPC**
File-based Bibkeeper	6002	208	81	15
Datom-based Bibkeeper	5570	469	76	40

Table 6.2: High-level comparison of the file-based Bibkeeper against its Datom-based version.

Table 6.2 reports the high-level differences between both version of the applications. It shows a *reduction* in both the lines of code (LOC) and the number of classes (NOC) for the Datom-based Bibkeeper; it was possible to eliminate 432 LOCs contained in the 5 classes that were totally removed together with other lines of code scattered in the rest of the application. This reduction in LOCs represents a shrinkage of approximately 7.2% of the total size of the original Bibkeeper.

However, if compared with the values reported by the original version of the application the number of persistent lines of code (PLOC) and the total of persistent classes (NOPC) increased by approximately 125% and 166%, respectively. This is explained by the fact that the PLOCs reported for the ported version of Bibkeeper truly represent the places in which persistent abstractions are manipulated. Every PLOC counted in the Datom-based Bibkeeper corresponds to either explicit calls to persistent functionality or to implicit invocation of functions in the objects that are members of the graph of persistence. On the contrary, in the file-based version of the application it is correct to identify as PLOCs only those lines in which explicit persistent activity is invoked. Therefore, run-time abstractions containing the processed file data are not considered as persistent elements and calls to them are not accounted as PLOCs.

Detailed metrics for the original version of the application are shown in Table 6.3, which is ordered by the number of persistent lines of code (PLOC). Non-persistent classes have not been included in the table for brevity. Persistent code of the file-based version of Bibkeeper groups persistent code in classes that are in charge of managing file content, parsing it, and placing it in application

Class Name	PLOC	PIROL(%)	LOC	ADPLOC	PIBPLOC(%)
BibtexBaseFrame	52	5.30	980	18	97
FileActions	31	46.90	66	2	83
BibtexParser	25	13.70	182	7	97
FileLoader	21	11.20	186	5	59
MetaData	14	22.50	62	2	54
OpenofficeTextExport	14	17.90	78	3	48
BibtexEntry	12	12.00	100	2	29
ExampleFileFilter	10	14.70	68	4	54
EntryTypeForm	8	1.60	474	50	75
Bibkeeper	7	28.00	25	1	36
TransferableBibtexEntry	5	19.20	26	2	26
BibtexParserTest	4	26.60	15	2	46
HelpContent	2	4.50	44	34	79
DragNDropManager	2	2.70	72	27	38
SaveSpecialDialog	1	1.30	76	0	1
Totals	**208**	—	**2454**	—	—

Table 6.3: Persistent code measurements of the original version of Bibkeeper.

abstractions. As expected, the highest persistence impact ratio on lines of code (PIROL) was obtained for the class `FileActions` which contains code to save application data to a file.

Other files in the application that group fundamental code for persistence purposes are `BibtexParser` and `FileLoader`. However, the persistent impact ratios on lines of code (PIROL) for these two files do not reach the top values. This may be explained by considering that after an invocation to the file API there is usually work to do with the recovered data such as parsing or placing it in the appropriate application variables. Thus, persistent lines of code are not placed together in the application code as corroborated by values in the average distance between persistent lines of code (ADPLOC) column.

Table 6.4 shows the breakdown of the persistence affected classes in the Datom-based version of Bibkeeper; non-persistent classes have been omitted for brevity. Two main observations can be drawn from the data included in the table. First, classes with the highest number of persistent lines of code (PLOC) manipulate persistent data early in the beginning of the file until positions close to the end of it as indicated by the intervals between the first and last persistence affected lines of code (PIBPLOC). Second, in classes with the highest persistence impact ratio on lines of code (PIROL) values (i.e. $\geq 33\%$) the average distance between persistence affected lines of code (ADPLOC) is always 1. This suggests that there is a considerable amount of locality of PLOCs for those files that heavily use the Datom API.

Class Name	PLOC	PIROL(%)	LOC	ADPLOC	PIBPLOC(%)
BibtexBaseFrame	86	8.20	1047	12	97
BibtexDatabase	66	61.10	108	1	92
SimpleSearchRule	31	32.60	95	2	90
BibtexEntryType	28	13.70	204	6	82
EntryTypeForm	25	5.70	432	17	95
BibtexEntryTest	24	68.50	35	1	80
EntryTableModel	15	12.80	117	5	64
BibtexDatabaseTest	13	30.20	43	2	72
OpenofficeTextExport	13	16.60	78	2	44
RegExpRule	11	27.50	40	2	75
BookLabelRule	10	47.60	21	1	85
LabelMaker	10	47.60	21	1	61
ArticleLabelRule	9	27.20	33	3	90
InproceedingsLabelRule	9	27.20	33	3	90
EntryComparator	9	20.40	44	3	56
SaveException	8	66.60	12	1	91
UndoableInsertEntry	8	26.60	30	3	76
UndoableRemoveEntry	8	26.60	30	2	70
DragNDropManager	7	9.70	72	4	31
StringDialog	7	2.80	246	11	27
QuickSearchRule	6	21.40	28	4	82
UndoableStringChange	6	19.30	31	5	83
BibtexEntry	6	5.20	114	20	91
CrossRefEntryComparator	5	27.70	18	1	44
EntrySorter	5	23.80	21	3	71
UndoableFieldChange	5	19.20	26	5	88
MetaData	5	18.50	27	3	51
UndoableInsertString	4	14.80	27	6	70
UndoableRemoveString	4	14.80	27	4	55
DatabaseSearch	4	13.30	30	1	20
BibtexString	3	13.60	22	2	27
Unit	3	11.50	26	1	11
EntryTable	3	2.40	122	3	5
DefaultLabelRule	2	50.00	4	1	50
FieldChangeListener	2	16.60	12	2	25
AndOrSearchRuleSet	2	13.30	15	4	33
SearchRuleSet	2	12.50	16	5	75
UndoableMoveString	2	6.20	32	3	12
Util	2	2.70	73	4	6
SidePaneManager	1	1.80	53	0	1
Totals	**469**	—	**3465**	—	—

Table 6.4: Persistent code measurements of the ported version of Bibkeeper.

There are some notable differences between the source code metrics of both applications. First, for the file-based Bibkeeper the persistence impact ratios at the class (PIROC) and line level (i.e. PIROL on the total size of the application) are 18.5% and 3.4%, respectively. In contrast, the ported version of Bibkeeper presented values of 52% and 8.4% for the same metrics. Accordingly, the persistence impact ratio on lines of code per class in the Datom-based Bibkeeper are consistently larger than those in the original version. To an important degree, these increments take place because file data is not tracked once resident in transient objects.

These values stress that although a small portion of code in an application could be explicitly dedicated to persistent operations (i.e. retrieve and store), normally the run-time objects that contain and consequently modify persistent data will be manipulated at many places in the source code. Therefore, having larger impact ratios in the Datom-based Bibkeeper should not be considered a disadvantage of the API in itself because these values are providing an accurate characterisation of where in the application persistent data is manipulated.

6.1.4 Removing the XML DOM API

Addition of Datom Types in Gradebook

In order to migrate Gradebook, all the functions performed through calls to objects of the XML DOM API were replaced with appropriate Datom abstractions. In general, the DOM-based version of Gradebook reads data from XML files to a DOM tree and then processes the elements at each tree node through calls of the type `Element.getChildren()`, which returns a list of other XML elements. The returned DOM elements are processed sequentially following *list* semantics; if appropriate, the content of these elements is mapped to application's abstractions or processed once more as lists of children.

The graph of persistence created for the Datom-based version of Gradebook is shown in Figure 6.4. `DatomList` items were used to model in a more precise way the *list-based* data access semantics of the application. In addition, these lists were rooted to `DatomMap` items when the original data access was dictated by a keyword such as the set of courses and the classification of information corresponding to each course.

Furthermore, corresponding Datom Elements were used to hold the application's persistent data. However, instead of fully replacing the application's data objects with these Datom Elements, a successful method used previously in the Bibkeeper porting, a more suitable strategy for the code organisation of Gradebook was to include references to Datom Elements inside the original application data objects. This decision was taken for various reasons. First, the correct operation of Gradebook highly depends on the graphical elements used by the application; these elements are tightly coupled with the objects in charge of managing persistent data and with the inheritance model of the application. Since the current prototype of the Datom API relies on inheritance to enable persistence, and the Java programming language limits the inheritance model to a single base class, it was infeasible to extend these objects to become persistence capable.

Furthermore, separating persistent data elements from the graphical utilities of the Java programming language seems to be the correct strategy even in

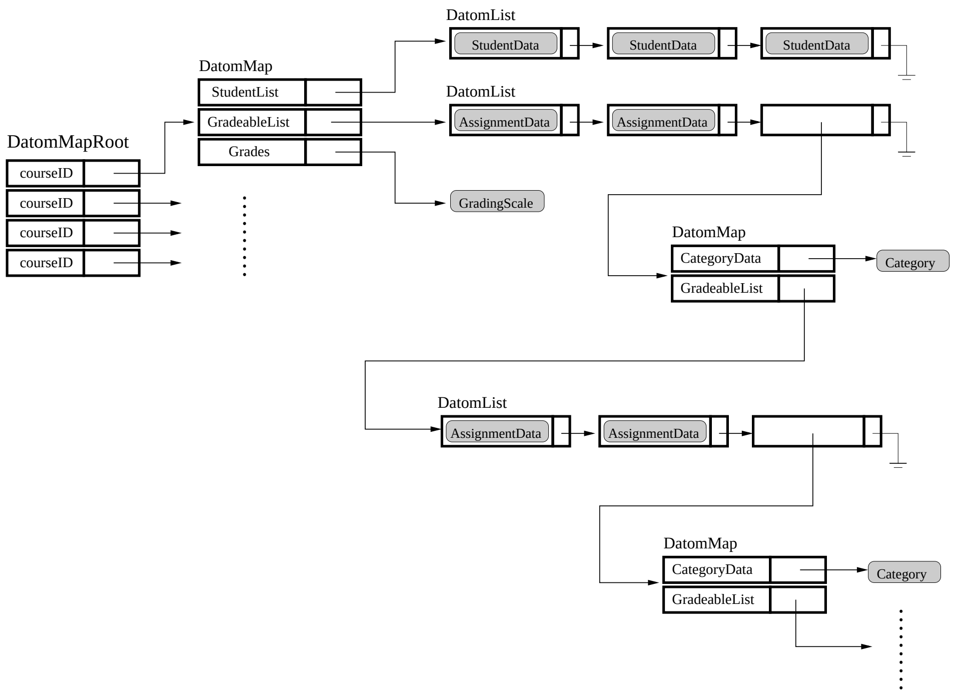


Figure 6.4: Layout of the graph of persistence in the ported version of Gradebook.

environments supporting orthogonal persistence due to the dependencies of the graphical utilities with external state [JA98]. A different porting strategy would have altered the structure of the application to the point in which any source code comparison would be meaningless.

Metrics Obtained

Table 6.5 reports the overall impact on source code caused by migrating the Gradebook application. The values reported in this table were collected using the keyword files shown in Appendices A.3 and A.4 for the Datom and DOM persistent keywords, respectively. In its original version, Gradebook has a total of 6161 lines of code of which 111 are explicitly dedicated to manipulate persistent data through the facilities provided by the DOM API. In contrast, the ported version of the application reported a value of 6219 lines of code representing an increase of approximately 1% of the total size of the application. The number of lines of code referring to persistent elements also augmented 26% with respect to the original value. Following the same tendency, the total number of classes in the application also increased by 3.

The persistence impact ratio at the class (PIROC) and line level (PIROL) for the original version of Gradebook are 1.9% and 1.8%, respectively. In contrast, its ported version presented values of 14.81% and 2.25% for the same categories. The wide difference in the persistence impact ratio at the class level is mainly due to the line counting strategy used by the PCMT tool in which file data, once resident in the application's abstractions, is not counted as persistence related.

Application Name	**LOC**	**PLOC**	**NOC**	**NOPC**
DOM-based Gradebook	6161	111	105	2
Datom-based Gradebook	6219	140	108	16

Table 6.5: High-level comparison of the DOM-based Gradebook against its Datom-based version.

The expected increase in the number of persistent lines of code and the number of persistent classes is explained by the fact that, in the ported version of the application, persistent lines of code include both explicit calls to persistent functionality or to invocations of the functions of those objects that are members of the graph of persistence and accordingly, are accessed during the normal operation of the application. Furthermore, it was necessary to add three new classes for the Datom Elements holding the application's persistent data.

A detailed analysis of persistent code shows the reasons for the increase in the size of the application. Table 6.6 reports code metrics for the original version of Gradebook ordered by persistent lines of code (PLOC); non-persistent classes have been omitted for brevity purposes. The DOM-based Gradebook aggregates persistent code in two classes only, i.e. `SchemaSaver` and `SchemaLoader`. Both classes exhibit a elevated persistence impact ratios on lines of code (PIROL) and high densities of persistent lines of code (ADPLOC) since their only purpose in the applications is to read and save data to the XML file.

Class Name	**PLOC**	**PIROL(%)**	**LOC**	**ADPLOC**	**PIBPLOC(%)**
SchemaSaver	62	63.20	98	1	98
SchemaLoader	49	42.20	116	2	98
Totals	**111**	—	**214**	—	—

Table 6.6: Persistent code measurements of the original version of Gradebook.

Table 6.7 shows detailed metrics for the Datom-based Gradebook. Following a similar pattern to that observed in the Bibkeeper porting, the table indicates an increment in the number of persistent classes and in the number of persistent lines of code (PLOC). In general, these increments occur since the persistent lines of code reported account not only for the code used to retrieve and save persistent data but also they include the lines in the application in which Datom abstractions are used to manipulate persistent data through the normal execution of the application.

Although a *reduction* in the code needed to retrieve and restore data from persistent storage was observed, this decrement was surpassed by the code needed to adapt other aspects of the application. The comparison of the two persistent classes in the original Gradebook against their corresponding new versions shows a reduction from 116 to 93 lines of code for the `SchemaLoader` class and from 98 to 16 for the `SchemaSaver` class. However, the size of the application increased mainly for two reasons. First, 100 lines of code were added by three new classes, i.e. `PerStudentData`, `PerCompositeData`, and `PerAssignmentData`. Second, as has already been mentioned, these persistent abstractions were incorporated in the application as attributes of the original classes, a situation that created a level of indirection in the code and ultimately, an increase in the total size of the application.

Class Name	**PLOC**	**PIROL(%)**	**LOC**	**ADPLOC**	**PIBPLOC(%)**
SchemaLoader	40	43.00	93	2	77
Student	22	32.30	68	2	88
GradeableElement	18	29.50	61	3	96
Course	16	14.10	113	7	92
GradebookSystem	12	27.90	43	3	79
GradeableComposite	9	23.00	39	2	46
GradingModelProperties	5	4.50	111	23	84
SchemaSaver	3	18.70	16	2	31
PerStudentData	3	5.30	56	4	17
PerCompositeData	2	12.50	16	3	25
GradingModel	2	8.00	25	3	16
PerAssignmentData	2	7.10	28	5	21
GradebookFileManager	2	4.50	44	3	9
GradebookScreen	2	0.80	232	2	1
StudentTableModel	1	1.60	61	0	1
GradebookTableModel	1	1.30	72	0	1
Totals	**140**	—	**1078**	—	—

Table 6.7: Persistent code measurements of the ported version of Gradebook.

6.1.5 Discussion

The results and metrics presented through this section provided an insight on the potential source code transformations experienced by applications ported to the

Datom API. Additionally, they show the feasibility of migrating file-based applications to the set of persistent abstractions provided by the new API. Although porting a larger number of applications would provide a more comprehensive understanding of the corner cases that may arise as a result of the migration process, the applications that have been ported are representative of the type of code used by a considerable number of applications running on top of the file API. The conclusions drawn from the portings corroborate the positive impact of adopting a more abstract API as an underlying storage infrastructure. The analysis and comparison of the original programs against their corresponding portings suggest an improvement in software quality.

The Datom API **reduced the size** of source code devoted to persistence related tasks. The extent of these reductions was determined by the original API used by the application and to the amount of code needed to adapt the original source code to the Datom API and programming model. Datom abstractions enabled the elimination of code in one of the most tedious and error-prone coding tasks, those related to low-level interactions with the persistent layer. Code used to parse, serialise, and manually build run-time abstractions disappeared from the application. For the most part, this kind of code simply wastes programmers effort and draws their attention away from application-specific functionality.

The use of the **high-level abstractions** provided by the Datom API represented a step ahead in the manipulation of persistent data for both portings. One of the core assumptions in the design of the API is that applications running on top of the file abstraction commonly use collections to manage persistent data at run-time. Since this assumption was valid for Bibkeeper, the migration from the original volatile objects that used to hold persistent data to those provided by the Datom API was fairly natural causing a considerable impact to the morphology of the application.

The Gradebook porting also experienced an improvement in its persistence related code by the use of appropriate persistent abstractions. Thus, the tree-view of the DOM API was replaced with the map and list abstractions available in the Datom API, which represent more adequate abstractions for the data access of the application. Thus, the Datom API positively affected the understandability of persistent code in both applications through the use of more suitable persistent abstractions.

Finally, code was **simplified** by making persistent data manipulation implicit in the abstractions provided by the Datom API. Instead of recovering persistent data to many unrelated objects, and then, restoring it by gathering important data from different sources, the Datom API makes possible to group persistent data in a graph of persistent abstractions. Persistent data manipulation through a graph of persistence made persistent code simpler and self-describing in both applications. As a consequence, understandability as well as maintainability features of the applications were improved.

