Copyright

by

Jeffrey Alan Thomas

1998

P2: A Lightweight DBMS Generator

by

Jeffrey Alan Thomas, B.S., M.Eng., M.S.C.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 1998

P2: A Lightweight DBMS Generator

Approved by
Dissertation Committee:

____

____

____

____

____

For my Mother and Father

Acknowledgments

I am deeply indebted to my advisor, Don Batory, for many years of encouragement and advice, for providing constant direction and focus to my research, and for reading countless drafts of my dissertation. I am a much better researcher because of his excellent guidance.

I am also thankful to my committee members, Professors J. C. Browne, Don Fussell, and Dan Miranker, and, especially, to Dr. Marie-Anne Neimat for their meticulous reading of my dissertation and probing questions.

This research was supported in parts by a National Defense Science and Engineering Graduate Fellowship, an Intel Foundation Graduate Fellowship, Digital Equipment Corporation, and the University of Texas Applied Research Laboratories.

JEFFREY ALAN THOMAS

The University of Texas at Austin
November 1998

P2: A Lightweight DBMS Generator

Publication No. _______________

Jeffrey Alan Thomas, Ph.D.
The University of Texas at Austin, 1998

Supervisor: Don S. Batory

Generators are tools that automate well-understood software development tasks. Unfortunately, generators are generally difficult to build and/or use, and often produce code with poor run-time performance. *GenVoca generators* are a special class of generators that provide a domain-independent blueprint for constructing customized systems from pre-fabricated software building blocks called components. Previously, GenVoca generators have failed to simultaneously provide high run-time performance and programmer productivity while scaling to produce complex systems.

Database management system (*DBMS*) construction is well-understood, but has refused to be automated. Much data management functionality is hand-coded, because appropriate DBMSs are not available, difficult to use, or offer inadequate run-time performance. A *lightweight database management system* (*LWDB*) is a high-performance, application-specific DBMS that omits one or more features and specializes the implementation of its features to maximize performance. LWDBs offer the potential to extend DBMS sup-

port to much of the data management functionality that is still hand-coded. GenVoca generators offer a powerful means of constructing LWDBs.

In this dissertation, we describe P2, a prototype GenVoca generator of LWDBs. Given an application developed using an implementation-independent, domain-specific programing language; together with a feature specification written using a concise, high-level notation; P2 generates a high-performance LWDB specialized to the requirements of the application. Our target applications are LWDBs for the LEAPS production system compilers and the Smallbase high-performance, main-memory DBMS. We demonstrate how P2 shows that GenVoca generators can scale to produce complex systems while maintaining high generated code performance and programmer productivity.

Contents

Acknowledgments **v**

Contents **viii**

Chapter 1 Introduction **1**
1.1 Overview . . . 1
1.2 Outline . . . 3

Chapter 2 Literature Review **6**
2.1 Effective Software Construction . . . 6
2.1.1 High-level Languages . . . 7
2.1.2 Parameterized Programming . . . 7
2.1.3 Object-oriented Programming and Frameworks . . . 8
2.1.4 Software Reuse . . . 9
2.1.5 The Feature Combinatorics Problem . . . 10
2.1.6 Glue . . . 12
2.1.7 Domain Analysis . . . 14
2.1.8 Transformation Systems . . . 15
2.1.9 Subjectivity . . . 17
2.1.10 Generators and Hybrid Kits . . . 18
2.2 GenVoca . . . 19
2.3 Lightweight DBMSs . . . 24

Chapter 3 P2 **28**
3.1 Application Development . . . 29
3.1.1 Abstractions . . . 29
3.1.2 Operations . . . 34
3.2 Feature Specification . . . 35
3.2.1 Components . . . 37
3.2.2 Implementation . . . 39
3.2.3 Transformation . . . 41

Chapter 4 LEAPS **46**
4.1 The LINK Realm . . . 46
4.2 The LEAPS Algorithms . . . 51
4.2.1 literalize Statements and OPS5 Terminology . . . 52
4.2.2 LEAPS Overview . . . 54
4.2.3 Rule Translation . . . 56

4.2.4 Other Issues 61
4.2.5 Notes on Data Structures 62
4.2.6 Optimizations 64
4.3 Results 68

Chapter 5 Smallbase **74**
5.1 Decomposition 75
5.2 The PROCESS Realm 78
5.2.1 Process Manager 80
5.3 The XACT Realm 82
5.3.1 Lock Manager 82
5.3.2 Lock Protocol Manager 86
5.3.3 Transaction Manager 88
5.3.4 Log Manager 89
5.3.5 Operation Vector Manager 98
5.4 Methodology 99
5.4.1 Standard TPC-B benchmark 99
5.4.2 Modified TPC-B Benchmark 102
5.5 Results 105
5.6 Conclusions 110

Chapter 6 Retrospective **112**
6.1 P2 112
6.1.1 ddl 113
6.1.2 Design Rule Checking 115
6.1.3 xp 117
6.1.4 pb 121
6.2 GenVoca 125
6.2.1 Realms 125
6.2.2 Components 128
6.2.3 Generators 129
6.3 Lightweight DBMSs 130
6.3.1 Performance Improvement Limits 131
6.3.2 Software Engineering Challenges 132
6.3.3 GenVoca and P2 Solutions 133

Chapter 7 Conclusions **136**
7.1 Results 136
7.2 Contributions 137
7.3 Future Research 138

Appendix The P2 Log Manager **140**

Bibliography **165**

Vita **183**

Chapter 1

Introduction

1.1 Overview

This dissertation shows that GenVoca generators can scale to produce complex systems while maintaining performance and productivity. We developed P2 as a prototype to demonstrate the scalability of the GenVoca model of software generation. P2 generates high-performance, application-specific database management systems (DBMSs).

Generators (e.g., LEX [Les79] and YACC [Sch79]) are tools that automate well-understood software development tasks. Generators are an old idea; many have been constructed and used to great advantage [Kru92]. Yet, in many domains, programming is still done by hand, because of the unavailability of suitable generators. The problems are that generators are generally difficult to build and/or use, and often produce code with poor run-time performance.

GenVoca generators (e.g., Genesis and Avoca) [Bat92a, Bat94c] are a special class of generators that combine aspects of domain-specific languages, parameterized programming, reuse, software architectures, transformation systems, and subjectivity. The GenVoca method of generator construction uses a domain-independent blueprint for constructing customized systems from pre-fabricated software building blocks (a.k.a., components). Generators developed before P2 have failed to simultaneously provide high run-time performance and programmer productivity while scaling to produce complex systems. GenVoca offers great promise to extend generator technology to the many well-understood domains in which programming has not yet been successful automated.

The construction of DBMSs is a domain that is particularly well-understood, but has stubbornly refused to be automated. DBMSs abound in many shapes (e.g., main-memory-optimized, temporal, distributed) and sizes (e.g., heavyweight client-server systems, lightweight persistent object stores, and libraries). But much data management functionality is still hand-coded (e.g., without using any DBMS, by modifying a DBMS, or as middleware on top of a DBMS), because appropriate DBMSs are not available, difficult to use, or offer inadequate run-time performance. The LEAPS production system compilers [Mir90-91] are examples of systems with data management needs that have not (previously) been adequately supported performance-wise by conventional DBMSs.

A *lightweight database management system* (LWDB) (e.g., Smallbase [Hew96]) is a high-performance, application-specific DBMS [Tho95, Bat97b]. It differs from a general-purpose (heavyweight) system in that it omits one or more features and specializes the implementation of its features to maximize performance. LWDBs offer the potential to extend DBMS support to much of the data management functionality that is still hand-coded; GenVoca generators offer a powerful means of constructing LWDBs.

P2 is a prototype generator of LWDBs. P2 users follow a two-phase approach to the development of LWDBs and their applications. The first phase is application development using a implementation-independent, domain-specific programming language; the second phase is feature specification using a concise, high-level notation. As input, P2 takes the product of these two phases. As output, P2 generates a high-performance DBMS specialized to the requirements of the application. The separation of application development from feature specification results in substantially higher programmer productivity, and GenVoca generation does not compromise generated code performance.

In building the P2 prototype, the intellectual challenge was to scale GenVoca generators to generate systems the complexity of the LEAPS and Smallbase LWDBs while maintaining generated code performance and application programmer productivity. Previous generators have failed to meet this challenge. For example, Genesis did not offer adequate performance; P1 did not scale. GenVoca, however, was not formalized until after these systems were constructed. By following the GenVoca blueprint from the ground up,

developing the concept of a component definition language (xp, see Chapter 3), and solving some domain modeling problems (PROCESS and XACT, see Chapter 5) along the way, P2 has risen to this challenge.

1.2 Outline

This dissertation explores the problems of building LWDBs, explains our proposed solution, describes its embodiment in P2, and analyzes our experimental results.

In Chapter 2 we put GenVoca in context with other technical approaches to software engineering, and P2 in context with other LWDBs. First, we describe the following approaches to software engineering: high-level languages, parameterized programming, object-oriented programming and frameworks, software reuse, glue, domain analysis, transformation systems, subjectivity, and generators and hybrid kits. Next, we explain how GenVoca combines aspects of these approaches. GenVoca is a domain-independent model and concise notation for defining families of hierarchical systems called realms as reusable component compositions specified as type expressions and annotations. Finally, we examine lightweight systems as a natural evolution from heavyweight extensible systems.

In Chapter 3 we describe the P2 LWDB and its fundamental realms, top, data structure, and memory (a.k.a., TOP, DS, and MEM). We present a two phase approach to developing LWDBs and their applications. The first phase is application development. We explain how in this phase, applications are coded using the data types and operations that P2 adds to ANSI C. We describe the **element**, **cursor**, **container**, and **schema** data types; operations added by; and components implementing the TOP, DS, and MEM realms. The second phase is LWDB feature specification. We explain how in this phase, features of the LWDB are declared using type expressions and annotations which describe how the P2 generator will implement these data types and operations.