The explicitness of type in the manipulation of persistent abstractions provided by the Datom API made the graph of persistence self-evident in the source code. This is an advantage in code understandability and assists developers to accurately know where potentially-persistent data elements are accessed and modified, which ultimately reduces programming mistakes that may push incorrect data to the persistent store. In contrast, in a file-based approach programmers do not have the capacity to directly identify where code operations affect the graph of persistence. The same argument can be extended to orthogonally persistent database systems in which failing to accurately determine the membership of an object in the graph of persistence might easily lead to pushing a potentially large number of unwanted objects into the persistent store.

Applications written from scratch on top of the facilities provided by the Datom API should exhibit a tighter integration with its programming model than that achieved in both portings and, consequently, potentiate its advantages. However, the portings, as such, showed that the adoption of the API improves the quality of persistent code. It reduces work-effort and error proneness by reducing the size of the application and augmenting the level of abstraction in the manipulation of persistent data elements.

The PJama Case

The source code metrics analysed in this study have been previously used to report usability aspects of the PJama technology on a diverse set of applications [GSAW98]. The primary goal of the PJama study is to show how PJama introduces only minimal changes into applications' source code. It argues that the "almost subliminal" use of orthogonal persistence leads to good usability.

The experimental results of the Datom API and the PJama technology differ on a fundamental issue: they aim to evaluate different models of persistence. Therefore, there is a conceptual difference between what is considered a line of persistent code in each result. Due to the data-centric approach enabled by selective reachability, it is feasible to identify all the explicit references to persistent code in the metrics of the Datom API by including the persistent data types in the keyword file. Used in this way, the PCMT tool provides a precise characterisation of how much of the code in an application is indeed related to the manipulation of the graph of persistence. This is clearly reflected by the presence of persistent code in many classes of the application.

In contrast, the PJama team reports only explicit references to the PJama keyword set and not the ramifications of the graph of persistence via conventional Java objects. Consequently, it is not possible to keep track of implicit objects that are part of the graph of persistence since they have not been included in the keyword set. This is evidenced by the small number of classes reported in the PJama study as persistent.

Ultimately, to produce results with the same level of detail to those obtained for the Datom API, the keyword set for the PJama case would have to be defined as the programming language itself. For this reason, it seems infeasible to closely trace a graph of persistence in application code that uses an orthogonally persistent object system with the PCMT tool. A reachability-based strategy may be more appropriate for this analysis.

6.2 Performance Evaluation

This section presents the performance evaluation of the prototype of the Datom API. Its main purpose is to measure two fundamental issues of its operation: the read and the write barriers. The results characterise the performance of the prototype and indicate that its operation degrades steadily as a larger number of persistent items needs to be handled.

6.2.1 Evaluation Description

One of the key motivating factors for the development of the Datom API is the reduction of the conceptual barrier between persistent and transient data. The programmer manipulates a model in which the data store can be seen as a persistent extension of volatile memory; data is dynamically allocated according to data usage patterns. Additionally, the data model enables fine-grained manipulation of data elements which could lead to performance problems.

To properly support this view of persistence there are two mechanisms that the system must implement efficiently and that can be used as the basis for any study of the performance of persistent systems [Hos95]: object faulting and detecting updates, also known as the read and write barriers, respectively. In order to evaluate these properties, the experiments presented next use a similar evaluation strategy to that used by the OO1 benchmark [CS92] and their offsprings. The main goal is to manipulate a synthetically-created graph of persistence to measure the elapsed time taken to operate on the graph.

Instead of creating a complex graph of objects, as done in the OO1 benchmark which aims to reproduce operations for engineering databases, the evaluation data sets were designed to isolate the behaviour of individual Datom programming objects. This decision was taken since the typical data complexity and data access strategies supported by the Datom API differ from those seen in engineering applications.

First, the Datom API has been designed to provide predetermined data access facilities through a group of managed abstractions. Second, the amount of persistent data manipulated through the API is expected to be much lower than those manipulated in the engineering database domain. Finally, this study aims to concentrate on high-level issues of the performance exhibited by the prototype, considering that the most accurate measure of performance will always be related to the execution patterns of the application.

Five different data sets were synthetically created. Each of the data sets consists of 36,000 Datom Elements stored in a different Composite Entity. This number is useful to provide an assessment of the scalability of the system and its ability to manipulate a large amount of objects. The Element type employed in the tests resembles a typical programming object that contains a mixture of data types. It stores the following information:

```
Data   {
    int id;
    double number;
    String type;
    int x;
    int y;
    Date build;
}
```

The `id` field is assigned a unique integer value ranging from 1 to 36,000. The `x` and `y` attributes have values randomly chosen in the range [0-10,000]. The `number` field stores randomly distributed values in the range [0.0-1.0]. The `type` fields have values randomly selected from the strings {"part-type0", ..., "part-type9"}. Finally, the `build` date is randomly distributed in a 20-year range.

The Read Barrier: Persistent Item Faulting

The first mechanism mediates retrieval of stable storage into memory for program manipulation. Any operation that directly accesses a data value whose residency is in doubt must first check that the value is available in memory. Such residency checks constitute a read barrier to any operation that accesses persistent data: before the system can read (or write) any data, it must first make sure the data is resident. Furthermore, if the residency check fails the data should be loaded from disk to memory incurring an additional penalisation time.

The experiments used to measure the impact of the read barrier in the prototype emphasise selective retrieval for the map, list, and matrix abstractions and one-way retrieval for the stack, and queue abstractions. These two types of data retrieval strategies are aimed to reproduce the data access semantics of the holding Composite Entity. They operate as follows:

- **Selective retrieval.** It fetches 9,000 items randomly chosen using the access semantics of the holding Composite Entity, either a `DatomMap`, a `DatomList`, or a `DatomArrayMatrix`. It emphasises data access strategies based on items' attributes. Once the item has been fetched a null procedure is invoked for each of them taking as its arguments the `number`, `x`, `y`, and `type` attributes of the Data Element.

  Similar to the the OO1 benchmark experiments [CS92], the performance tests assess the operation of the prototype in three different states: cold, warm, and hot. A run consists of 25 iterations over the data sets. The first 24 iterations cause a gradual warming of the prototype's caches. The 25th iteration accesses the same items as the last warm iteration to visit only resident Elements, freeing the system of the overheads caused by retrieving a non-resident persistent item such as swizzling and data translation.

- **One-way retrieval.** It fetches a fixed number of items using the interface of the holding Composite Entity, either a `DatomStack`, or a `DatomQueue`.

Similar to the selective retrieval operation, once the item has been fetched a null procedure is invoked taking as its arguments the `number`, `x`, `y`, and `type` attributes of the Data Element.

A run in this case consists of 8 iterations over the data sets. On each iteration a number of $1000n$ Data Elements are retrieved with n taking the corresponding iteration number. Accordingly, an increasing number of items are processed at each iteration reaching a total of 36,000 items at the end of the run. The data access semantics of the Composite Entities evaluated in this experiment obviates the need of any of the prototype's caches. Therefore, the effects of cache warming are not taken into consideration.

The Write Barrier: Detecting and Logging Updates

The second key persistence mechanism ensures that updates to persistent objects are reflected in the store. Making these updates permanent means propagating them to stable storage. Thus, every operation that modifies persistent data requires some immediate or subsequent action to commit the modification to disk. The strategy adopted in the implementation of the Datom API is to keep modified data in memory and wait for the programmer to indicate the actual moment in which changes should be propagated to the stable store. Therefore, any operation that modifies persistent data in memory must indicate to the storage system to remember those elements that have been updated. Recording updates in this way constitutes a write barrier that must be imposed on every operation that modifies persistent data; writes require additional overhead to record the updates.

The experiment used to measure the impact of the write barrier aims to measure the cost of updating persistent items and making those modifications permanent. It operates in the following way:

- **Update.** It uses similar data access strategies to those used to evaluate the read barrier for each of the Composite Entities of the Datom API, i.e. selective and one-way retrieval. However, it performs a simple update on the fetched Data elements according to a fixed probability. This probability varies from one run of the experiment to the next in order to modify the density of updates. The update consists of a method call that increments the `x`, `y`, and `number` attributes of the Data Element. Updates are pushed to the persistent store at the end of each iteration. Thus, it measures the cost of modifying persistent items and preserving those changes in the persistent store.

Experimental Setup

The machine in which the experiments were run is an AMD Athlon^{TM}1400 MHz processor with a 256 KB L2 cache and 1 GB DRAM running Fedora Core 3 and using the *ext3* file system. The data sets were placed in an internal IBM 60GXP disk with 60 GB of capacity, 100 MB/s peak data rate, 20.9-40.8 MB/s sustained data rate, and 8.5 ms average seek time.

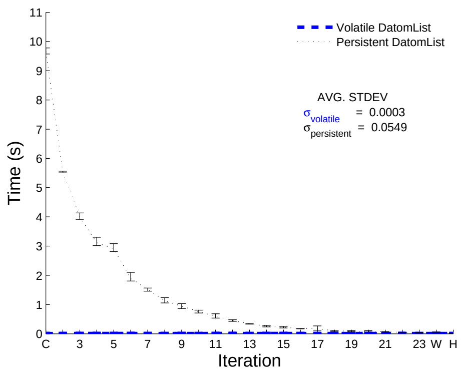
The experiments were executed in single-user mode and disconnected from the network, to minimise interference from network traffic and other system activity. The programs were compiled for the Java^{TM}2 Runtime Environment, Standard Edition, version 1.4.2. The maximum Java heap size was set to 300 MB using the -Xmx runtime option and the Berkeley DB cache acting as the buffer pool set to 300 MB to ensure that the synthetic data sets could be totally cached in memory. The elapsed times for the experiments were obtained using the `System.currentTimeMillis()` Java call with a precision of 1 ms.

Buffer management policies can be ignored when interpreting the experimental results. Since the amount of memory in the testing system is 1 GB, it is also possible to ignore the effects of virtual memory paging. Before each run the system is cooled by reading a 1 GB file to clean the operating system cache. All the values reported correspond to averages computed from running the experiments four times; they exclude the initialisation and setup of the system prior to the beginning of the corresponding test.

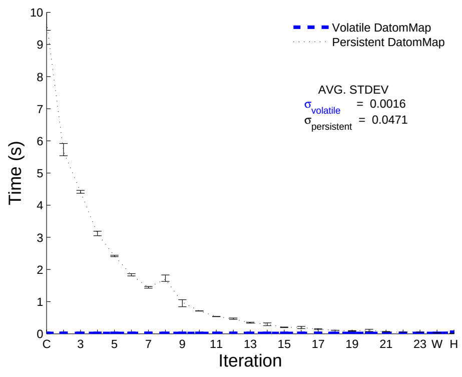
6.2.2 Persistent Item Faulting

Figure 6.5 reports the performance and the impact of the read barrier in those Composite Entities that offer selective retrieval of Elements. The subplots 6.5(a), 6.5(b), and 6.5(c) provide the results for `DatomList`, `DatomMap`, and `DatomMatrix`, respectively. Finally, the subplot 6.5(d) illustrates the average result considering the three abstractions.

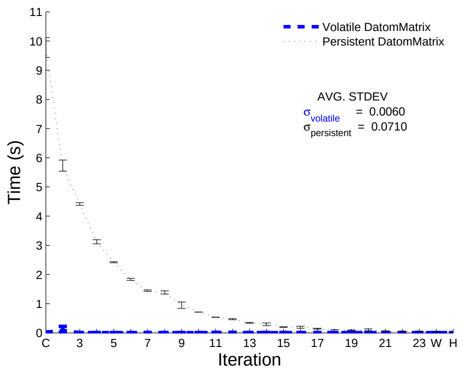
Figure 6.5 reports the total elapsed time for the execution of the experiments from cold through warm to hot iterations. The *volatile* version represents a non-persistent implementation of the data sets and it has been coded using only in-memory objects. It characterises the best performance that would be possible to obtain and it is presented only as a baseline measurement.



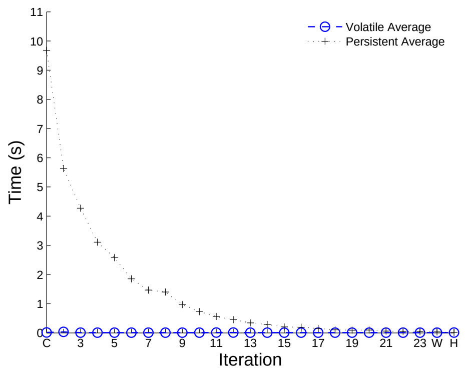
(a) Datom List.



(b) Datom Map.



(c) Datom Matrix.



(d) Results averaged.

Figure 6.5: The read barrier performance in the Composite Entities supporting selective retrieval.

All the Composite Entities exhibited similar performance patterns. The graphs show a clear warming tendency due to the presence of an increased number of items in the prototype's caches from one iteration to the next. However, the performance of the cold iteration together with the first six cold to warm iterations is heavily dominated by disk retrieval. The reduction in the elapsed time is mainly due to the decreasing number of disk accesses and the diminution of overheads related to data translation.

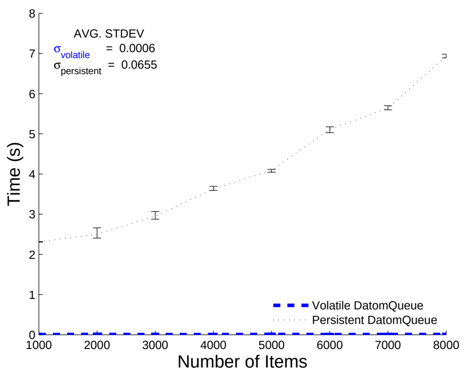
The design decision of loading into memory one item at a time in the current prototype may imply disk activity for every item requested. As a consequence, the warming is gradual and the plots show a right-skewed distribution. The values for the persistent graph are close to those of the volatile version only after the 17^{th} iteration. After this point the overhead for the read barrier decreases gradually and becomes trivial. However, the warming tendency could be manipulated through the implementation of more aggressive prefetching strategies or by augmenting the granularity of the transfers from disk, always having in mind the trade-off caused by the corresponding higher overheads on initial data retrieval due to swizzling and data transformations.

Figure 6.6 reports the performance of those Composite Entities that allow one-way retrieval of Elements. It reports the total elapsed time for the execution of the experiments in seconds. Once more, the figure is divided in subplots providing detailed results for the `DatomQueue`, and the `DatomStack` in graphs 6.6(a) and 6.6(b), respectively. The subplot 6.6(c) illustrates the average of the two abstractions.

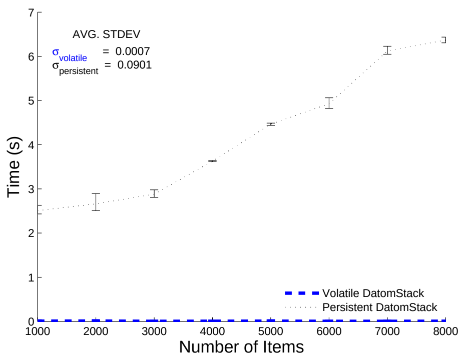
The `DatomQueue` and the `DatomStack` showed similar performance numbers. The plots in Figure 6.6 also include the time for the volatile version of the abstractions to provide a baseline. In general, the access semantics for the queue and the stack abstractions indicate that every access retrieves a new object. Therefore, the effect of cache warming is not considered in these experiments. These results can be seen as a worst-case scenario since every access to the store fetches a non-resident item.

As would be expected, the latencies clearly reflect the overhead of disk retrieval and data translation. The elapsed times for each iteration increase steadily according to the number of objects fetched from the store and, most importantly, the performance degrades steadily for successively larger numbers of items. The one-item-at-a-time retrieval policy enforced by the implementation of the Datom API could be easily adapted for these abstractions in a way in which applications always access resident items.

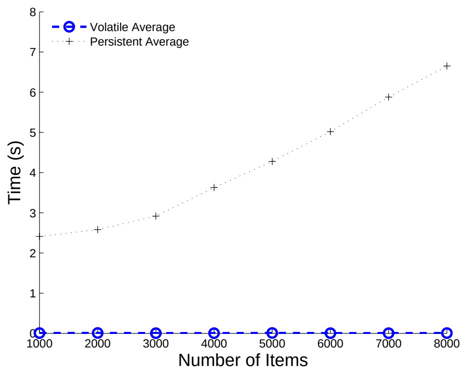
Thus, the Datom abstractions can be used by applications to provide assertive hints about their persistent data access requirements. However, the trade-off between memory consumption, swizzling eagerness, and observed latencies should be balanced according to the execution patterns of applications.



(a) Datom Queue.



(b) Datom Stack.



(c) Results averaged.

Figure 6.6: The read barrier performance in the Composite Entities supporting one-way retrieval.

6.2.3 Detecting and Logging Updates

The evaluation of the write barrier is done through the measurement of the elapsed time needed to perform a number of updates in each of the managed abstractions of the Datom API. Figure 6.7 illustrates the latency of updating the graph of persistence for Composite Entities that support selective data retrieval. Updates were applied with a probability of $p = \{0, 0.05, 0.10, 0.2, 0.4, 0.7, 1.0\}$ to vary the density of modified items. The graphs report the measurements obtained for the cold, warm, and hot iterations at each of the p values.

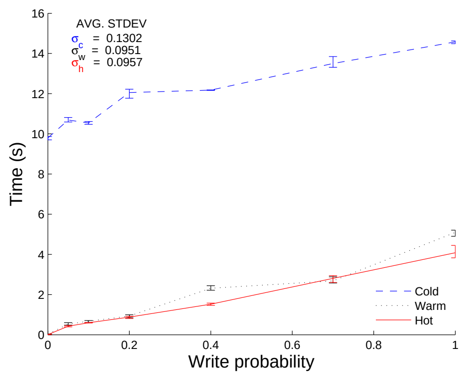
The write barrier for the `DatomList` is illustrated in subplots 6.7(a), and 6.7(b). The former subplot shows the overall elapsed time while the latter concentrates only on the time taken to checkpoint the graph of persistence. The checkpoint call is performed after each iteration of the test causing a single atomic operation to transfer all the mutated items from memory to disk. Similar pairs of subplots are presented for the `DatomMap` (6.7(c), and 6.7(d)), and the `DatomMatrix` (6.7(e), and 6.7(f)) abstractions.

Similar trends were observed for total update latencies for the three kinds of abstractions shown in Figures 6.7(a), 6.7(c), and 6.7(e). Cold iterations exhibit the largest update latency. They are importantly affected by disk access times needed to first load persistent items into memory and then to flush them to disk whether they have been updated. In contrast, a considerable reduction in the total update latency was observed for the warm and hot iterations. Warm iterations will sporadically incur in the overhead of fetching a non-resident item and consequently may obtain slightly larger total update latencies than those obtained for hot iterations. As expected, hot updates consistently showed the smallest update latencies since the working set is fully contained in memory and thus, they only perform disk accesses for write operations. In the three types of iterations the performance degraded steadily.

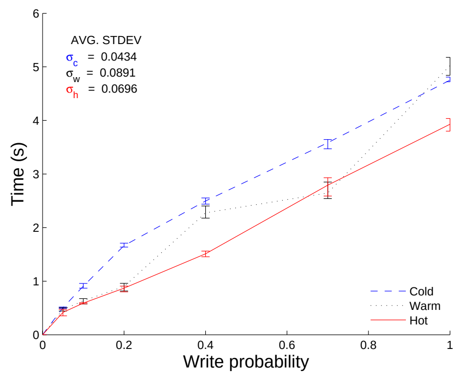
Checkpoint latencies illustrated in Figures 6.7(b), 6.7(d), and 6.7(f) correspond to the time taken to traverse the set of mutated objects, translate these objects to their on-disk representation, and finally write them to disk as a single atomic operation. Checkpoints are invoked through the corresponding `save()` method of the holding Composite Entity. The checkpoint latencies increased constantly as the density of updates increased. The graphs show similar values for the three types of iterations since they involve transferring roughly the same amount of persistent items to disk; in this process, the impact of the cache state in latency times is negligible.

Delaying a group of updates until checkpoint time trades durability of operations for efficiency. Higher latencies would be obtained if each update is protected with a checkpoint as it involves disk activity and transaction management overhead. However, the ability to hold a group of updates in memory until checkpoint is a feature that can be sensibly used by applications to fine-tune their recoverability semantics.

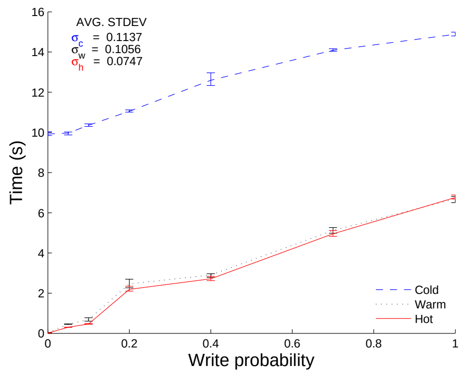
The write barrier for the two abstractions supporting one-way retrieval, i.e. `DatomQueue` and `DatomStack`, is illustrated in Figure 6.8. It includes subplots



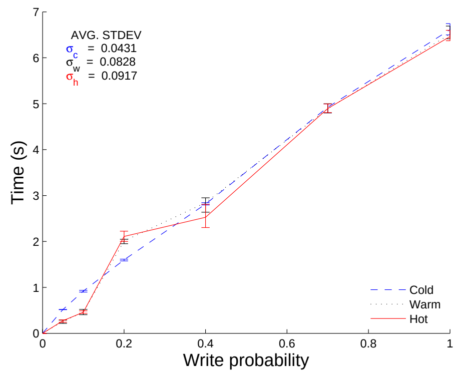
(a) Total update latency for DatomList.



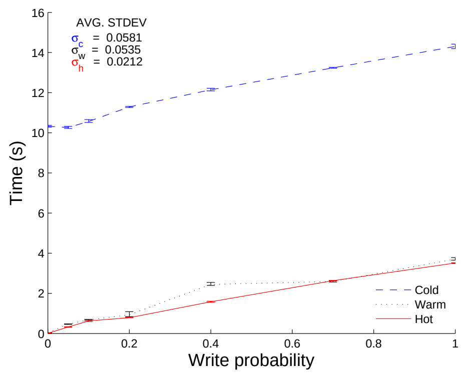
(b) Checkpoint latency for DatomList.



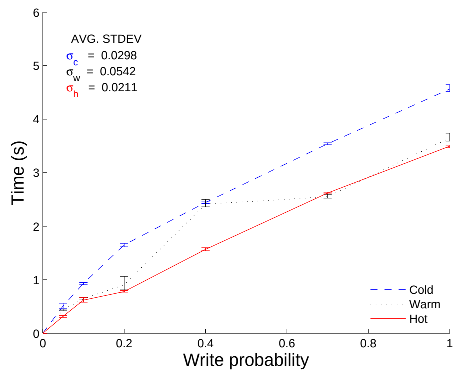
(c) Total update latency for DatomMap.



(d) Checkpoint latency for DatomMap.

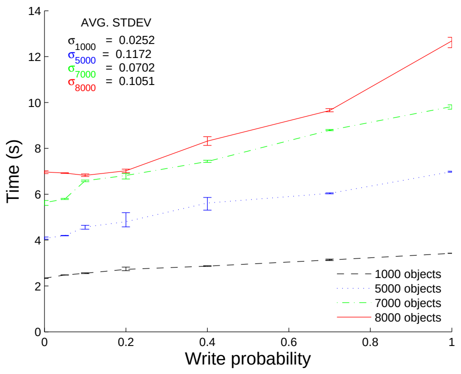


(e) Total update latency for DatomMatrix.

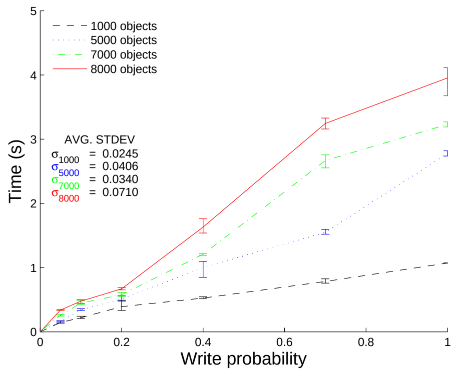


(f) Checkpoint latency for DatomMatrix.

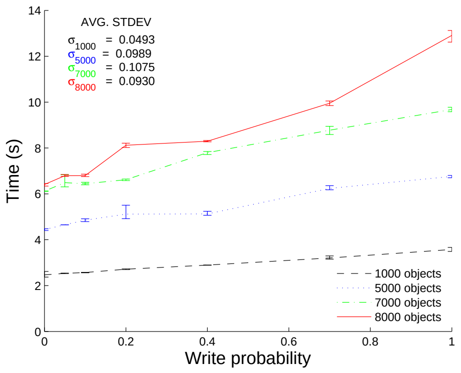
Figure 6.7: The write barrier in Composite Entities supporting selective retrieval.



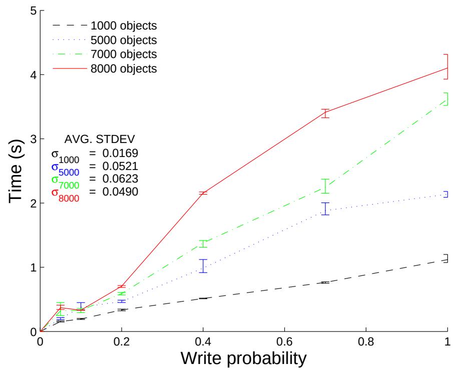
(a) Total update latency for DatomQueue.



(b) Checkpoint latency for DatomQueue.



(c) Total update latency for DatomStack.



(d) Checkpoint latency for DatomStack.

Figure 6.8: The write barrier in Composite Entities supporting one-way retrieval.

to report both total update latency and checkpoint latency. The checkpoint is performed after each iteration as a single atomic update. Subplots 6.8(a) and 6.8(c) show the total update latency for four different iterations which update an increasing number of items; they correspond to the first, fifth, seventh, and eighth iteration of the one-way retrieval access strategy. Both plots show a similar latency trends which are influenced by the number of objects manipulated and the density of updates; total update latencies increase proportionally to these factors.

A detailed analysis of checkpoint latencies is illustrated in graphs 6.8(b), and 6.8(d). They prove that performance times increase uniformly as more items are flushed to disk. As expected, smaller data sets produce comparable latencies independently of the density of updates since less disk accesses are needed to checkpoint the graph of persistence. Checkpoint times for larger data sets tend to be largely dominated by disk access times.

6.2.4 Discussion

The performance evaluation of the prototype focused on two fundamental implementation features: the read and the write barriers. The tests focused on coarse-grained issues to characterise the API's performance and scalability. Benchmarking or performing a face to face comparison of the prototype with other storage technologies was considered inappropriate for two main reasons. First, the current implementation of the Datom API stresses functionality instead of performance; there is, consequently, opportunity for performance tuning. Second, the data model enabled by Datom's persistent abstractions differs from those employed in other storage technologies.

However, the results obtained for the performance study provide a good indicative of the latencies that the current implementation exhibits. Furthermore, they highlight the areas in which the prototype's performance could be improved. Overall, the results show that the performance degrades gracefully as long as the prototype's caches are large enough to avoid unnecessary disk accesses. As expected, experiments indicate that there is a considerable performance gap between having an item ready to be used in-memory and having to fetch it from disk. The reduction of this gap would be possible if computing systems with large amounts of memory are considered, something that seems practical since memory is getting cheaper all the time.

However, eager prefetching strategies may not be a comprehensive solution since eager prefetching benefits may be limited by initialisation costs. To load an item into memory when the system is cold involves a number of disk accesses and the translation of the item from its on-disk representation to its in-memory version. The break-even point for data prefetching strategies depends on various issues such as the execution characteristics of the application, data clustering strategy used by the storage layer, and the swizzling scheme.

Eventually, the most accurate measure for performance will always be related to the execution patterns of the application. In this respect, the Datom API provides assertive hints about expected data access patterns through high-level programming abstractions from which different prefetching, data clustering, and fine-tuning strategies could be implemented and tested. However, a detailed study on all these aspects forms part of the future work of this research since, to have an important effect, they invariably involve the creation of an ad-hoc storage layer, which is out of the scope of this dissertation.

6.3 Usability Aspects of the Datom API

This section comments on usability aspects of the Datom API. The contribution of the usability evaluation presented here is twofold. First, it describes the API and provides a common background and vocabulary to be of use in the discussion of its usability features. Second, it exposes the strengths and weaknesses that different types of developers may find while interacting with the functionality provided by the Datom API.

6.3.1 Measuring Usability

This section introduces the methodology employed in evaluating usability features of the Datom API. The usability study presented in this section is based on the Cognitive Dimensions (CD) framework [GP96] which presents a number of dimensions upon which judgements are made regarding cognitive demands made by a particular notation or programming language. Although initially employed as a usability analysis tool for visual programming environments, the CDs framework has been generalised by Blackwell and Green [BG00] to be applied to any sort of information devices. Consequently, the CDs framework has been successfully used in performing API usability studies [Cla04], in evaluating new programming languages such as C# [Cla01], and in assessing tools such as spreadsheets [Tuk01].

The CDs framework was chosen as an evaluation tool for the usability of the Datom API for various reasons. First, it represents a well-analysed and proven methodology for performing usability tests which motivates the discussion of software attributes from a broad range of perspectives. Furthermore, the evaluation is performed from a programmer point of view rather than from the perspective of the software designer; this provides a more impartial assessment of the usability of the API and points out aspects that designers may have initially overlooked [BG00].

To evaluate the usability of the Datom API, the CDs framework has been employed as proposed by Clarke and his usability group at Microsoft [Cla03, Cla04]. Accordingly, the usability study performed on the Datom API comprised different stages. It started by identifying the typical programming *scenarios* supported by the Datom API; they are shown in Table 6.8. They range from the most basic use of the API to advanced functionality that has to be implemented using different parts of the API simultaneously. These scenarios represent the fundamental use cases for the API and embrace, to an important degree, most of the functionality that it offers to developers.

Id number	**Description of scenario**
1	Open a collection of persistent objects.
2	Prepare the persistent data layout for a collection of data items.
3	Add a data element to the collection of persistent items.
4	Update a data element in the collection of persistent items.
5	Read data from a persistent data element.
6	Delete a data element from the collection of persistent items.
7	Remove a collection of persistent items.
8	Apply updates atomically to a collection of persistent items.
9	Modify the organisation of the persistent data layout.
10	Query the collection of persistent items.
11	Navigate and update the collection of persistent items.

Table 6.8: Typical short scenarios of the Datom API.

In a second stage, *code snippets* were created for each of these scenarios. These code samples represent the kind of code that developers have to write to implement the functionality exemplified by each scenario. To simplify the readability of this document the code samples have been included in Appendix B. In a third stage, a *task analysis* for each of the scenarios was done. The task analysis does not describe how the API supports each scenario, instead it describes the different actions that a programmer would expect to perform in order to accomplish the goal represented by the given scenario. The breakdown of tasks for each of the scenarios is presented in Appendix C. The final stage in the usability study includes the *analysis and evaluation* of the API based on the code samples and task analyses for each of the scenarios.

The evaluation of an API has to contemplate the target audience since different *programming styles* require different types of APIs. In general, an API that works well for a specific type of programmer may not be appropriate for another. For this reason, the results presented in the following section are contrasted with the typical cognitive demands made by the three main developer profiles that have been identified by Clarke et al. [Cla03]. This categorisation is supported by the empirical evidence obtained through the observation of large numbers of programmers. Accordingly, conclusions were drawn according to the demands of developers on each of the cognitive dimensions.

The ultimate contribution of this study indicates for which developer profile the Datom API would be suitable and why. The three main types of developers are:

- **Opportunistic programming style.** Developers in this category feel comfortable using APIs that expose high-level abstractions. They will be satisfied using an API that can be used without any major extensions and that does not force them to acquire a large conceptual background to be successful with the API; thus, they like to gain confidence in the use of APIs by handling only the parts that are significant to their immediate goals.
- **Pragmatic programming style.** Pragmatic programmers favour APIs in which components have been factored to exhibit a direct correspondence with programming goals. They like to be able to extend the standard behaviour of the API and fine-tune relevant aspects of its behaviour if needed. Thus, they want to understand the effect of the API functionality in the global context of the application. They like to learn APIs using a top-down approach and incrementally.
- **Systematic programming style.** Developers matching this programming style enjoy working with APIs that exhibit low-level abstractions in the form of primitive components. Systematic programmers like APIs in which components can be replaced and new types introduced. To feel comfortable with an API, they like to learn it using a top-down approach and to understand its architecture even before writing any code. Programmers of this type want to know the effect of code at both the application and the system level. Accordingly, they like APIs that expose the intricate details of the operation of the API.

6.3.2 Usability of the Datom API

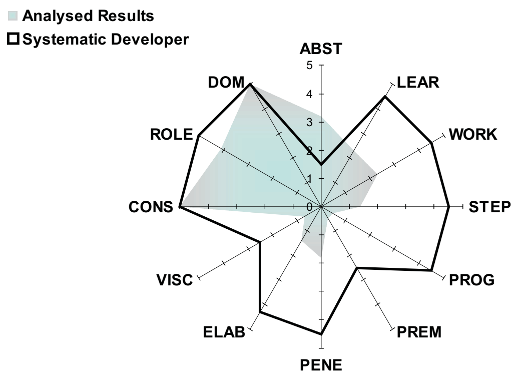
Figure 6.9 shows the results of the usability study compared with the three developer profiles. The spokes in the radar graphs constitute each of the different dimensions being evaluated; the end points on each spoke are the minimum (i.e. centre of the radar) and maximum (i.e. outer edges) values on the scale for each dimension. Accordingly, the points where the lines cross the spokes represent the values on the scale for a specific dimension. By comparing the points where the solid area and the black line cross each spoke, it is possible to evaluate how well the Datom API matches the profile's ideal API.

Each graph indicates what developers matching a given profile would expect from an API in terms of the cognitive dimensions evaluated by the CDs framework. Developers' profiles are illustrated in the graphs with black lines; they correspond to the characterisation produced by Clarke. The solid areas in the graphs show the results obtained in the usability analysis of the Datom API. These values are the averaged results of grading all the scenarios in a scale with values low, medium, or high for every dimension. The next paragraphs analyse each of the cognitive dimensions and explain the results shown in Figure 6.9.