In Chapter 4 we present the link (a.k.a., LINK) realm, the LEAPS algorithms, and an experiment reengineering LEAPS to test P2's productivity and performance relative to hand-coding. First, we describe the composite cursor (a.k.a., **compcurs**) data type, opera-

tions added by, and components implementing the LINK realm. Next, we explain the LEAPS algorithms, and how our RL translator reengineers them using these data types, operations, and components. Finally, we compare the productivity and performance of RL versus the LEAPS compilers, OPS5.c and DATEX. We found that P2 increased programmer productivity by a factor of three versus hand-coding. We also found that the run-time performance of the code generated by RL was substantially better than that generated by the LEAPS compilers. When using the standard nested loops join algorithms, the code generated by RL was typically two times faster than OPS5.c and fifty times faster than DATEX. When we optimized RL (but not the LEAPS compilers) to use a hashed implementation of the nested loops join algorithm, the code generated by RL was up to two orders of magnitude faster than OPS5.c and three orders of magnitude faster than DATEX.

In Chapter 5 we present the Smallbase DBMS, the process and transaction (a.k.a., PROCESS and XACT) realms, and an experiment reengineering Smallbase to test P2's scalability. First, we describe the Smallbase monolithic, lightweight, main-memory-optimized DBMS and how we reengineered it by decomposing it into components. Then, we describe the **process**, **semaphore**, and **condition** data types; operations added by; and component implementing the PROCESS realm. We also describe the transaction (a.k.a., **xact**), **lock**, and log sequence number (a.k.a., **lsn**) data types; operations added by; and components implementing the XACT realm. Next, we describe the standard TPC-B benchmark, and a version of it modified for main-memory-optimized DBMSs. Finally, we compare P2 and Smallbase using the modified version of the TPC-B benchmark. We found that the P2 generated code performance and application programmer productivity was at least as good as that of Smallbase. Thus, we conclude that we were able to scale P2 to generate systems the complexity of Smallbase while maintaining performance and productivity. In addition, the flexibility of GenVoca generation allows P2 to generate more compact code and offer algorithmic optimizations that Smallbase cannot.

In Chapter 6 we analyze the lessons that P2 has taught us. We begin with specific (low-level) details about the P2 tool itself. We continue with general issues concerning GenVoca Generators. Finally, we consider (high-level) issues concerning LWDBs.

In Chapter 7 we review the central results of our experimental work, summarize the primary contributions of our research, and discuss a few areas of future research and enhancement to P2.

Chapter 2

Literature Review

It is difficult to construct software, particularly large systems with high run-time performance. This difficulty is manifest in low programmer productivity, high initial and maintenance costs, long time to market, and large proportions of defects (a.k.a., *bugs*). These problems have been known since (at least) 1968, when the NATO Software Engineering Conference identified them, gave the name *software crisis* to them, and launched the field of software engineering to solve them [Nau68]. They still have not been adequately solved.

2.1 Effective Software Construction

Software engineers use many interrelated approaches to reduce the difficulty of software construction. These approaches consider both technical and non-technical issues. While the non-technical issues such as management, culture, and politics are (doubtless) quite important, they are not the focus of our research. We limit our discussion here to technical issues that many researchers have recognized to be important. For simplicity we impose the following partition: high-level languages, parameterized programming, object-oriented programming and frameworks, software reuse, glue, domain analysis, transformation systems, subjectivity, and generators and kits. We delay analysis of their similarities and differences with GenVoca until Section 2.2.

2.1.1 High-level Languages

High-level languages are perhaps the most effective approach to reduce the difficulty of software construction, resulting in at least a factor of five improvement in programmer productivity [Bro86]. Examples include Fortran [Met96], C [Ker88], and C++ [Ell90]. Although they do not allow programmers to perform some *low*-level machine-specific optimizations, high-level languages have *not* been found to significantly reduce run-time performance. This is due to the fact that, they make it significantly easier to construct software, and thus allow programmers to concentrate on *high*-level domain-specific optimizations, rather than low-level details. Thus, high-level languages can actually *increase* run-time performance.

Two interesting variants of high-level languages are so called very high level languages and domain-specific languages. *Very high level languages* emphasize formality and elegance of expression [Kru92]; examples include SETL [Kru84] and SML [Pau96]. *Domain-specific languages* [USE97] (a.k.a., *narrow spectrum languages* [Nei89]) are specialized to a particular *family* [Par79] (a.k.a., class or product line) of related systems; they are typically implemented using either transformation systems (see Section 2.1.8) or generators (see Section 2.1.10). Neither very high level languages nor domain-specific languages need be restricted to text-based languages, but may include graphical diagrams and menu-driven dialogs (e.g. [Nov93, Bro95b]).

2.1.2 Parameterized Programming

Parameterized programming (a.k.a., *software schemas*) allows generic software to be written once, and instantiated many times for different uses. Goguen identifies two types of parameterization: horizontal and vertical [Gog86]. *Horizontal parameterization* is used to factor out common design elements (e.g., constant values or data types). *Vertical parameterization* is used to layer progressively higher programming abstractions (i.e., abstract machines) in order to progressively implement functionality. Goguen's library interconnection language, LIL, simultaneously provides both horizontal and vertical

parameterization. This provides a powerful model of software, allowing maximum reuse of existing software artifacts, and greatly increasing productivity. Other important parameterized programming systems include GLISP [Nov83], LILEANNA [Tra93], PARIS [Kat89], and RESOLVE [Sit94].