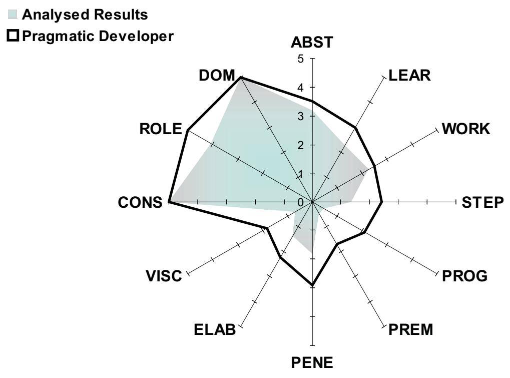
Abstraction Level (ABST)

The abstraction level dimension measures the type and number of abstractions that the developer has to contend with. APIs in which individual programming tasks identified in the task analysis have to be accomplished by using two or more components in conjunction are considered of *primitive* type; this kind of API offers low levels of abstraction. Components correspond to the programming classes that compose the API. An API is said to be made of *factored* components when each class in the API has been designed with a particular programming goal in mind; factored components show a direct correspondence between programming tasks and classes used in the source code. Finally, *aggregate* components offer the highest level of abstraction; in these APIs the set of programming goals could all be accomplished with the same set of programming components.

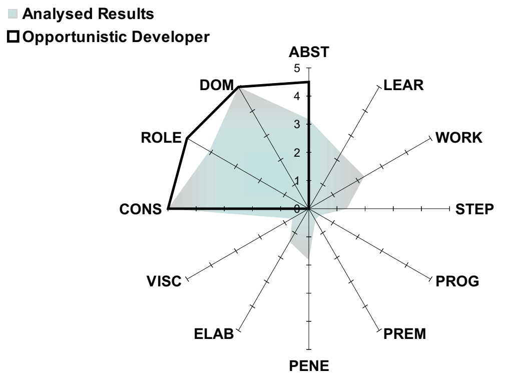
The usability study for this dimension indicates that the number of classes that users are exposed to with the Datom API is relatively small and that the fragmentation of individual programming tasks into more than one line of code is unusual. The programming scenarios show that the API has been factored to the level of individual tasks; there is a correspondence between components in the API and the set of tasks programmers might expect to perform. Even the most complicated programming scenarios employed a combination of factored components to accomplish the programming goal. Although the Datom API also



(a) Systematic programming style.



(b) Pragmatic programming style.



(c) Opportunistic programming style.

Figure 6.9: The results of the usability study of the Datom API according to the cognitive demands of different programming profiles.

enables the creation of aggregate components through the use of rich Datom Elements, this is a programmer concern and part of the flexibility of the API to adapt to different abstraction levels rather than an intrinsic feature of the API.

The abstraction level shown by the Datom API seems to fulfil quite well the expectations of pragmatic programmers. Moreover, opportunistic programmers should also feel comfortable with the abstraction level exposed by the API since the number of classes directly manipulated by them is small.

Learning Style (LEAR)

This dimension measures the learning requirements posed by the API. An *incremental and minimal* learning style enables developers to gain confidence in the API by using only the parts of an API relevant to their goals. Programming goals written with APIs that support an incremental and minimal learning will normally use a small number of classes with minimal dependencies. APIs that demand a *structured* learning style require a large number of classes to perform a determined programming goal and each of those classes have a large number of dependencies.

The minimal knowledge needed by programmers in order to begin working with the Datom API is that they need to create a `Store` object in which a graph of objects can be rooted; this forces programmers to start learning the API in a structured way as they need to know the different types of persistent elements that the API supports and their relationships. However, the components of the API are written and documented in a way that make these features evident from the use of the API. After learning these fundamental properties, the Datom API is close to a minimal and incremental learning style.

In the majority of the scenarios developers are able to accomplish the programming goals using loosely coupled programming objects. The more complex programming scenarios require the understanding of a greater number of relationships between programming components demanding a more structured learning style. However, these scenarios are made of simpler tasks that allow the programmer to write a couple of lines of code to corroborate that the API is working as desired and to understand the totality of the functionality provided by the Datom API in this way. Furthermore, most of the dependencies observed are due to particular persistent data layouts required by the logic of the application and not to intrinsic properties of the API.

The learning style enabled by the Datom API is suited to pragmatic programmers, who show a step-wise learning style. They like to gain confidence with an API in a structured way but gradually, piece by piece. Opportunistic programmers have no desire to understand a large number of classes and their dependencies before being able to do something useful with the API; to some extent, they would also feel comfortable with the API since the number of abstractions that have to be understood to start using the API is small.

Working Framework (WORK)

The working framework dimension evaluates the size of the conceptual chunk needed to work effectively with the API; this is different from the LEAR dimension that indicates how developers learn to use the different API components. If the information that a developer needs to maintain in order to perform a specific programming goal is directly represented in the API, then the working framework is described as *local*. However, if the information developers need to maintain is not directly represented in the API but can be inferred from the way the source code is structured, the API supports a *global* working framework. Finally, the most demanding APIs are those that force the programmer to manipulate plenty of information at various levels (e.g. application and system) to be able to work with the API effectively; these working frameworks are described as *systems*.

Although much of the information that the programmer needs to track is well represented in the individual components of the Datom API, which makes the code self-describing at the local level, there is still an amount of extra information that developers have to maintain to be able to successfully manipulate persistent data through the Datom API. First, the code that developers have to write in order to manipulate persistent data through the API depends on the morphology of the graph of persistence and the type of persistent items that compose it. However, these two features are made explicit in the source code through components' relationships and objects' castings which enable developers to obtain this information through the way code is structured. Second, as persistence by reachability is used by the Datom API, developers have to keep a mental representation of the objects that are members of the graph of persistence in order to properly update applications' persistent data.

The Datom API consistently exhibits a global working framework in all but the simplest of the scenarios. Developers that better identify with the global working framework are pragmatic programmers, who are willing to know the effect of code on the whole application. To a certain degree, the API may suit opportunistic programmers as well, since the demands placed by the API can be tolerated due to the small number and consistent behaviour of the abstractions that have to be manipulated.

Work Step Unit (STEP)

This dimension indicates how much of a programming task can be completed in a single step; it measures the amount of work that needs to be done for specific programming tasks. A *local-incremental* work step unit enables developers to fully contain programming tasks in single local code blocks. However, if a programming task is contained within multiple code blocks, or if the code requires the instantiation of multiple classes that interact, the work step unit is defined as *parallel*. Between local-incremental and parallel there is the *functional* work step unit.

After revising the work involved for each of the tasks highlighted in the scenarios' task analysis, it is possible to corroborate that in general there is a linear progression from the start to the end of the tasks since only one line of code is consistently required to perform a task. In the most elaborate scenarios a given programming task might involve writing a few lines of code with interacting components; however, this code shows a clear incremental progression to accomplish a specific programming task, which defines the work step unit for these scenarios as functional.

Both opportunistic and pragmatic developers prefer to work incrementally, with the latter profile being able to adapt better to functional work step units. From the evaluation of the API in this dimension one can conclude that opportunistic and pragmatic programmers should be able to manipulate the Datom API with comfort. In contrast, systematic programmers prefer powerful and extremely flexible APIs built around primitive types in which individual programming tasks can be decomposed and customised.

Progressive Evaluation (PROG)

This dimension indicates to what extent it is possible for a programmer to execute partially completed code to obtain feedback on its behaviour. Progressive evaluation at the *line of code* level takes place when an API enables developers to stop and check progress after each line of code they write. If developers can only evaluate their progress for one task after writing enough code to perform two or more tasks, the API supports progressive evaluation at the *functional chunk* level. If multiple classes in parallel need to be in a consistent state with respect to one another in order to check progress, the API supports progressive evaluation at the *parallel components* level.

By examining each of the scenarios, using the task analysis and the code samples, it is possible to observe that with the Datom API the programmer is able to effortlessly check the progress of individual tasks and evaluate how much they contribute towards accomplishing the goal of the given scenario. In general, it is possible to check progress at the line of code level. However, similar to the STEP dimension, there are a couple of scenarios for which it is necessary to write code to accomplish two or more tasks before progression to the goal of the scenario can be checked. In these cases (i.e. atomic updates or querying the graph of persistence) the API shows progressive evaluation at the functional chunk level.

As mentioned before, opportunistic as well as pragmatic programmers prefer to work incrementally. The Datom API enables this type of programming style as it consistently enables progressive evaluation at the line of code level. However, this dimension may be affected by the extent to which programmers think in the same terms that are represented in the Datom API. If they do not think in the terms of the components of the API, it would be more difficult for them to judge if the outcome is correct at each incremental step.

Premature Commitment (PREM)

Premature commitment indicates the amount of decisions that programmers have to make when writing code and the consequences of those decisions, i.e. how easy is to recover from initial decisions. If an API presents users with a minimal number of choices about how to accomplish some goal and if the differences between the alternatives are minimal, the API exposes a *minimal and reversible* level of premature commitment. When the number of choices is significant but the differences among them are minimal, the API presents a *significant and reversible* level of premature commitment. In the cases in which the alternatives are abundant and the differences between them are significant, the level of premature commitment is said to be *significant and irreversible*.

The Datom API shows a minimal and reversible level of premature commitment for most of the programming scenarios. This is mainly due to the reduced number of components that have to be directly managed by programmers and to the fact that many of the components correspond to programming abstractions with standard access semantics (i.e. maps, lists, matrices, etc.). These features minimise the number of decisions that a programmer has to face when deciding how to accomplish a given programming scenario.

However, there are two situations in which programmers are presented with options. First, programmers have to decide which of the groups of Composite Entities should be employed and their respective relationships when setting up a graph of persistence. The final decision in this case will be generally guided by application-specific data access semantics. In the worst case, inadequate or inefficient data layouts can be modified with relatively small effort by properly manipulating the components provided by the API. Second, programmers face the decision of when a persistent item should be integrated into the graph of persistence.

Independent of the initial decision of the programmer, it should be relatively easy to modify the moment in which a given object is included or evicted from the persistent space, since the API exhibits well-defined semantics at the Composite Entity level and coarse-grained reachability at the Element level. In conclusion, the Datom API exposes a close to minimal and reversible level of premature commitment. All programmers prefer to work with minor and reversible APIs; however, different profiles are able to better manage APIs that offer a significant number of options, as shown in Figure 6.9.

Penetrability (PENE)

The penetrability dimension describes how much programmers must understand about the underlying implementation details of an API and the extent to which programmers are able to understand those details directly from the API usage. An API is said to provide a *snapshot* view into its details if it exposes only enough information to allow programmers to distinguish between methods and classes provided by the API, and this is the only information that programmers need to take note of. A *context driven* view of the details of a given API is one in which the API exposes enough information to allow a programmer to understand the context or scope of the particular part of the API the programmer is working with. If the API exposes enough information to allow the programmer to understand the intricate working details of the API and if the user must attend to this information to work effectively with the API, it provides and requires an *expansive view* of the details of the API.

The navigational approach to persistent data access, the presence of two storage spaces (i.e. the volatile and the persistent), and the different types of persistent items supported by the Datom API demand from developers a degree of understanding of the API's implementation details. The source code samples indicate that the Datom API by itself enables programmers to understand the context and scope of persistent code in these areas. Furthermore, the API is designed in a way that implicitly forces developers to observe these implementation details. A greater degree of penetrability into implementation aspects of the API can be gained by the developer through the API documentation or source code. However, the Datom API by itself does not expose low-level implementation details.

For these reasons, the penetrability of the Datom API is close to context driven. This type of API suits pragmatic programmers. However, opportunistic programmers should be able to work successfully with the API attending only to information exposed by its methods and classes. Most of the implementation details required to use the API can be worked out using the common sense or are implicitly embedded in the abstractions exposed by the API.

API Elaboration (ELAB)

This dimension measures the extent to which developers must and can extend the API in order to accomplish a given task. If an API allows a programmer to perform all of their goals only by using the types exposed by the API without requiring any further elaboration, the API can be used *as-is*. However, if the API allows developers to accomplish their goals only by extending some of the types exposed by the API and providing their own implementation of custom behaviour, the API supports elaboration at the *fine tune* level. Finally, if the API allows programmers to perform all their goals only by replacing or introducing whole new types, the API demands elaboration at the *replace* level.

The study of the different programming scenarios show a remarkable pattern. All the programming goals can be accomplished by using the components of the API as they are provided except those in which Datom Elements are required. Since the goal of the Element type is to provide a universal and customisable recipient for application specific data, programmers are required to extend this type in order to properly manipulate application-specific persistent types.

Once more, the Datom API seems to match opportunistic and pragmatic developers since both like to manipulate components that can be reused as-is. However, the latter also want to be able to fine-tune components of the API to some extent. The use of Datom Elements may appeal to pragmatic programmers since they enable fine tuning of data access strategies and customisation of persistent data layouts. The fact that the defined group of Composite Entities in the Datom API has to be used as-is may particularly annoy systematic programmers who want to manipulate APIs in which components can be replaced, although not necessarily being forced to do so if it is not needed.

API Viscosity (VISC)

The API viscosity dimension measures the barriers to change inherent in the API. When the API allows users to make changes to code written with an API easily, the API has *low* viscosity. If programmers spend moderate effort when trying to change code written with an API, the API is said to exhibit *medium* viscosity. If the API demands users to make a significant effort to change code written against an API, the API has *high* viscosity.

The Datom API presents a small number of options in terms of the alternative ways in which the tasks comprising each of the programming scenarios can be accomplished. Furthermore, in the scenarios in which programmers might expect to be able to use alternative ways to complete a programming goal such as changing the morphology of the graph of persistence or modifying the moment in which application data is promoted to the persistent space, the high level abstractions provided by the Datom API simplify code updates. The fact that all the persistent elements of the API are derived from the type `PersistentItem` helps to keep the effort of changing the form of the graph of persistence low and reduces the viscosity of the code.

Moderate effort is required to modify application-specific Elements' types as updates to these components imply tracing their use through the application program. However, as in the API implementation they are all derived from the Datom Element class, not only most of the code should remain valid but also this type of Element should be easily traceable. In the worst case, change effort is directly related to the extent of the changes performed. In general, all programmer profiles prefer APIs that exhibit low viscosity; however, they can tolerate increased resistance to change at different levels.

Consistency (CONS)

The consistency dimension measures the extent to which programmers are able to infer the use of new components of an API once a part of it has been learnt. An API exhibits *arbitrary* consistency if it does not use the same design patterns and idioms when it would be appropriate to do so. If an API employs the same design patterns and idioms for two or more similar user goals but does not use the same design patterns and idioms in all appropriate circumstances, the API exposes *core* consistency. Finally, an API is *fully* consistent if it uses the same design patterns and idioms when appropriate throughout the whole of the API.

The Datom API is fully consistent as developers are able to easily reuse the knowledge obtained from coding a given programming scenario in similar cases. The inspection of code samples for similar goals, such as deleting an Element and removing a Composite Entity, modifying and setting up the graph of persistence, or querying and navigating persistent data, indicates that source code written against the Datom API consistently follows the same persistent data programming principles. Source code snippets for similar programming tasks tend to be highly comparable. In general, all types of programmers prefer APIs that are fully consistent.

Role Expressiveness (ROLE)

Role expressiveness indicates to what extent the relationship between each component exposed by an API and a program as a whole is visible. If the code created with an API to accomplish a specific programming goal cannot be interpreted correctly and does not fully match developers' expectations, the API is said to be *opaque*. When code written against an API can be interpreted correctly but does not fully match users' expectations, the role expressiveness of the API is *plausible*. Finally, the API is *transparent* if the code required to accomplish a user goal can be interpreted correctly and if it matches programmers' expectations.

Source code written with the Datom API successfully expresses its role. Visual inspection of the code samples for the typical use scenarios of the API indicates that programmers should be able to easily describe what the API components do directly from the source code. A fundamental reason for this is that the Datom API has been built on top of a group of ADTs whose semantics are familiar to most developers. Therefore, the API is mostly transparent.

However, the API fails to meet users expectations mainly in tasks involving shaping the graph of persistence. In this case, some developers may feel tempted to have more flexibility to create arbitrary connections through managed components, an expectation that the API fails to fulfil. In these cases the API is only plausible. Yet, it was a deliberate design decision to reduce the complexity of data layouts by limiting the types of object that can be used to create the persistent object networks of applications. Complex application-specific data representations remain hidden within Datom Elements. All developer profiles prefer transparent APIs, although increased levels of opaqueness are better tolerated by pragmatic and systematic programmers in that order.

Domain Correspondence (DOM)

This dimension indicates how clearly the API components map the conceptual objects that users think about manipulating when using the API. A *direct* domain correspondence occurs when the types exposed by the API map directly on to the types and concepts expected by users. If the API exposes types that map to those expected by developers only after describing the mapping, the API has *plausible* domain correspondence. Finally, if the types exposed by the API do not map directly on to the types and concepts expected by users even after describing the mapping, the API is said to have an *arbitrary* domain correspondence.

The components of the Datom API make use of popular abstract data types. This supports the provision of an extremely direct and clear correspondence between the conceptual objects that users picture and the classes and methods exposed by these components. It is unlikely that the Datom API could give place to different conceptual domain models since the API is built around components for which a prescribed behaviour is well-known and accepted by developers. Furthermore, the mapping between these objects in memory and on disk is transparent as far as programmers need to be concerned. Some programmers may find it unusual to think in the terms of a graph of persistence made of related Composite Entities and Elements, which are managed as a transparent extension of disk space. These developers would require some adjustment to properly manipulate the Datom API. With respect to this dimension, all developers prefer APIs with a direct domain correspondence.

6.3.3 Review of Findings

The Datom API may be a programming tool not favoured by systematic developers. Lacking the flexibility and power of building arbitrary networks of objects represents an annoying characteristic of the API. The Abstraction Level dimension (ABST) analysis indicates that the Datom API works at the wrong level of abstraction for systematic developers, it lacks the power and flexibility that the preferred primitive types provide. Furthermore, the API deliberately hides intricate system details about the construction and implementation of the API as evidenced by the Penetrability dimension (PENE); this has a negative impact for this type of developer as it impedes the building of trust on the API. Finally, the fact that some parts of the API cannot be easily extended or replaced when needed may irritate systematic developers; an issue exhibited by the API Elaboration dimension (ELAB).

A programming tool for persistent data access with the sort of functional characteristics of the Datom API (such as navigational data access, high-level abstraction support, and type safeness) that may represent a more appealing alternative for systematic programmers could be a full-fledged persistent programming language such as PJama. However, many intrinsic features of the PPL approach do not match desirable cognitive features of the other two group of programmers specially in the Abstraction Level (ABST), Learning Style (LEAR), Penetrability (PENE), and Work Step Unit (STEP) dimensions.

In summary, the Datom API seems to be factored for pragmatic programmers in the first place, and for opportunistic programmers in the second. The analysis of the cognitive dimensions consistently showed important similarities between the characteristic behavioural patterns of these developers' profiles and the type of code produced with the Datom API. Developers in these categories should be able to command the conceptual and implementation features of the API naturally. Furthermore, their coding experience should be comfortable to a great extent.

6.3.4 Future Versions of the Datom API

The application of the CDs framework is useful not only to evaluate the Datom API as a programming tool but also to study the consequences of potential changes in future versions of the API. The evaluation of the Datom API revealed good software attributes. In this respect, it is possible to say that the API is fully consistent (CONS) since it follows the same programming principles and design patterns when appropriate. The role expressiveness (ROLE) of the different components that comprise the API shows a high level of transparency.

Furthermore, the API components also exhibit a direct domain correspondence (DOM), giving little room for conceptual misunderstandings since these components are well-known ADTs with predefined semantics. These properties, combined with the small number of core components that comprise the Datom API, facilitate a minimal and reversible level of premature commitment (PREM) as well as a low level of viscosity (VISC) on code written with the Datom API. The aforementioned software attributes should be pursued, and maintained, in the face of any future changes to the API because they are all desirable software properties.

It has been argued before that the Datom API seems to be remarkably factored for pragmatic programmers. Thus, any future changes in the API should ideally be focused on improving the usability of the Datom API for this type of developer instead of targeting any of the other developer profiles. The use of the CDs framework revealed some areas in which the services of the API do not entirely fulfil the expectations of pragmatic developers. These issues are analysed in the following paragraphs. The CDs framework is used to illustrate the impact on the usability of the API due to potential changes to the design of the Datom API; the discussion focuses only on the relevant dimensions of the framework.

Enriching the Components of the Datom API

Pragmatic programmers mind the impact of code in the global context of the application. Accordingly, they want to be able to extend and fine-tune the standard behaviour of an API. The current version of the Datom API supports fine-tuning at the level of Datom Elements. However, there are important areas of the API that lack this ability. For example, Composite Entities cannot be customised or extended. They offer only one standard behaviour for properties such as caching, swizzling eagerness, data clustering, and optimisation of storage management. As a consequence, it is not possible to use the Datom API to adapt applications to different data access requirements.

A desirable modification to the Datom API would be to enrich the abstractions acting as Composite Entities. The main goal of this change is to give programmers the capacity to choose appropriate configuration parameters for different data access patterns. For example, programmers could be able to manipulate the amount of incremental reallocation and the trade-off between time and space costs. The API could be modified in different ways. First, the constructor methods of Composite Entities could be overloaded to give developers the ability to select the most suitable option among alternative back-end data structures (e.g. arrays, hash tables, or trees). Second, the API could be extended with methods to define the initial capacity, capacity increments, and load factors of these data structures.

With respect to pragmatic developers, these changes may have a positive effect on the usability of the API, since their main goal is to augment the penetrability (PENE) of the Datom API. They should allow programmers to understand and manipulate the effect of the API functionality in the global context of the application. These changes would modify the previous value for the working framework dimension (WORK) because they would increase the size of the conceptual chunk that developers would need to manage in order to work effectively with the modified API.

However, the new form of the API would still support the preferred learning style (LEAR) of pragmatic programmers; they should be able to process the additional concepts piece by piece and in a structured way because the changes would preserve the original relationship between the different components of the API. Furthermore, the changes would not force developers to adopt an expansive view of the API. The increase in the number of decisions that programmers face with the new API may affect the minimal level of premature commitment (PREM) currently exhibited by the Datom API. However, the modified API would retain its ability to produce reversible code to a great extent if the new API is able to produce similar code for alternative versions of a given programming task.

Adding Data Types to the Datom API

The usability study revealed that the Datom API may fail to meet the expectations of pragmatic programmers in tasks involving shaping the graph of persistence. Both data layouts and data access code are built using exclusively a minimal set of ADTs. It is likely that programmers may find this feature limiting if the data access requirements of the application cannot be fulfilled with the current set of Composite Entities. In this respect, a convenient modification to the

Datom API would be the addition of new data types to facilitate a more diverse group of data access semantics. For example, programmers may find it useful to have richer abstractions at hand such as multidimensional matrix, priority queue, different types of search trees, circular lists, etc.

The aim of this sort of modification is to augment the role expressiveness (ROLE) of the API. Greater diversity regarding the number and type of Composite Entities is likely to assist programmers in shaping the data models of applications in a more accurate way. Thus, developers should be able to interpret with more precision, and more easily, code written using the Datom API. In addition, they should also find that the new abstractions enable them to write code that matches their expectations concerning more complex data models.

The level of premature commitment (PREM) exhibited by the Datom API is likely to augment if more data types are added to the library. This dimension would also be affected by the degree to which programmers conceive a set of components as equivalent in order to solve a determined programming task; for example, using a list instead of a circular list to store a collection of objects. However, one may expect data layouts to be dictated by the data access semantics of the application reducing with it the number of decisions programmers need to make when choosing a particular component of the API. The learning requirements (LEAR) posed by the modified version of the Datom API would still demand a step-wise learning style. The amount of information that programmers would need to learn should augment linearly as new data types are incorporated. Yet, they should be able to learn them piece by piece as they are required.

Modifying and designing an API is a task in which several trade-offs have to be faced. In combination with a usability study, it is also necessary to consider implementation issues. Designers must keep in mind that one of the principal goals in the design of the Datom API is simplicity; for this reason, the addition of new abstractions and functionality should receive careful consideration, especially the impact of these new features on the performance of the library.

6.4 Summary

In this chapter I evaluated the implementation of the Datom API using three different criteria. First, I ported a couple of applications that use file-based APIs to the facilities provided by the Datom API. The portings show that the Datom API not only reduces the size of applications but also makes persistent code simpler and self-describing. Second, I measured the performance of the implementation of the Datom API. The experimental results demonstrate that the prototype's performance degrades gracefully even for a large number of data elements. Third, I assessed the usability of the API using the Cognitive Dimensions framework. This study highlights that pragmatic programmers, as well as opportunistic programmers, should be able to use the API comfortably. All together, the experimental results of this chapter demonstrate the feasibility of creating sophisticated data access strategies with minimal effort on top of the Datom API.

Chapter 7

Conclusions

In this dissertation I have studied two different techniques for decomposing file data into discernible items. The first strategy employs the functionality of the file API in order to disclose identical portions of file data. The second technique takes a more radical approach and proposes an alternative storage API for the manipulation of file contents. Experimental results in each of the two separate approaches showed that augmented levels of abstraction improve the manipulation of persistent data. In this the last chapter, I summarise the contributions of my dissertation and present opportunities for future research.

7.1 Summary

In Chapter 1 I presented the motivation of my research. I highlighted the necessity of providing better programming tools for the manipulation of file contents. I then stated my thesis, namely that programmatic APIs departing from the view of file data as a monolithic and flat object to a more abstract data representation, capable of managing a sensible amount of structure and data type, have important advantages for programmers. A data manipulation tool of this type improves software quality and supports the creation of sophisticated data access strategies with less effort.

In Chapter 2 I described current data storage paradigms. I discussed the impact that data models and programmatic APIs have on the implementation of storage systems, as well as on the typical interactions of programmers with persistent code. I highlighted that, from a programmer's point of view, persistent code in the area of file data manipulation fails to deliver appropriate levels of programmability.

In Chapter 3 I presented the first major contribution of this dissertation: a comprehensive data redundancy study of file systems contents. I introduced a series of experiments that measured the degree to which sub-file data correlations can be unveiled through duplication detection techniques that make use of the services provided by the file API.

In Chapter 4 I introduced the second major contribution of this dissertation: a novel API for the manipulation of file data called Datom. Once I had showed that the view of file contents as a flat storage space represents an expensive solution from a programmability standpoint, I proposed an API whose data model enables fine-grained data manipulation relying on a minimal set of ADTs and a typed interface to file contents. Finally, I discussed those features of Datom that make my proposal a unique programmatic API for the manipulation of persistent data.

In Chapter 5 I presented the implementation of the Datom API. I began by introducing the concept of selective reachability, i.e. the persistence model supported by the prototype. I then explained how it enhances the data model of the Datom API by enabling features that simplify the manipulation of persistent abstractions. In the rest of the chapter I concentrated on showing how the data and the persistence models are materialised in the implementation.

In Chapter 6 I presented the evaluation of the Datom API according to different criteria. First, I migrated two file-based applications on top of the implementation of the Datom API. I measured the impact on the morphology of the portings at the source-code level and compared these results with the original versions. Second, I assessed the performance of the prototype, including scalability and run-time latencies. Finally, I measured the usability of the API by analysing how well it fulfils the cognitive demands of different developer profiles based on the Cognitive Dimensions framework. I used these results not only to suggest potential changes to the Datom API but also to analyse their impact on the overall usability of the API. The results presented in this chapter demonstrate the practicality of the Datom API.

In conclusion, my thesis — that data manipulation techniques which depart from the fundamental flat file paradigm to a more abstract data representation, capable of managing a sensible amount of structure and data type, facilitate the creation of sophisticated data access strategies — is justified as follows. I evaluated redundancy detection techniques that decompose file contents into unique and identifiable items and showed how they can be used to improve the implementation of storage systems. Then I proposed and implemented a storage system API that not only decomposes file data into a minimal set of semantically rich programming abstractions, but also enforces type safeness. The evaluation of this API demonstrates that it improves software quality and enables the creation of advanced persistent data access strategies with minimal effort.

7.2 Future Research

This dissertation can be extended in a number of ways.

The study on data redundancy techniques presented in Chapter 3 demonstrates that real world data sets exhibit remarkable similarity patterns. For the techniques to be employed successfully in real systems, designers have to find the right balance between the desired redundancy detection accuracy and the overheads introduced by each technique. This dissertation thoroughly addresses the former issue. However, a complete study of storage and computational overheads on live data sets remains to be carried out. It is not yet clear at which point the use of a particular technique outperforms the others if practical issues are considered. Overheads are greatly affected by different file access patterns such as temporal and spatial locality of updates that are exhibited by different data sets, as well as by the underlying technology used to implement these techniques.

The prototype of the Datom API emphasises the provision of a fully operational implementation of a novel storage API, which provides the core features to support the persistent data manipulation ideology of the Datom API. On the other hand, there are opportunities to optimise and fine tune different aspects of the prototype for future versions of the software. For example, the study based on the Cognitive Dimensions framework suggested ways to improve the usability of the Datom API. The cognitive requirements of pragmatic programmers can be better fulfilled by augmenting the level of penetrability and role expressiveness of the API. Therefore, in Section 6.3.4 I have proposed potential changes to the design of the API that can be addressed in future versions of the library.

Aside from the engineering techniques that may be used to improve the performance of the prototype, it is clear that the most accurate measure for performance is directly related to the execution pattern of applications. In this respect, the ADTs employed by the Datom API open a window into the data access semantics of applications. The bottom line in this research direction is the incorporation of the data access semantics of applications into the internal algorithms that control the operation of the Datom API. Currently, it is not fully explored how the data model can be used as a vehicle for research into the behavioural properties of alternative data organisations. For instance, different prefetching, caching, and data clustering strategies, as well as an ad-hoc storage substrate, could be implemented and tested with the aim of exploiting the semantic knowledge provided by the abstractions of the Datom API.

A different avenue of future research is the creation of novel services that exploit the advantages provided by the Datom API. In this direction, I have already proposed two ways in which my work could be further developed. First, mobile data management creates a computing environment where data adaptation plays a fundamental role. The services provided by the Datom API can be exploited to support data transmission adaptation according to the changing networking environment [PCV03]. Second, information exposure threats can be mitigated by limiting the amount of information accessible. The inherent capabilities of the Datom API to perform information subsetting and information reduction can be used to manage the leakage of sensitive data [DP05].

This dissertation represents a step towards the more ambitious goal of fully replacing the file as the primary underlying storage abstraction. Initial research in this direction might be encouraged by the design of a storage system shell that exposes the abstractions and operations of the Datom API. A shell of this sort is interesting because it shifts the attention from the programs used to manipulate passive data objects (i.e. files) to the data itself. As a direct consequence, application programs may be moved to a second level of significance, since the shell itself opens the possibility of intelligently manipulating application data without the necessity of relying extensively on external application code.

Appendix A

Keyword Files

A.1 PCMT Keyword File Content for the Datom-Based Bibkeeper

KEYWORDS	
BibtexEntry	DatomQueueRoot
BibtexString	DatomStack
CompositeEntity	DatomStackRoot
DatomArrayMatrix	DElement
DatomArrayMatrixRoot	EmptyDatomQueueException
DatomList	EmptyDatomStackException
DatomListOutOfBoundsException	InvalidStoreException
DatomListRoot	PersistentItem
DatomMap	Root
DatomMapRoot	Store
DatomMatrixOutOfBoundsException	Unit
DatomQueue	UpdateException

Table A.1: List of the persistent keywords of the Datom-based Bibkeeper used to collect source code metrics with the PCMT tool.

A.2 PCMT Keyword File Content for the File-Based Bibkeeper

KEYWORDS		
BufferedInputStream	FilterInputStream	OutputStreamWriter
BufferedOutputStream	FilterOutputStream	PipedInputStream
BufferedReader	FilterReader	PipedOutputStream
BufferedWriter	FilterWriter	PipedReader
ByteArrayInputStream	InputStream	PipedWriter
ByteArrayOutputStream	InputStreamReader	PrintStream
CharArrayReader	InterruptedIOException	PrintWriter
CharArrayWriter	InvalidClassException	PushbackInputStream
CharConversionException	InvalidObjectException	PushbackReader
DataInput	IOException	RandomAccessFile
DataInputStream	LineNumberInputStream	Reader
DataOutput	LineNumberReader	SequenceInputStream
DataOutputStream	NotActiveException	Serializable
EOFException	NotSerializableException	SerializablePermission
Externalizable	ObjectInput	StreamCorruptedException
File	ObjectInputStream	StreamTokenizer
FileDescriptor	ObjectInputValidation	StringBufferInputStream
FileFilter	ObjectOutput	StringReader
FileInputStream	ObjectOutputStream	StringWriter
FilenameFilter	ObjectStreamClass	SyncFailedException
FileNotFoundException	ObjectStreamConstants	UnsupportedEncodingException
FileOutputStream	ObjectStreamException	UTFDataFormatException
FilePermission	ObjectStreamField	WriteAbortedException
FileReader	OptionalDataException	Writer
FileWriter	OutputStream	

Table A.2: List of the persistent keywords of the file-based Bibkeeper used to collect source code metrics with the PCMT tool.

A.3 PCMT Keyword File Content for the Datom-Based Gradebook

KEYWORDS	
CompositeEntity	DElement
DatomArrayMatrix	EmptyDatomQueueException
DatomArrayMatrixRoot	EmptyDatomStackException
DatomList	GradingModel
DatomListOutOfBoundsException	InvalidStoreException
DatomListRoot	PerAssignmentData
DatomMap	PerCompositeData
DatomMapRoot	PersistentItem
DatomMatrixOutOfBoundsException	PerStudentData
DatomQueue	Root
DatomQueueRoot	Store
DatomStack	UpdateException
DatomStackRoot	

Table A.3: List of the persistent keywords of the Datom-based Gradebook used to collect source code metrics with the PCMT tool.

A.4 PCMT Keyword File Content for the DOM-Based Gradebook

KEYWORDS		
AbstractDOMAdapter	ElementFilter	Namespace
AbstractFilter	EntityRef	OracleV1DOMAdapter
Attribute	EscapeStrategy	OracleV2DOMAdapter
BuilderErrorHandler	Filter	Parent
CDATA	Format	ProcessingInstruction
Comment	Format.TextMode	SAXBuilder
Content	IllegalAddException	SAXHandler
ContentFilter	IllegalDataException	SAXOutputter
CrimsonDOMAdapter	IllegalNameException	Text
DataConversionException	IllegalTargetException	UncheckedJDOMFactory
DefaultJDOMFactory	JAXPDOMAdapter	Verifier
DocType	JDOMException	XercesDOMAdapter
Document	JDOMFactory	XML4JDOMAdapter
DOMAdapter	JDOMLocator	XMLOutputter
DOMBuilder	JDOMParseException	XPath
DOMOutputter	JDOMResult	XSLTransformer
Element	JDOMSource	XSLTransformException

Table A.4: List of the persistent keywords of the DOM-based Gradebook used to collect source code metrics with the PCMT tool.