2.1.3 Object-oriented Programming and Frameworks

Object-oriented programming combines the notions of objects and inheritance. *Objects* encapsulate data and operations (called *methods*). *Inheritance* hierarchies allow new object interfaces called *classes* and implementations to be derived from existing ones. Programming language support for object-oriented programming originated with SIMULA [Bea73]. The most important object-oriented programming languages are probably Smalltalk [Gol89], C++ [Ell90], and Java [Gos96]. Others include BETA [Mad93], Eiffel [Mey91, Mey97], and Self [Ung87].

An *object-oriented framework* consists of a suite of interrelated abstract classes that embodies an *abstract* design for software in a family of related systems [Joh88]. Each major component of the system is represented by an abstract class. Thus, frameworks have the advantage of allowing reuse at a granularity larger than a single abstract class. But frameworks have the disadvantage that users may have to manually specify system-specific functionality.

In a *white-box framework*, users specify system-specific functionality by adding *methods* to the framework's classes. Each method must adhere to the *internal* conventions of the classes. Thus, using white-box frameworks is difficult, because it requires knowledge of their implementation details. In a *black-box framework*, the system-specific functionality is provided by a set of *classes*. These classes need adhere only to the proper *external* interface. Thus, using black-box frameworks is easier, because it does not require knowledge of their implementation details. Using black-box frameworks is further simplified when they include a library of pre-written classes that can be used as-is with the framework.

The main advantages of object-oriented languages are that they promote encapsulation and reuse [Lew91, Big92]. Encapsulation tends to lead to clean, compact, and elegant designs. Classes and frameworks tend to encourage larger and more abstract reusable components than functions.

2.1.4 Software Reuse

Software reuse is the process of creating new systems from existing *artifacts* (a.k.a., *assets*) rather than building new systems from scratch [Kru92, Pri93]. Reuse has obvious and significant appeal. It is much easier to reuse existing artifacts than build new ones from scratch [Sel88]. The most obvious example of artifacts that can be reused are source code fragments. But reusable artifacts may be drawn from the full life cycle: requirements, analysis, specifications, designs, documentation, and object code. Potentially reusable design artifacts include specifications written in a design modeling language such as the Unified Modeling Language (UML) [Rat98], design patterns [Gam95], and transformations (see Section 2.1.8).

The most naive approach to reuse is scavenging. *Code scavenging* (a.k.a., *leverage*, *cloning*, [Gri94] *copying*, or *cut-and-paste*) is an ad hoc technique by which software engineers accumulate or locate source code of existing systems *not* specifically designed to be reused (called *legacy systems* if they are *still* in use), find relevant fragments in these systems, and either (1) use them *as-is* (a.k.a., *black-box reuse*) or (2) manually adapt them for use in new systems (a.k.a., *white-box reuse*).[1] Unfortunately, finding relevant source code fragments may require considerable searching, and modifying existing systems for reuse requires understanding them, which itself may require more effort than writing the code from scratch. Thus, although credible, the benefits of scavenging are modest [Big87].

---

1. The term *design scavenging* is sometimes used to describe scavenging in which a large block of source code is used, but many of the internal details are deleted, while the global template of the design is retained [Kru92]. We avoid using this term, which we consider misleading, because the artifact that is being scavenged is still source *code*, rather than a *design*.

A more successful approach to reuse is based on libraries. A *library* (a.k.a., *repository* or *knowledge base* [Nei94]) is a collection of artifacts called *components* specifically designed to be reused. In 1968, McIlroy [McI68] originally envisioned that the components in the library would be functions (a.k.a., subroutines or procedures), because functions were the only suitable language feature available at that time. Since then, however, libraries have been so successful that high level languages have evolved features specifically designed to support components: modules, packages, subsystems, and classes [Kru92].

Reuse can greatly simplify software construction–it has the potential to provide an order of magnitude increase in programmer productivity. Unfortunately, it has three major disadvantages:

- *Difficulty of construction*. It is more difficult to build an object if it is intended to be reused than if it isn't. In general, it is 2 to 3 times more difficult [Bro95a]. But the payoff of building for reuse can be substantial.
- *Limited domain of applicability.* The *domain* may be the most important factor in reuse success. The domain must be narrow, well-understood, and slowly changing. Biggerstaff estimates that these properties of the domain account for 80% of the success of software reuse [Big92].
- *The feature combinatorics problem*. This is the subject of the next section.

2.1.5 The Feature Combinatorics Problem

Large components result in a higher gain in programmer productivity (a.k.a., *payoff*) than small components, because when reusing a large component, a programmer typically has to write fewer lines of code than when reusing a small component [Big94]. In fact, as components grow, the payoff involved in reusing them increases *more* than linearly, because of the additional costs introduced as the complexity of the objects grow [Big87]. In concrete terms, building a system out of x components each with size 10 is more than 10 times cheaper than building a system out of 10x components of size 1. Thus, large-scale reuse is *much* better for the reuser than small-scale reuse.

Component size (a.k.a., *scale*) is sometimes characterized in terms of features. A *feature* is a granule of functionality. In McIlroy's sine function domain, features would be precision, floating-versus-fixed computation, argument ranges, and robustness [McI68]. *Vertical scaling* (a.k.a., *encapsulation*) refers to adding more features to the components in a library; *horizontal scaling* (a.k.a., *aggregation*) refers to adding more variations of these features to a library [Big94]. We would like to simultaneously increase both the vertical and horizontal scale of libraries.

Brute force is one way to increase horizontal scale. That is, library implementors may attempt to populate their library with enough components to provide all possible feature choices. Unfortunately, the number of such components increases *exponentially* in the number of feature choices. In concrete terms, if there are at least 2 variations for each of x features, at least 2^x components are necessary to fully populate the library. In general, there may be more than 2 variations for each feature, so the problem is even worse. For example, McIlroy envisioned 10 precisions, 2 scalings, 5 ranges, and 3 robustnesses for a total of $(10)(2)(5)(3) = 300 > 2^4 = 16$ variations of the sine routine [McI68]. This is known as the *feature combinatorics*, *combinatorial explosion*, or *library scalability* problem [Tho93, Bat93].

The feature combinatorics problem has several solutions: standards, parameterization, and factoring. Unfortunately, all of these solutions tend to result in reduced run-time performance.

- *Standards* are commonly agreed upon feature choices, which limit feature variation, and enhance compatibility. Standards have mitigated the feature combinatorics problem in many domains. For example, in the mathematics domain of McIlroy's sine function, by adhering to number representation and other mathematics standards, programmers can use sine and other functions and variables without having to worry about precision, scaling, range, or robustness compatibility. Standards may result in reduced run-time performance when they specify more generality (and thus more computational expense) than an system requires. For example, in the mathe-

matics domain the standards may specify higher (and thus computationally more expensive) degree of precision, scaling, ranges, or robustness than an application requires.

- The crux of the feature combinatorics problem is that components are excessively concrete [Big94]. That is, each component embodies a particular set of feature choices. *Parameterization* is good method to decrease component concreteness. Instead of concretely embedding features in components, we can replace concrete feature choices with parameters that may be instantiated as needed. The C++ Standard Template Library [Mus96] is an example of a highly parameterized library. Parameterization can result in reduced run-time performance when vestiges of the generality it introduces cannot be entirely eliminated at compile-time. For example, when parameterization is implemented using virtual functions [Ell90], the virtual dispatch overhead results in reduced run-time performance.
- *Factoring* is a good method to decrease component size. In a fully factored library, each component embodies exactly one feature choice [Big94]. A large scale component is built from these factored components (a.k.a., *layers of abstraction*) by composing exactly the components needed to provide the chosen features. Factoring can result in reduced run-time performance when vestiges of the layering remain at run-time. For example, when factored components are implemented as functions, the layers of abstraction remain at run-time as deep call chains, and the function call overhead results in reduced run-time performance.

2.1.6 Glue

Some researchers see the main challenge to reuse as being the interconnection of components [Tra95]. Components do not always fit together just right (by analogy to physics, this phenomenon is often called *impedance* mismatch between components). Thus, the software engineer must supply some sort of *glue* (or *connectors*) to make them fit. Research into this glue includes both technological and engineering approaches, under

various names including wrappers, module [DeR76, Pri86] (a.k.a., library [Gog86]) interconnection languages, and software buses. The primary disadvantage of glue is that it tends to reduce run-time performance.

Wrappers (called *method wrappers* in object-oriented programming) are used to encapsulate and manage interactions with existing artifacts. Wrappers can change what is executed prior to (*before method*) or subsequent to (*after method*) the original method. In object-oriented systems, such as SOM [For94] and CLOS [Kic91], wrappers are implemented as class objects called *metaclasses*. Sometimes, wrappers can be used to modify an existing method without modifying its source code or recompiling it. Sometimes, wrappers are used to give an object-oriented interface to a non-object-oriented legacy system [Jac97]. Other applications of wrappers include function tracing, invariant checking, and object locking [For94].

Module interconnection languages extend high level languages with facilities designed to integrate independently developed components (called *modules*) into a complete system [Pri86]. Modules separate interface from implementation; they explicitly *export* the functions they implement, and *import* the functions they (re)use. Modules are assembled into a complete system by interconnecting modules with matching exports and imports [Kru92]. Module interconnection languages provide syntax to formally specify exports and imports, and semantics to automatically verify that they match. Important module and library interconnection languages include MIL75 [DeR76] and LIL [Gog86]. But many other high level languages also support module interconnection language technology.

Software buses (a.k.a., *middleware)* emphasize support for heterogeneous, distributed processing. They are designed to allow software engineers to reuse objects written in different programming languages or running in different processes. The most important software buses are probably CORBA [OMG98], OLE/COM/DCOM/OCX/ActiveX [Cha96, Mic98], and Java Beans [Sun98].