Appendix B

Code Samples for Usability Scenarios

1. Open a collection of persistent objects

```
try  {
   StoreConn storeConn = new StoreConn();
   DatomMapRoot rootMap = (DatomMapRoot) storeConn.openRoot("RootName");
   rootMap.close();
   storeConn.close();
}
catch(InvalidStoreException ise)  {}
catch(UpdateException ue)  {}
```

2. Prepare the persistent data layout for a collection of data items

```
try {
   String[] keys = {"Island", "Epic", "HMV", "Sony"};
   StoreConn storeConn = new StoreConn();
   DatomMapRoot rootMap = new DatomMapRoot();
   storeConn.createRoot("MusicLabels", rootMap);
   for(int i=0; i<keys.length; i++)  {
      rootMap.put(keys[i], new DatomList());
   }
   rootMap.close();
   storeConn.close();
}
catch(UpdateException ue)   {}
catch(InvalidStoreException ise)   {}
catch(NotSerializableException nse)   {}
```

3. Add a data element to the collection of persistent items

```
try   {
   StoreConn storeConn = new StoreConn();
   DatomMapRoot rootMap = (DatomMapRoot) storeConn.openRoot("MusicLabels");
   DatomList labelList = (DatomList) rootMap.get("Sony");
   // ...
   String name = "Aerosmith";
   int numHits = 14;
   Vector members = new Vector();
   members.add("Steven Tyler (born March 26, 1948), is the singer ...");
   members.add("Joe Perry (born September 10, 1950), is the lead ...");
   members.add("Tom Hamilton ...");
   members.add("Joey Krammer ...");
   Band band = new Band(name, numHits, members);
   labelList.append(band);
   rootMap.save();
   rootMap.close();
   storeConn.close();
}
catch(UpdateException ue)  {}
```

4. Update a data element in the collection of persistent items

```
try   {
   StoreConn storeConn = new StoreConn();
   DatomMapRoot rootMap = (DatomMapRoot) storeConn.openRoot("MusicLabels");
   DatomList labelList = (DatomList) rootMap.get("Sony");
   // ... getting value for bandIndex;
   int newNumHits = 16;
   Band band = (Band) labelList.get(bandIndex);
   Vector members = band.getMembers();
   band.setNumHits(newNumHits);
   members.add("Brad Withford ...");
   band.setMembers(members);
   rootMap.save();
   rootMap.close();
   storeConn.close();
}
catch(UpdateException ue)   {}
```

5. Read data from a persistent data element

```
try   {
   StoreConn storeConn = new StoreConn();
   DatomMapRoot rootMap = (DatomMapRoot) storeConn.openRoot("MusicLabels");
   DatomList labelList = (DatomList) rootMap.get("Sony");
   int numElements = labelList.size();
   for(int i=0; i<numElements; i++)   {
      Band tempBand = (Band) labelList.get(i);
      System.out.println("Band name: " + tempBand.getName());
      System.out.println("Number of hits: " + tempBand.getNumHits());
      Vector members = tempBand.getMembers();
      int numMembers = members.size();
      for(int j=0; j<numMembers; j++)   {
         System.out.println(members.get() + "\n");
      }
   }
   rootMap.close();
   storeConn.close();
}
catch(UpdateException ue)   {}
```

6. Delete a data element from the collection of persistent items

```
try   {
   StoreConn storeConn = new StoreConn();
   DatomArrayMatrixRoot gameMap =
      (DatomArrayMatrixRoot) storeConn.openRoot("MapOne");
   // ... getting value for rowIndex and colIndex
   DatomStack zoneActions = (DatomStack) gameMap.get(rowIndex, colIndex);
   if(!zoneActions.isEmpty())   {
      PlayerAction action = (PlayerAction) zoneActions.pop();
   }
   gameMap.save();
   gameMap.close();
   storeConn.close();
}
catch(UpdateException ue)   {}
```

7. Remove a collection of persistent items

```
try  {
   StoreConn storeConn = new StoreConn();
   DatomMapRoot rootMap = (DatomMapRoot) storeConn.openRoot("MusicLabels");
   DatomList labelList = (DatomList) rootMap.delete("Epic");
   rootMap.save();
   rootMap.close();
   storeConn.close();
}
catch(UpdateException ue)  {}
catch(InvalidStoreException ise)  {}
```

8. Apply updates atomically to a collection of persistent items

```
try {
   StoreConn storeConn = new StoreConn();
   DatomMapRoot tasks = (DatomMapRoot) storeConn.openRoot("Tasks");
   DatomQueue pendings = (DatomQueue) tasks.get("pending");
   DatomQueue processed = (DatomQueue) tasks.get("processed");
   while(!pendingTasks.isEmpty())  {
      Task currentTask = (Task) pendings.dequeue();
      // ... doing something with currentTask
      processed.queue(currentTask);
      tasks.save();
   }
   tasks.close();
   storeConn.close();
}
catch(UpdateException ue)  {}
```

9. Modify the organisation of the persistent data layout

```
try  {
   StoreConn storeConn = new StoreConn();
   DatomMapRoot mainStructure = (DatomMapRoot) storeConn.openRoot("main");
   // .. getting idPart
   DatomMap parts = (DatomMap) mainStructure.get(idPart);
   // .. getting idSubpart
   DatomList subparts = (DatomList) parts.delete(idSubpart);
   // .. generating newIdPart
   mainStructure.put(newIdPart, subparts);
   mainStructure.save();
   mainStructure.close();
   storeConn.close();
}
catch(UpdateException ue)  {}
```

10. Query the collection of persistent items

```
try  {
   StoreConn storeConn = new StoreConn();
   DatomMapRoot mainStructure = (DatomMapRoot) storeConn.openRoot("main");
   // ... getting id
   if(mainStructure.hasKey(id))  {
      DatomMap subparts = (DatomMap) mainStructure.get(id);
      Iterator subpartsIt = subparts.getKeys().iterator();
      while(subpartsIt.hasNext())  {
         String subPartCode = (String) subpartsIt.next();
         System.out.println("Code: " + subPartCode);
      }
   }
   else  {
      System.out.println("SORRY: No such an ID present in the structure.");
   }
   mainStructure.close();
   storeConn.close();
}
catch(UpdateException ue)  {}
```

11. Navigate and update the collection of persistent items

```
try  {
   StoreConn storeConn = new StoreConn();
   DatomMapRoot dataFrame = (DatomMapRoot) storeConn.openRoot("localData");
   Iterator dataFrameIt = dataFrame.getKeys().iterator();
   while(dataFrameIt.hasNext())  {
      String frameId = (String) dataFrameIt.next();
      DatomStack blocks = (DatomStack) dataFrame.get(frameId);
      while(!blocks.isEmpty())  {
         Block dataBlock = (Block) blocks.pop();
         // ... do something with dataBlock
         dataFrame.save();
      }
```

```
    }
    dataFrame.close();
    storeConn.close();
}
catch(UpdateException ue)   {}
```

Appendix C

Task Analysis for Usability Scenarios

1. Open a collection of persistent objects

- Open the store.
- Recover a root of persistence.
- Close the root of persistence.
- Close the store.

2. Prepare the persistent data layout for a collection of data items

- Open the store.
- Create a root of persistence in memory.
- Insert the root of persistence in the store.
- Create a set of persistent items.
- Include the set of persistent items in the graph of persistence.
- Save the graph of persistence.
- Close the root of persistence.

- Close the store.

3. Add a data element to the collection of persistent items

- Open the store.
- Recover a root of persistence.
- Create the Element.
- Reach the position to insert the Element.
- Insert the Element.
- Save the graph of persistence.
- Close the root of persistence.
- Close the store.

4. Update a data element in the collection of persistent items

- Open the store.
- Recover a root of persistence.
- Reach the position where the Element is stored.
- Fetch the Element.
- Modify the Element
- Save the graph of persistence.
- Close the root of persistence.
- Close the store.

5. Read data from a persistent data element

- Open the store.
- Recover a root of persistence.
- Reach the position where the Element is stored.
- Fetch the Element.
- Read the Element
- Close the root of persistence.
- Close the store.

6. Delete a data element from the collection of persistent items

- Open the store.
- Recover a root of persistence.
- Reach the position where the Element is stored.
- Remove the Element from the graph of persistence.
- Save the graph of persistence.
- Close the root of persistence.
- Close the store.

7. Remove a collection of persistent items

- Open the store.
- Recover a root of persistence.
- Reach the position where the Composite Entity is stored.
- Remove the Composite Entity from the graph of persistence.
- Save the graph of persistence.

- Close the root of persistence.
- Close the store.

8. Apply updates atomically to a collection of persistent items

- Open the store.
- Recover a root of persistence.
- Repeat set of operations while needed.
  - Reach the position where the persistent item is stored.
  - Update the persistent item.
  - Save the graph of persistence.
- Close the root of persistence.
- Close the store.

9. Modify the organisation of the persistent data layout

- Open the store.
- Recover a root of persistence.
- Open a persistent item located in the graph of persistence at second-level depth.
- Temporarily remove the persistent item from the graph of persistence.
- Create a third-level depth Composite Entity.
- Insert the second-level persistent item in the newly created third-level Composite Entity.
- Save the graph of persistence.
- Close the root of persistence.
- Close the store.

10. Query the collection of persistent items

- Open the store.
- Recover a root of persistence.
- Query the graph of persistence.
- Present results.
- Close the root of persistence.
- Close the store.

11. Navigate and update the collection of persistent items

- Open the store.
- Recover a root of persistence.
- Traverse and update the graph of persistence.
- Save the graph of persistence.
- Close the root of persistence.
- Close the store.

Bibliography

[ABC$^+$83] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, W. P. Cockshott, and R. Morrison. An Approach to Persistent Programming. *The Computer Journal*, 26(4):360–365, 1983.

[ABC$^+$02] S. Agarwal, J. A. Blakeley, T. Casey, K. Delaney, C. Galindo-Legaria, G. Graefe, M. Rys, and M. Zwilling. Chapter 27: Microsoft SQL Server. In A. Silberschatz, H. F. Korth, and S. Sudarshan, eds., *Database System Concepts*, pp. 969–1006. McGraw-Hill, fourth edition, 2002.

[ABD$^+$89] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The Object-Oriented Database System Manifesto. In *Proceedings of the First International Conference on Deductive and Object-Oriented Databases (DOOD '89)*, pp. 223–240. Kyoto, Japan, 1989.

[Abi97] S. Abiteboul. Querying Semi-Structured Data. In *Proceedings of 6th International Conference on Database Theory (ICDT '97)*, volume 1186 of *Lecture Notes in Computer Science*, pp. 1–18. Springer, 1997.

[ABS00] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: from Relations to Semistructured Data and XML.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[ADJ$^+$96] M. P. Atkinson, L. Daynés, M. J. Jordan, T. Printezis, and S. Spence. An Orthogonally Persistent Java. *SIGMOD Record*, 25(4):68–75, 1996.

[AGH00] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, third edition, 2000.

[AJDS96] M. P. Atkinson, M. J. Jordan, L. Daynès, and S. Spence. Design Issues for Persistent Java: A Type-Safe, Object-Oriented, Orthogonally Persistent System. In *Proceedings of the 7th Workshop on Persistent Object Systems (POS7)*, pp. 33–47. Cape May, New Jersey, USA, 1996.

[AM95] M. P. Atkinson and R. Morrison. Orthogonally Persistent Object Systems. *Very Large Data Bases Journal*, 4(3):319–401, 1995.

[ANS86] American National Standards Institute: The Database Language SQL, Document ANSI X3.135, 1986.

[ANS92] American National Standards Institute: Database Language — SQL, Document ANSI X3.135-1992, 1992.

[AQM$^+$97] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The Lorel Query Language for Semistructured Data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.

[BA03] K. Barr and K. Asanovic. Energy Aware Lossless Data Compression. In *Proceedings of the First International Conference on Mobile Systems, Applications, and Services (MobiSys '03)*. San Francisco, CA, USA, May 2003.

[Bac73] C. W. Bachman. The Programmer as Navigator. *Communications of the Association for Computing Machinery (ACM)*, 16(11):653–658, 1973.

[Ban88] F. Bancilhon. Object-Oriented Database Systems. In *Proceedings of the Seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '88)*, pp. 152–162. ACM Press, New York, NY, USA, 1988.

[BC85] A. L. Brown and W. P. Cockshott. The CPOMS Persistent Object Management System. Technical Report PPRR-13-85 (Persistent Programming Research Report 13), University of Glasgow and University of St. Andrews, Scotland, 1985.

[BCD89] F. Bancilhon, S. Cluet, and C. Delobel. A Query Language for the O_2 Object-Oriented Databases. In *Proceedings of the Second International Workshop on Database Programming Languages*, pp. 122–138. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1989.

[BCF$^+$05] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML Query Language. Technical report, World Wide Web Consortium, April 2005.

[BCG$^+$99] C. Baru, V. Chu, A. Gupta, B. Ludascher, R. Marciano, Y. Papakonstantinou, and P. Velikhov. XML-Based Information Mediation for Digital Libraries. In *Proceedings of the Fourth ACM Conference on Digital Libraries (ACM DL '99)*, pp. 214–215. ACM Press, New York, NY, USA, August 1999.

[BCGD00] W. J. Bolosky, S. Corbin, D. Goebel, and J. R. Douceur. Single Instance Storage in Windows 2000. In *Proceedings of the 4th USENIX Windows System Symposium (WinsSys '00)*. USENIX Association, August 2000.

[BDB04] Sleepycat Software, Inc. *Berkeley DB Collections Tutorial*, September 2004.

[BDHS96] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A Query Language and Optimization Techniques for Unstructured Data. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data (SIGMOD '96)*, pp. 505–516. ACM Press, New York, NY, USA, June 1996.

[BFH02] R. Bryant, R. Forester, and J. Hawkes. Filesystem Performance and Scalability in Linux 2.4.17. In *Proceedings of USENIX Annual Technical Conference, FREENIX Track (USENIX '02)*, pp. 259–274. USENIX Association, Berkeley, CA, 2002.

[BG00] A. F. Blackwell and T. R. G. Green. A Cognitive Dimensions Questionnaire Optimised for Users. In *Proceedings of Twelfth Annual Meeting of the Psychology of Programming Interest Group (PPIG-12)*, pp. 137–154. 2000.

[BHK$^+$91] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. Measurements of a Distributed File System. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles (SOSP '91)*, pp. 198–212. ACM Press, New York, NY, USA, 1991.

[BIB05] Bibkeeper, 2005. Available at `http://bibkeeper.sourceforge.net` [accessed May, 2005].

[BKKM00] S. Banerjee, V. Krishnamurthy, M. Krishnaprasad, and R. Murthy. Oracle8i - The XML Enabled Data Management System. In *Proceedings of the Sixteenth International Conference on Data Engineering (ICDE '00)*, pp. 561–568. San Diego, CA, USA, March 2000.

[BM93] E. Bertino and L. Martino. *Object-Oriented Database Systems: Concepts and Architectures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993.

[BMO$^+$89] R. Bretl, D. Maier, A. Otis, D. J. Penney, B. Schuchardt, J. Stein, E. H. Williams, and M. Williams. The GemStone Data Management System. In *Object-Oriented Concepts, Databases, and Applications*, pp. 283–308. ACM Press and Addison-Wesley, 1989.

[BNTW95] P. Buneman, S. Naqvi, V. Tannen, and L. Wong. Principles of Programming with Complex Objects and Collection Types. In *Selected Papers of the Fourth International Conference on Database Theory (ICDT '92)*, pp. 3–48. Elsevier Science Publishers B. V., Amsterdam, The Netherlands, 1995.

[BOS91] P. Butterworth, A. Otis, and J. Stein. The GemStone Object Database Management System. *Communications of the Association for Computing Machinery (ACM)*, 34(10):64–77, 1991.

[BP98] S. Balasubramaniam and B. C. Pierce. What is a File Synchronizer? In *Proceedings of the Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM '98)*, pp. 98–108. Dallas, Texas, USA, October 1998.

[BPS$^+$04] T. Bray, J. Paoli, C. M. Sperberg, E. Maler, F. Yergeau, and J. Cowan. Extensible Markup Language (XML) 1.1. Technical report, World Wide Web Consortium, February 2004.

[Bra03] T. Bray. Why XML Doesn't Suck, March 2003. Available at `http://tbray.org/ongoing/When/200x/2003/03/24/XMLisOK`, [accessed July, 2005].

[Bro93] A. Z. Broder. Some Applications of Rabin's Fingerprinting Method. In R. Capocelli, A. D. Santis, and U. Vaccaro, eds., *Sequences II: Methods in Communications , Security, and Computer Science*, pp. 143–152. Springer-Verlag, 1993.

[Bro94] K. Brockschmidt. *Inside OLE 2*. Microsoft Press, 1994.

[Bro97] A. Z. Broder. On the Resemblance and Containment of Documents. In *Proceedings of Compression and Complexity of Sequences (SEQUENCES '97)*. IEEE Computer Society, Washington, DC, USA, 1997.