2.1.7 Domain Analysis

Domain analysis (a.k.a., *domain modeling*) is the process of identifying and organizing knowledge about a family of related systems–the *domain*–to support the description and implementation of such systems [Nei81, Nei84, Ara91]. That is, domain analysis is about finding commonalities among the systems; it is a learning process intended to determine a domain model. A *domain model* (a.k.a., *domain design*) is a definition of the operations, data objects, properties, abstractions, requirements, relationships, and variations appropriate for designing software in the domain. That is, a domain model is a representation of the essential aspects of the domain [Pri95]. A *domain architecture* (a.k.a., *domain specific software architecture* (*DSSA*) [SEI90, Tra94], *application framework* [Joh88], *reference architecture* [Bat95], or *software architecture*) is a domain model that embodies an abstract design for software in a family of related systems. Thus, a domain architecture is analogous to an object-oriented framework, but without the object-oriented implementation detail. *Domain engineering* includes domain analysis and modeling, as well as the development of reusable artifacts in the domain, and tools to assist in these tasks.

Domain analysis is a metalevel version of *requirements analysis*; domain analysis differs from requirements analysis in that its goal is the construction of a domain model, rather than a specification or design. Domain analysis is a specialization of *knowledge engineering*; domain analysis differs from knowledge engineering in that its goal is specifically to produce a domain model, rather then solving some more general problem. But, many of the techniques used in requirements analysis and knowledge engineering can be applied to domain analysis [Ara91].

Domain analysis must be performed before implementing many of the approaches to reduce the difficulty of software construction discussed in this chapter. Domain analysis is particularly important for object-oriented frameworks and reuse [Pri91]. Historically, this domain analysis has been performed informally–neither systematically nor explicitly. Sometimes domain analysis is the implicit result of years of *learning*. Sometimes domain analysis is embedded in *standards* arrived at by a community of software engineers [Ara91]. Researchers are unhappy with this informal state of affairs [Pri91]. An active

area of research is to make domain analysis systematic and explicit. These researchers hope that the body of knowledge concerning domain analysis can be usefully applied to improve the practice of software engineering.

2.1.8 Transformation Systems

Transformation systems are generalizations of conventional high level language compilers.[2] In a transformation system, software is constructed in two clearly separated phases, definition and transformation:

- *Definition*. In the definition phase, the programmer writes a specification that defines the desired semantics (i.e., behavior). The definition phase is exactly analogous to writing a program in a high level language, except that the high-level language is usually domain specific.
- *Transformation*. In the transformation (a.k.a., *synthesis*) phase, the system applies correctness-preserving transforms, with the goal of enhancing program efficiency without changing its behavior, until the specification has been transformed to executable code.[3]

General-purpose transformation systems and models of transformation systems include Draco [Nei84, Nei89], Draco-PUC [Lei94], IP [Sim95], Kids [Smi85, Smi91] and REFINE [Rea92], POPART/Paddle [Wil93], TAMPR [Boy89], and TXL [Cor95]. Domain-specific transformation systems include AP5 [Coh93] in the domain of data structures and Sinapse [Kan93] in the domain of mathematical-modeling.

---

2. Conventional high-level language compilers are analogous to transformation systems with fixed specification languages, predefined transformation libraries, and fully automatic choice of transformations [Bax97a].
3. Many researchers take a more expansive view of transformation systems, admitting non-correctness-preserving transforms and/or seeing the goal of the transformation phase being more generally to navigate the space of possible implementations, rather than merely generate code [Fea87]. In this view, the transformation of specification into code is only one possible goal. Other possible goals include the transformation of code into code (e.g., porting, restructuring, maintenance), and code into specification (i.e., *comprehension* or *reverse engineering*). We omit discussion of these other goals, however, since they are beyond the scope of our present research.

A *transform* (a.k.a., *enzyme* [Sim95], *refinement,* or *optimization*) is a partial function from specifications to specifications and/or code [Bax95]. Transforms are often represented as rewrite rules with pattern variables. A *rewrite rule* has a match pattern, applicability conditions, and a substitution pattern. The *match pattern* (a.k.a., *input schema*) is a template that defines what to search for in the initial program. The *substitution pattern* (a.k.a., *output schema*) defines the code that replaces the constructs in the transformed program. The *applicability conditions* define the domain of a transform by giving semantic constraints on when the transformation should be applied [Par90, Kru92]. Transformations include, for example, decomposition, generalization/specialization, algorithm and data structure selection, interleaving, delocalization, resource sharing, data caching/memoization, [Bax97b] as well as standard compiler optimizations such as fusion, loop combination, peephole optimization, register allocation, tupling, and unfold/fold.

The goal of the transformation phase is to produce an executable system that satisfies the high-level specification. The sequence of transformations applied during this phase is called a *history.* This phase is analogous to the code generation phase of a conventional compiler, except that the control regime is often not fully automatic and requires human guidance. The artificial intelligence (AI) field of *automatic programming* [Ric88] attempts to eliminate the need for human guidance, but domain-independent algorithm and data structure synthesis is generally beyond current compiler technology [Kru92].[4]

Rather than transforming the specification directly into the desired target code, many systems use an *intermediate* (a.k.a, *internal*) *format* (a.k.a., *abstract syntax*). The process of translating the specification into an intermediate format is called *abstraction* [Wat88]. The use of intermediate formats is quite common, for example, in high level language compilers. Typically, these formats are three address codes or abstract syntax trees

4. The AI fields of *knowledge based software engineering* (KBSE) and *automated software engineering* (ASE) do *not* attempt to eliminate the need for human guidance. KBSE and ASE are distinguished from transformation systems and the rest of the software engineering community in general primarily by their emphasis on classical AI techniques, such as automated reasoning and knowledge representation.