[Bro00] A. Z. Broder. Identifying and Filtering Near-Duplicate Documents. In *Proceedings of the 11th Annual Symposium of Combinatorial Pattern Matching (CPM '00)*. Montreal, Canada, June 2000.

[Bun97] P. Buneman. Semistructured Data. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '97)*, pp. 117–121. ACM Press, New York, NY, USA, 1997.

[CAC$^+$90] W. P. Cockshott, M. P. Atkinson, K. J. Chisholm, P. J. Bailey, and R. Morrison. Persistent Object Management System. In S. B. Zdonik and D. Maier, eds., *Readings in Object-Oriented Database Systems*, pp. 251–272. Kaufmann, San Mateo, CA, 1990.

[Car86] L. Cardelli. Amber. In *Proceedings of the Thirteenth Spring School of the LITP on Combinators and Functional Programming Languages*, pp. 21–47. Springer-Verlag Inc., New York, NY, USA, 1986.

[CBB$^+$00] R. G. G. Cattell, D. K. Barry, M. Berler, J. Eastman, D. Jordan, C. Russell, O. Schadow, T. Stanienda, and F. Velez. *The Object Data Standard: ODMG 3.0*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[CD99] J. Clark and S. DeRose. XML Path Language (XPath) Version 1.0. Technical report, World Wide Web Consortium, November 1999.

[CDF$^+$86] M. J. Carey, D. J. DeWitt, D. Frank, M. Muralikrishna, G. Graefe, J. E. Richardson, and E. J. Shekita. The Architecture of the EXODUS Extensible DBMS. In *Proceedings of the 1986 International Workshop on Object-Oriented Database Systems*, pp. 52–65. IEEE Computer Society Press, Los Alamitos, CA, USA, 1986.

[CDG$^+$90] M. J. Carey, D. J. DeWitt, G. Graefe, D. M. Haight, J. E. Richardson, D. T. Schuh, E. J. Shekita, and S. Vandenberg. The EXODUS Extensible DBMS Project: An Overview. In D. Maier and S. Zdonik, ed., *Readings on Object-Oriented Database Systems*. Morgan Kaufmann, San Mateo, CA, 1990.

[CL01] C. Chan and H. Lu. Fingerprinting Using Polynomial (Rabin's Method). Faculty of Science, University of Alberta, CMPUT690 Term Project, 2001.

[Cla99] J. Clark. XSL Transformations (XSLT) Version 1.0. Technical report, World Wide Web Consortium, November 1999.

[Cla01] S. Clarke. Evaluating a New Programming Language. In *Proceedings of the 13th Annual Workshop of the Psychology of Programming Interest Group (PPIG-13)*, pp. 275–289. 2001.

[Cla03] S. Clarke. Using the Cognitive Dimensions Framework to Design Usable APIs, 2003. Available at `http://blogs.msdn.com/stevencl/archive/2003/11/14/57065.aspx` [accessed May, 2005].

[Cla04] S. Clarke. Measuring API Usability. *Dr. Dobb's Journal Special Windows/.NET Supplement*, pp. S6–S9, May 2004.

[CM82] A. J. Cole and R. Morrison. *An Introduction to Programming with S-ALGOL*. Cambridge University Press, New York, NY, USA, 1982.

[CM84] G. Copeland and D. Maier. Making Smalltalk a Database System. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data (SIGMOD '84)*, pp. 316–325. ACM Press, New York, NY, USA, 1984.

[CMN02] L. P. Cox, C. D. Murray, and B. D. Noble. Pastiche: Making Backup Cheap and Easy. *ACM SIGOPS Operating Systems Review*, 36(SI):285–298, 2002.

[CNTR97] V. Cahill, P. Nixon, B. Tangney, and F. Rabhi. Object Models for Distributed or Persistent Programming. *The Computer Journal*, 40(8):513–527, 1997.

[CO96] A. B. Chaudhri and P. Osmon. A Comparative Evaluation of the Major Commercial Object and Object-Relational DBMSs: GemStone, O_2, Objectivity/DB, ObjectStore, VERSANT ODBMS, Illustra, Odapter and UniSQL, 1996. Not published. Available at `http://citeseer.ist.psu.edu/280722.html` [accessed May, 2005].

[Cod70] E. F. Codd. A Relational Model for Large Shared Databanks. *Communications of the Association for Computing Machinery (ACM)*, 13(6):377–387, June 1970.

[Cod82] E. F. Codd. Relational Database: A Practical Foundation for Productivity. *Communications of the Association for Computing Machinery (ACM)*, 25(2):109–117, 1982.

[CPS95] B. Callaghan, B. Pawlowski, and P. Staubach. *NFS Version 3 Protocol Specification. RFC 1813.*. Sun Microsystems, Inc., June 1995.

[CRF01] D. D. Chamberlin, J. Robie, and D. Florescu. Quilt: An XML Query Language for Heterogeneous Data Sources. In *Selected Papers from the Third International Workshop WebDB 2000 on The World Wide Web and Databases*, pp. 1–25. Springer-Verlag, London, UK, 2001.

[CS92] R. G. G. Cattell and J. Skeen. Object Operations Benchmark. *ACM Transactions on Database Systems (TODS)*, 17(1):1–31, 1992.

[CX00] J. M. Cheng and J. Xu. XML and DB2. In *Proceedings of the Sixteenth International Conference on Data Engineering (ICDE '00)*, pp. 569–573. San Diego, CA, USA, 2000.

[DA97] L. Daynès and M. Atkinson. Main-Memory Management to Support Orthogonal Persistence for Java. In *Proceedings of the Second International Workshop on Persistence and Java (PJW2)*. Half Moon Bay, CA, USA, 1997.

[DB99] J. R. Douceur and W. J. Bolosky. A Large-Scale Study of File-System Contents. In *Proceedings of the 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '99)*, pp. 59–70. ACM Press, New York, NY, USA, 1999.

[DB205] IBM DB2 Information Management Software, 2005. Web site at `http://www.ibm.com/software/data` [accessed May, 2005].

[DCBM89] A. Dearle, R. Connor, F. Brown, and R. Morrison. Napier88 – A Database Programming Language? In *Proceedings of the Second International Workshop on Database Programming Languages (DBPL '89)*, pp. 213–229. Salishan Lodge, Gleneden Beach, Oregon, June 1989.

[DD95] H. Darwen and C. J. Date. The Third Manifesto. *ACM SIGMOD Record*, 24(1):39–49, 1995.

[Deu91] O. Deux. The O_2 System. *Communications of the Association for Computing Machinery (ACM)*, 34(10):34–48, 1991.

[DFF$^+$99] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A Query Language for XML. In *Proceeding of the Eighth International Conference on World Wide Web (WWW '99)*, pp. 1155–1169. Elsevier North-Holland, Inc., Toronto, Canada, 1999.

[DFS99] A. Deutsch, M. Fernandez, and D. Suciu. Storing Semistructured Data with STORED. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data (SIGMOD '99)*, pp. 431–442. ACM Press, New York, NY, USA, 1999.

[DH03] T. E. Denehy and W. W. Hsu. Duplicate management for reference data. Research Report RJ 10305 (A0310-017), IBM, October 2003.

[DI03] F. Douglis and A. Iyengar. Application-Specific Delta-Encoding via Resemblance Detection. In *Proceedings of 2003 USENIX Technical Conference (USENIX '03)*, pp. 113–126. USENIX Association, 2003.

[DJ96] A. D'Andrea and P. Janus. UniSQL's Next-Generation Object-Relational Database Management System. *SIGMOD Record*, 25(3):70–76, 1996.

[DMOW05] S. DeRose, E. Maler, D. Orchard, and N. Walsh. XML Linking Language (XLink) Version 1.1. Technical report, World Wide Web Consortium, April 2005.

[DN65] R. C. Daley and P. G. Neuman. A General Purpose File System for Secondary Storage. In *Proceedings of AFIPS Fall Joint Computer Conference*, pp. 213–229. Spartan Books, New York, USA, 1965.

[DP05] B. Dragovic and C. Policroniades. Information SeeSaw: Availability vs. Security Management in the UbiComp World. In *Proceedings of the 2nd VLDB Workshop on Secure Data Management (SDM '05)*, volume 3674 of *Lecture Notes in Computer Science*, pp. 200–216. Springer-Verlag, Berlin Heidelberg, Germany, 2005.

[DSRS01] N. N. Dalvi, S. K. Sanghai, P. Roy, and S. Sudarshan. Pipelining in Multi-Query Optimization. In *Proceedings of the Twentieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS '01)*, pp. 59–70. ACM Press, New York, NY, USA, 2001.

[DYC95] A. Dan, P. S. Yu, and J. Y. Chung. Characterization of Database Access Pattern for Analytic Prediction of Buffer Hit Probability. *The International Journal on Very Large Data Bases*, 4(1):127–154, 1995.

[EM99] A. Eisenberg and J. Melton. SQLJ-Part 1: SQL Routines Using the Java Programming Language. *SIGMOD Record*, 28(4):58–63, 1999.

[EN00a] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*, chapter Appendix D: An Overview of the Hierarchical Data Model, pp. 941–955. Addison-Wesley, third edition, 2000.

[EN00b] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*, chapter Appendix C: An Overview of the Network Data Model, pp. 917–940. Addison-Wesley, third edition, 2000.

[EN00c] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley, third edition, 2000.

[ES90] M. A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[FEB03] M. Fisher, J. Ellis, and J. Bruce. *JDBC API Tutorial and Reference*. Sun, third edition, 2003.

[FW04] D. C. Fallside and P. Walmsley. XML Schema Part 0: Primer Second Edition. Technical report, World Wide Web Consortium, October 2004.

[FZT+92] M. J. Franklin, M. J. Zwilling, C. K. Tan, M. J. Carey, and D. J. DeWitt. Crash Recovery in Client-Server EXODUS. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data (SIGMOD '92)*, pp. 165–174. ACM Press, New York, NY, USA, 1992.

[GBHC00] S. Gribble, E. Brewer, M. Hellerstein, and D. Culler. Scalable, Distributed Data Structures for Internet Service Construction. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI '00)*, pp. 319–332. USENIX Association, October 2000.

[Gei95] K. Geiger. *Inside ODBC*. Microsoft Press, 1995.

[GEM05] GemStone Systems, 2005. Web site at `http://www.gemstone.com` [accessed May, 2005].

[GMMW03] P. Grosso, E. Maler, J. Marsh, and N. Walsh. XPointer Framework. Technical report, World Wide Web Consortium, March 2003.

[GP96] T. R. G. Green and M. Petre. Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. *Journal of Visual Languages and Computing*, 7(2):131–174, 1996.

[GR83] A. Goldberg and D. Robson. *SmallTalk-80: The Language and its Implementation*. Addison-Wesley, Boston, MA, USA, 1983.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[Gra93] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[GRA05] The Tab Completion Grade Book, 2005. Available at `http://tabgradebook.sourceforge.net/` [accessed July, 2005].

[Gri97] S. Grimstad. Persistent Code Measurement Tool (PCMT), 1997. Available at `http://www.ifi.uio.no/~pjama/pcmt/` [accessed May, 2005].

[GRR$^+$98] R. G. Guy, P. L. Reiher, D. Ratner, M. Gunter, W. Ma, and G. J. Popek. Rumor: Mobile Data Access Through Optimistic Peer-to-Peer Replication. In *Proceedings of Advances in Database Technologies, ER '98 Workshop on Mobile Data Access*, volume 1552 of *Lecture Notes in Computer Science*, pp. 254–265. Springer, 1998.

[GSAW98] S. Grimstad, D. I. K. Sjøberg, M. Atkinson, and R. Welland. Evaluating Usability Aspects of PJama Based on Source Code Measurements. In *Proceedings of the 8th International Workshop on Persistent Object Systems (POS8) and of the 3rd International Workshop on Persistence and Java (PJW3)*, pp. 307–321. Morgan Kaufmann, San Francisco, CA, USA, 1998.

[Gut02] J. V. Guttag. *Software Pioneers: Contributions to Software Engineering*, chapter Abstract Data Types, Then and Now, pp. 453–479. Springer-Verlag, Inc., New York, NY, USA, 2002.

[HB05] A. Hejlsberg and D. Box. The LINQ Project .NET Language Integrated Query, 2005. Available at `http://msdn.microsoft.com/netframework/future/linq/` [accessed November, 2005].

[Hen03] V. Henson. An Analysis of Compare-by-Hash. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS-IX)*. Lihue, Hawaii, USA, May 2003.

[Her03] J. D. Herrington. Code Generation: The One Page Guide. Code Generation Network and DevX, 2003. Available at `http://www.codegeneration.net/files/JavaOne_OnePageGuide_v1.pdf` [accessed November, 2005].

[HHW$^+$04] A. L. Hors, P. L. Hégaret, L. Wood, G. Nicol, J. Robie, M. Champion, and S. Byrne. Document Object Model (DOM) Level 3 Core Specification Version 1.0. Technical report, World Wide Web Consortium, April 2004.

[Hos95] A. L. Hosking. Benchmarking Persistent Programming Languages: Quantifying the Language/Database Interface. In *Proceedings of ACM OOPSLA'95 Workshop on Database Behaviour, Benchmarks, and Performance*. Austin, Texas, USA, 1995.

[HP01] H. Hosoya and B. C. Pierce. XDuce: A Typed XML Processing Language (Preliminary Report). In *Selected Papers from the Third International Workshop WebDB 2000 on The World Wide Web and Databases*, pp. 226–244. Springer-Verlag, London, UK, 2001.

[INF05] IBM Software - Informix product family, 2005. Web site at `http://www-306.ibm.com/software/data/informix/` [accessed May, 2005].

[ISO99] International Organization for Standardization. Information Technology – Database Languages – SQL – Part 2: Foundation (SQL/Foundation), 1999.

[JA98] M. J. Jordan and M. P. Atkinson. Orthogonal Persistence for Java^{TM}- A Mid-Term Report. In *Proceedings of the 8th International Workshop on Persistent Object Systems (POS8) and of the 3rd International Workshop on Persistence and Java (PJW3)*, pp. 335–352. Morgan Kaufmann, San Francisco, CA, USA, 1998.

[Jak02] H. Jakobsson. Chapter 25: Oracle. In A. Silberschatz, H. F. Korth, and S. Sudarshan, eds., *Database System Concepts*, pp. 921–947. McGraw-Hill, fourth edition, 2002.

[JAKC+02] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. V. S. Lakshmanan, A. Nierman, S. Paparizos, J. M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. TIMBER: A Native XML Database. *The International Journal on Very Large Data Bases*, 11(4):274–291, 2002.

[JK84] M. Jarke and J. Koch. Query Optimization in Database Systems. *ACM Computing Surveys*, 16(2):111–152, 1984.

[Jor96] M. Jordan. Early Experiences with Persistent Java. In *Proceedings of the First International Workshop on Persistence and Java (PJW1)*. Drymen, Scotland, UK, September 1996.

[Kak98] S. V. Kakkad. Address Translation and Storage Management for Persistent Object Stores. Technical Report CS-TR-98-07, Department of Computer Sciences. University of Texas at Austin, March 1998.

[KC86] S. Khoshafian and G. P. Copeland. Object Identity. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'86)*, pp. 406–416. 1986.

[KDLT04] P. Kulkarni, F. Douglis, J. LaVoie, and J. M. Tracey. Redundancy Elimination Within Large Collections of Files. In *Proceedings of 2004 USENIX Technical Conference (USENIX '04)*. USENIX Association, 2004.

[KH98] V. Kumar and M. Hsu. *Recovery mechanisms in database systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998.

[KKS92] M. Kifer, W. Kim, and Y. Sagiv. Querying Object-Oriented Databases. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data (SIGMOD '92)*, pp. 393–402. ACM Press, New York, NY, USA, 1992.

[Kle86] S. R. Kleiman. Vnodes: An Architecture for Multiple File System Types in SUN UNIX. In *Proceedings of USENIX Summer Technical Conference (USENIX '86)*. USENIX Association, 1986.

[KLMW90] B. Kent, P. Lyngback, S. Mathur, and K. Wilkinson. The Iris Database System. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data (SIGMOD '90)*, p. 392. ACM Press, New York, NY, USA, 1990.

[KM00] C.-C. Kanne and G. Moerkotte. Efficient Storage of XML Data. In *Proceedings of the International Conference on Data Engineering (ICDE '00)*, p. 198. 2000.

[KR81] R. M. Karp and M. O. Rabin. Efficient Randomised Pattern Matching Algorithms. Technical Report TR-31-81, Center for Research in Computing Technology, Harvard University, 1981.

[KR82] J. L. Keedy and I. Richards. A Software Engineering View of Files. *Australian Computer Journal*, 14(2):56–61, May 1982.

[KR01] L. Khan and Y. Rao. A Performance Evaluation of Storing XML Data in Relational Database Management Systems. In *Proceedings of the 3rd International Workshop on Web Information and Data Management (WIDM '01)*, pp. 31–38. ACM Press, New York, NY, USA, 2001.

[KS90] H. F. Korth and G. D. Speegle. Long-Duration Transactions in Software Design Projects. In *Proceedings of the Sixth International Conference on Data Engineering (ICDE '90)*, pp. 568–574. IEEE Computer Society, Washington, DC, USA, 1990.

[Lam05] C. Lamb. High-Performance Data Management in Java. *Dr. Dobb's Journal*, pp. 45–49, July 2005.