(ASTs) or directed acyclic graphs (DAGs). The Stanford University Intermediate Format (SUIF) [Hal96, SUI98] and Register Transfer Language (RTL) [Dav89] used by GNU's gcc compiler are two such formats. The AST internal format of IP is interesting in that it is designed to capture the programmer's "intentions" in a language independent fashion [Sim95]. Some transformation systems generalize from the idea of a single intermediate format to a hierarchy of several intermediate formats called *intermediate programming models* [Boy89] or *modeling domains* [Nei89].

The *transform control* (a.k.a., *navigation*) *problem* is how to choose what transforms to apply and why. Solutions to this problem include totally manual selection of transforms, repeated domain refinement/optimization, performance-directed search, metaprograms, and *replay* [Bal85] of previous transformation sequences. A *metaprogram* is a program that determines transform sequences; many systems emphasize reusable metaprograms [Che84]. A *design history* records previous transform sequences together with the methods used for selecting those transforms [Bax94, Bax95, Bax97a-b].

2.1.9 Subjectivity

Objects written for one application may not be reusable in another, because their interfaces are different, even though both application may deal with what is fundamentally the same object. The principle of *subjectivity* asserts that no single interface can adequately describe any object; objects are described by a family of related interfaces [Har93, Har94, Oss92, Oss95]. The appropriate interface for an object is application-dependent (or *subjective*).

Subjectivity arose from the need for simplifying programming abstractions–e.g., defining views that emphasize relevant aspects of objects and that hide irrelevant details [Shi89, Hai90, Gam94]. This lead to a connection of object modeling with view integration in databases [Elm89]–i.e., object models can be defined as a result of integrating different application (or sub-application) views of objects [Gol81, Har92]. Ossher and Harrison took an important step further by recognizing that application-specific views of inheritance hierarchies can be produced automatically by composing "building blocks" called extensions [Oss92]. An *extension* encapsulates a primitive aspect or "view" of a

hierarchy, whose implementation requires a set of additions (e.g., new data and method members) to one or more classes of the hierarchy. A customized "view" of an inheritance hierarchy could therefore be defined by composing extensions.

A rather different and powerful approach to views and software reuse has been proposed by Goguen [Gog86], Novak [Nov92, Nov97], and VanHilst [Van96]. The essence of this approach is to define generic abstract components that are automatically specialized to present a customized concrete implementation. A *view* is an isomorphism that defines a mapping of an object to a customized "perspective". A view *cluster* encapsulates a suite of interrelated views that maps multiple data objects simultaneously.

2.1.10 Generators and Hybrid Kits

The primary goal of generators and kits is to eliminate the mundane aspects of software construction and to permit the expenditure of proportionally more effort on the critical parts of target systems.

Domain-specific software *generators* are tools that automate well-known software development tasks. Generators are compilers for domain specific languages; they may be implemented as transformation systems. Examples of generators include fourth generation languages (4GLs), report generators, compiler-compilers such as YACC, graphical-user interface (GUI) builders, and language-based editors. By virtue of their domain-specificity, generators can automatically select algorithms and data structures from a fixed set of possibilities [Kru92]. Thus, they can result in an order of magnitude gain in productivity.

Domain-specific *kits* [Gri94] are a set of products designed to make it easy to build any of a family of related systems. All kits provide reusable components, a framework, and a glue language. Some kits may also provide design fragments, tests, templates, macros, documentation, tools, and examples. *Hybrid kits* combine kits with generators–in addition to the standard products provided by other kits, hybrid kits also provide a domain-specific language and a compiler for that language.

Like reuse, generators and kits can greatly simplify software construction–they have the potential to provide an order of magnitude increase in programmer productivity. Unfortunately, they have three major disadvantages:

- *Limited domain of applicability.* The domain of applicability (a.k.a., *domain width* or *domain coverage* [Cle88]) is the size of the family of related systems that a generator or kit can produce. The relatedness of these systems is the source of the strength, but also the fundamental weakness, of generators and kits–they greatly increase productivity in one domain, but one domain only.
- *Difficulty of construction*. Just as it is more difficult to build an object if it is intended to be reused that if it isn't, it is more difficult to build a generator or kit than to build a system by hand. But, instead of 2 to 3 times more difficult, it can be an order of magnitude more difficult to build a generator. Solutions to this difficulty including building the generator on top of a transformation system, using a specialized language such as P++ [Sin93a-b, Sin96], or using a generator kit such as Stage/MetaTool Specification-Driven Tool Builder [Cle88] or JTS [Bat98a].
- *Poor run-time performance relative to hand-coding*. Generators and kits may not be able to perform certain low-level optimizations or select optimal algorithm and data structures, and the resulting system may contain vestiges of generality (e.g., layering and interpretation) that add additional run-time overhead. This was clearly the case for Genesis. Part of our thesis is that generators can automatically build complex systems that have run-time performance that is comparable to or that exceeds that of hand-coded systems.