[LGWJ01] T. Lahiri, A. Ganesh, R. Weiss, and A. Joshi. Fast-Start: Quick Fault Recovery in Oracle. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data (SIGMOD '01)*, pp. 593–598. ACM Press, New York, NY, USA, 2001.

[Lis93] B. Liskov. A History of CLU. In *Proceedings of the Second ACM SIGPLAN Conference on History of Programming Languages (HOPL-II)*, pp. 133–147. ACM Press, New York, NY, USA, 1993.

[LLOW91] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore Database System. *Communications of the Association for Computing Machinery (ACM)*, 34(10):50–63, 1991.

[LOS02] Y. Leontiev, M. T. Ozsu, and D. Szafron. On Type Systems for Object-Oriented Database Programming Languages. *ACM Computing Surveys*, 34(4):409–449, 2002.

[Man94] U. Manber. Finding Similar Files in a Large File System. In *Proceedings of the USENIX Winter 1994 Technical Conference (USENIX '94)*, pp. 1–10. USENIX Association, 1994.

[McC97] S. McClure. Object Database vs. Object-Relational Databases. International Data Corporation. Bulletin No. 14821E, August 1997.

[McG77] W. C. McGee. The Information Management System IMS/VS Part I: General Structure and Operation. *IBM Systems Journal*, 16(2):84–95, 1977.

[MCK04] J. C. Mogul, Y. M. Chan, and T. Kelly. Design, Implementation, and Evaluation of Duplicate Transfer Detection in HTTP. In *Proceedings of the First Symposium on Networked Systems Design and Implementation (NSDI '04)*, pp. 43–56. San Francisco, CA, USA, March 2004.

[MCM01] A. Muthitacharoen, B. Chen, and D. Maziéres. A Low-Bandwidth Network File System. In *Proceedings of the Symposium on Operating Systems Principles (SOSP '01)*, pp. 174–187. ACM Press, New York, NY, USA, 2001.

[Mil84] R. Milner. A Proposal for Standard ML. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, pp. 184–197. ACM Press, New York, NY, USA, 1984.

[Mil99] K. W. Miller. The Long Transaction and Work Management Integration. In *Proceedings of the Twenthy-Second Annual Conference of the Geospatial Information and Technology Association (GITA '99)*. 1999. Online proceedings. Available at `http://www.gisdevelopment.net/proceedings/gita/1999/work/wm077.shtml` [accessed May, 2005].

[MMN$^+$04] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou. Boxwood: Abstractions as the Foundation for Storage Infrastructure. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, pp. 105–120. USENIX Association, December 2004.

[Moo91] J. W. Moore. The ANSI Binding of SQL to ADA. *ACM SIG Ada Letters*, XI(5):47–61, 1991.

[Mor02] T. Moreton. *Pasta: A Distributed Scalable File System for the Pastry Routing Substrate*. Part II Project. University of Cambridge, Computer Laboratory, 2002.

[MPH02] T. D. Moreton, I. A. Pratt, and T. L. Harris. Storage, Mutability and Naming in Pasta. In *Proceedings ot the NETWORKING 2002 Workshops on Web Engineering and Peer-to-Peer Computing*, pp. 215–219. Springer-Verlag, London, UK, May 2002.

[MS03] E. Meijer and W. Schulte. Unifying Tables, Objects and Documents. In *Proceedings of the Workshop on Declarative Programming in the Context of Object Oriented Languages (DP-COOL 2003)*. Uppsala, Sweden, August 2003.

[MSOP86] D. Maier, J. Stein, A. Otis, and A. Purdy. Development of an Object-Oriented DBMS. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '86)*, pp. 472–482. ACM Press, New York, NY, USA, 1986.

[MT03] R. J. T. Morris and J. Truskowski. The Evolution of Storage Systems. *IBM Systems Journal*, 42(2):205–217, 2003.

[MYS05] MySQL: The World's Most Popular Open Source Database, 2005. Web site at `http://www.mysql.com` [accessed May, 2005].

[Nag97] R. Nagar. *Windows NT File System Internals*. O'Reilly and Associates, September 1997.

[NET05] .NET Framework Developer Center, 2005. Available at `http://msdn.microsoft.com/netframework/` [accessed May, 2005].

[Nut00] G. Nutt. *Operating Systems. A Modern Perspective*, chapter 13 File Management, p. 360. Addison Wesley Longman, Inc., second edition, 2000.

[OBJ00] Objectivity, Inc. *Objectivity/C++ Standard Template Library, Release 6.0*, August 2000.

[OBJ05] ObjectStore Enterprise: Object Database Managament, 2005. Web site at http://www.progress.com/realtime/products/objectstore/index.ssp [accessed May, 2005].

[OBS99] M. A. Olson, K. Bostic, and M. I. Seltzer. Berkeley DB. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track (USENIX '99)*, pp. 183–191. USENIX Association, 1999.

[ODM05] ODMG. Object Data Management Group. The Standard for Storing Objects, 2005. Web site at http://www.odmg.org [accessed May, 2005].

[OMG05] OMG. Object Management Group, 2005. Web site at http://www.omg.org [accessed May, 2005].

[ORA05] Oracle Corporation, 2005. Web site at http://www.oracle.com [accessed May, 2005].

[PAD$^+$97] T. Printezis, M. P. Atkinson, L. Daynès, S. Spence, and P. Bailey. The Design of a New Persistent Object Store for PJama. In *Proceedings of the Second International Workshop on Persistence and Java (PJW2)*. Half Moon Bay, CA, USA, 1997.

[Pad02] S. Padmanabhan. Chapter 26: IBM DB2 Universal Database. In A. Silberschatz, H. F. Korth, and S. Sudarshan, eds., *Database System Concepts*, pp. 949–967. McGraw-Hill, fourth edition, 2002.

[PAKC$^+$03] S. Paparizos, S. Al-Khalifa, A. Chapman, H. V. Jagadish, L. V. S. Lakshmanan, A. Nierman, J. M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. TIMBER: a Native System for Querying XML. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD '03)*, pp. 672–672. ACM Press, New York, NY, USA, 2003.

[PCV03] C. Policroniades, R. Chakravorty, and P. Vidales. A Data Repository for Fine-Grained Adaptation in Heterogeneous Environments. In *Proceedings of the 3rd ACM International Workshop on Data Engineering for Wireless and Mobile Access (MobiDe '03)*, pp. 51–55. ACM Press, New York, NY, USA, 2003.

[PGMW95] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object Exchange Across Heterogeneous Information Sources. In *Proceedings of the 11th International Conference on Data Engineering (ICDE '95)*, pp. 251–260. IEEE Computer Society, Washington, DC, USA, 1995.

[PP04] C. Policroniades and I. Pratt. Alternatives for Detecting Redundancy in Storage Systems Data. In *Proceedings of the USENIX Annual Technical Conference (USENIX '04)*, pp. 73–86. USENIX Association, Boston, MA, USA, 2004.

[PP05] C. Policroniades and I. Pratt. Datom: An Abstract View of File Content. In *Proceedings of the Postgraduate Research Conference in Electronics, Photonics, Communications and Networks, and Computing Science (PREP 2005)*, pp. 150–151. Lancaster, UK, 2005.

[PRO02a] Pro*C/C++ Precompiler Programmer's Guide. Release 9.2. Part Number A97269-01, 2002.

[PRO02b] Pro*COBOL Precompiler Programmer's Guide. Release 9.2. Part Number A96109-01, 2002.

[PV04] B. C. Pierce and J. Vouillon. What's in Unison? A Formal Specification and Reference Implementation of a File Synchronizer. Technical Report MS-CIS-03-36, Departament of Computer and Information Science, University of Pennsylvania, 2004.

[QD02] S. Quinlan and S. Dorward. Venti: a New Approach to Archival Storage. In *Proceedings of the First USENIX Conference on File and Storage Technologies (FAST '02)*. USENIX Association, January 2002.

[Rab81] M. O. Rabin. Fingerprinting by Random Polynomials. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[Rab85] M. O. Rabin. Discovering Repetitions in Strings. In A. Apostolico and Z. Galil, eds., *Combinatorial Algorithms on Words*, pp. 279–288. Springer-Verlag, Berlin, 1985.

[RC89] J. E. Richardson and M. J. Carey. Persistence in the E Language: Issues and Implementation. *Software Practice and Experience*, 19(12):1115–1150, 1989.

[RCS93] J. E. Richardson, M. J. Carey, and D. T. Schuh. The Design of the E Programming Language. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(3):494–534, 1993.

[Rec05] B. Rector. Introducing Longhorn for Developers, 2005. Available at `http://msdn.microsoft.com/Longhorn/understanding/books/rector/default.aspx` [accessed May, 2005].

[Ric90] J. E. Richardson. Compiled Item Faulting: A New Technique for Managing I/O in a Persistent Language. In A. Dearle, G. M. Shaw, and S. B. Zdonik, eds., *Implementing Persistent Object Bases, Principles and Practice. Proceedings of the Fourth International Workshop on Persistent Objects (POS4)*, pp. 3–16. Morgan Kaufmann, 1990.

[RLA00] D. Roselli, J. R. Lorch, and T. E. Anderson. A Comparison of File System Workloads. In *Proceedings of 2000 USENIX Annual Technical Conference (USENIX '00)*. USENIX Association, June 2000.

[RLB03] S. C. Rhea, K. Liang, and E. Brewer. Value-Based Web Caching. In *Proceedings of the 12th International Conference on World Wide Web (WWW '03)*, pp. 619–628. ACM Press, New York, NY, USA, 2003.

[RLS98] J. Robie, J. Lapp, and D. Schach. XML Query Language (XQL). In *Proceedings of the Query Languages Workshop (QL '98)*. Boston, MA, USA, December 1998.

[RP02] K. Runapongsa and J. M. Patel. Storing and Querying XML Data in Object-Relational DBMSs. In *Proceedings of the Workshops XMLDM, MDDE, and YRWS on XML-Based Data Management and Multimedia Engineering-Revised Papers (EDBT '02)*, pp. 266–285. Springer-Verlag, London, UK, 2002.

[RSSB00] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and Extensible Algorithms for Multi Query Optimization. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD '00)*, pp. 249–260. ACM Press, New York, NY, USA, 2000.

[Rys01] M. Rys. Bringing the Internet to Your Database: Using SQLServer 2000 and XML to Build Loosely-Coupled Systems. In *Proceedings of the 17th International Conference on Data Engineering (ICDE '01)*, pp. 465–472. IEEE Computer Society, Washington, DC, USA, 2001.

[San86] R. Sandberg. The Sun Network Filesystem: Design, Implementation, and Experience. In *Proceedings of the Summer 1986 USENIX Technical Conference and Exhibition (USENIX '86)*. USENIX Association, 1986.

[SAS$^+$96] J. Sidell, P. M. Aoki, A. Sah, C. Staelin, M. Stonebraker, and A. Yu. Data Replication in Mariposa. In *Proceedings of the 12th International Conference on Data Engineering (ICDE '96)*, pp. 485–494. IEEE Computer Society, Washington, DC, USA, 1996.

[Sat89] M. Satyanarayanan. A Survey of Distributed File Systems. Technical Report CMU-CS-89-116, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1989.

[SAX05] Simple API for XML (SAX), 2005. Web site at `http://www.saxproject.org/` [accessed May, 2005].

[SCD90] D. T. Schuh, M. J. Carey, and D. J. DeWitt. Persistence in E Revisited - Implementation Experiences. In A. Dearle, G. M. Shaw, and S. B. Zdonik, eds., *Implementing Persistent Object Bases, Principles and Practice. Proceedings of the Fourth International Workshop on Persistent Objects (POS4)*, pp. 345–359. Morgan Kaufmann, 1990.

[Sch77] J. W. Schmidt. Some High Level Language Constructs for Data of Type Relation. *ACM Transactions on Database Systems (TODS)*, 2(3):247–261, 1977.

[Sch01] H. Schöning. Tamino - A DBMS Designed for XML. In *Proceedings of the 17th International Conference on Data Engineering (ICDE '01)*, pp. 149–154. IEEE Computer Society, Washington, DC, USA, 2001.

[SCP$^+$02] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the Migration of Virtual Computers. *ACM SIGOPS Operating Systems Review*, 36(SI):377–390, 2002.

[Ser90] Servio Logic Development Corporation. *Gemstone V 2.0: Programming in OPAL*, 1990.

[SGI05] Silicon Graphics, Inc. *Standard Template Library Programmer's Guide*, May 2005. Available at `http://www.sgi.com/tech/stl/index.html` [accessed May, 2005].

[SGT$^+$99] J. Shanmugasundaram, H. Gang, K. Tufte, C. Zhang, D. DeWitt, and J. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *Proceedings of 25th International Conference on Very Large Data Bases (VLDB '99)*, pp. 302–314. Morgan Kaufmann, September 1999.

[SHA95] National Institute of Standards and Technology. *FIPS Publication 180-1: Secure Hash Standard*, 1995.

[SK91] M. Stonebraker and G. Kemnitz. The POSTGRES Next Generation Database Management System. *Communications of the Association for Computing Machinery (ACM)*, 34(10):78–92, 1991.

[SKW92] V. Singhal, S. V. Kakkad, and P. R. Wilson. Texas: An Efficient, Portable Persistent Store. In A. Albano and R. Morrison, eds., *Proceedings of the Fifth International Workshop on Persistent Object Systems (POS5)*, pp. 11–13. Springer-Verlag, 1992.

[SMK$^+$93] M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler. Lightweight Recoverable Virtual Memory. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP '93)*, pp. 146–160. ACM Press, New York, NY, USA, 1993.

[Sol92] V. Soloviev. An Overview of Three Commercial Object-Oriented Database Management Systems: ONTOS, ObjectStore, and O_2. *ACM SIGMOD Record*, 21(1):93–104, 1992.

[SQL05] SQL Server Home, 2005. Web site at `http://www.microsoft.com/sql` [accessed May, 2005].

[SRL$^+$90] M. Stonebraker, L. A. Rowe, B. Lindsay, J. Gray, M. Carey, M. Brodie, P. Bernstain, and D. Beech. Third-Generation Data Base System Manifesto. *ACM SIGMOD Record*, 19(3):31–44, September 1990.

[SSU91] A. Silberschatz, M. Stonebraker, and J. Ullman. Database Systems: Achievements and Opportunities. *Communications of the Association for Computing Machinery (ACM)*, 34(10):110–120, 1991.

[Suc98] D. Suciu. An Overview of Semistructured Data. *ACM SIGACT News*, 29(4):28–38, 1998.

[SW00] N. T. Spring and D. Wetherall. A Protocol-Independent Technique for Eliminating Redundant Network Traffic. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '00)*, pp. 87–95. ACM Press, New York, NY, USA, 2000.

[SYB05] Sybase Inc - Sybase, 2005. Web site at `http://www.sybase.com` [accessed May, 2005].

[SYU99] T. Shimura, M. Yoshikawa, and S. Uemura. Storage and Retrieval of XML Documents Using Object-Relational Databases. In *Proceedings of the 10th International Conference on Database and Expert Systems Applications (DEXA '99)*, pp. 206–217. Springer-Verlag, London, UK, 1999.

[Tex96] Texas Persistent Store Source Code Distribution and Documentation, 1996. Available at `ftp://ftp.cs.utexas.edu/pub/garbage/texas/tdesc-4.15.1.tar.Z` [accessed May, 2005].

[Tri00] A. Tridgell. *Efficient Algorithms for Sorting and Synchronization*. Ph.D. thesis, Australian National University, April 2000.

[Tuk01] M. Tukiainen. Evaluation of the Cognitive Dimensions Questionnaire and Some Thoughts About the Cognitive Dimensions Spreadsheet Calculation. In *Proceedings of the 13th Annual Workshop of the Psychology of Programming Interest Group (PPIG-13)*, pp. 291–301. 2001.

[VER05] VERSTANT. A Leader in Object-Relational Mapping and Object Databases, 2005. Web site at `http://www.versant.com` [accessed May, 2005].

[Vog99] W. Vogels. File System Usage in Windows NT 4.0. In *Proceedings of the Symposium on Operating Systems Principles (SOSP '99)*, pp. 93–109. ACM Press, New York, NY, USA, 1999.

[WD92] S. J. White and D. J. DeWitt. A Performance Study of Alternative Object Faulting and Pointer Swizzling Strategies. In *Proceedings of the 18th International Conference on Very Large Data Bases (VLDB '92)*, pp. 419–431. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[Wed91] G. Weddell. The Technology of Object-Oriented Databases. Technical Report CS-91-46, University of Waterloo, September 1991.

[WK92] P. R. Wilson and S. V. Kakkad. Pointer Swizzling at Page Fault Time: Efficiently and Compatibly Supporting Huge Address Spaces on Standard Hardware. In *Proceedings of the 1992 International Workshop on Object Orientation and Operating Systems*, pp. 364–377. IEEE Computer Society, Washington, DC, USA, 1992.

[WL89] S. P. Weiser and F. H. Lochovsky. OZ+: an Object-Oriented Database System. In *Object-Oriented Concepts, Databases, and Applications*, pp. 309–337. ACM Press, New York, NY, USA, 1989.

[ZCC95] M. Zand, V. Collins, and D. Caviness. A Survey of Current Object-Oriented Databases. *ACM SIGMIS Database*, 26(1):14–29, 1995.

[ZL77] J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transaction on Information Theory*, IT-23(3):337–343, May 1977.