2.2 GenVoca

GenVoca is a domain-independent model and concise notation for defining families of hierarchical systems as compositions of reusable components. GenVoca takes its name from the first two GenVoca generators that were recognized as such: Genesis and Avoca, but our experiences with domain-specific software generators are not unique. Similar

experiences have been noted and virtually identical software organizations have been used in independently-conceived generators in many disparate domains: Avoca in network protocols [Oma92], Rosetta in data manipulation languages [Vil94, Vil97], Ficus in distributed file systems [Hei94], Brale in host-at-sea buoy systems [Wei90], and ADAGE in real-time avionics software [Bat95]. Thus it seems worthwhile to factor out the common, domain-independent ideas that underlie different software generators, and to build tools and develop design techniques that support these particular methods of software organization and construction. By doing so, we believe that other researchers in software engineering can benefit from these collective experiences without having to delve into the obscure details and vagaries of the particular domains from which they came:

- *Subsystems are the building blocks of generated systems.* Effective software synthesis requires that systems be constructed from combinations of *subsystems* (a.k.a., *components*) consisting of suites of interrelated functions and/or classes. It is too unwieldy to construct a large software by selecting and assembling hundreds or thousands of functions and classes from a reuse library (see Section 2.1.5). Thus, larger units of software encapsulation are needed.
- *Components import and export standardized interfaces.* The key to software synthesis is composition. Composition is much easier when component interfaces correspond to fundamental abstractions of the target domain and these interfaces have been standardized. Standardization encourages functionally similar components to be plug-compatible and interchangeable.
- *Component composition and customization is achieved through parameterization.* Parameterization is an easy-to-understand model for combining and customizing components. Simple forms of parameterization, i.e., constant and type parameters, are necessary but not sufficient for software generators. Components must also be able to import other components as parameters [Gog86, Tra93].

In GenVoca, the basic unit of software construction is the component. A *component* (a.k.a., *layer* or *subsystem*) encapsulates a suite of interrelated variables, functions, and classes that work together as a unit to implement a particular feature [Sin96]. Components

clearly separate interface from implementation. The interface of a component is anything that is visible externally of the component. Everything else belongs to its implementation. See Figure 2.1.

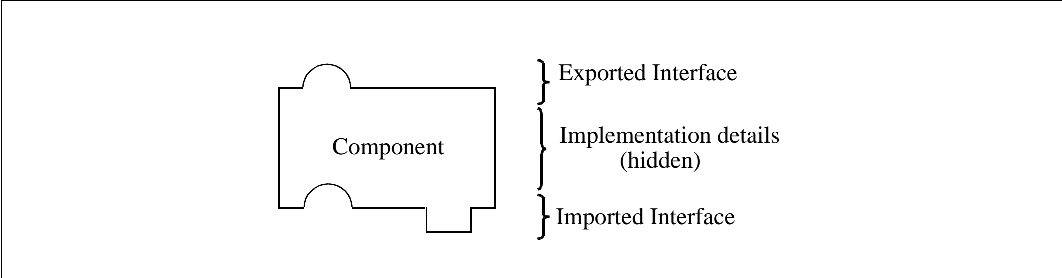

Figure 2.1 Anatomy of a component.

A *realm* is a set of components that implement a compatible interface in different ways. That is, all the components in a realm share what is to a first approximation the same interface, but have different implementations (see Section 2.1.9). Because their interfaces are compatible, all the members of a realm are plug-compatible and interchangeable. See Figure 2.2.

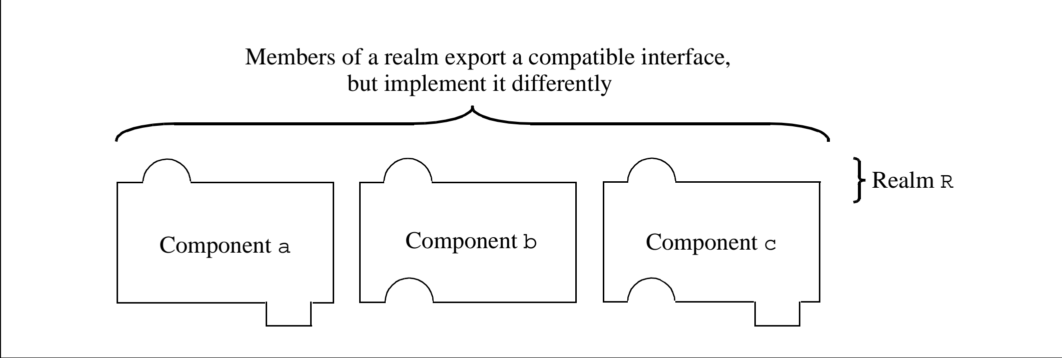

Figure 2.2 Members of a realm are interchangeable.

A GenVoca component encapsulates a set of data and function refinements, the purpose of which is to convert an abstract interface into a concrete implementation. Such refinements consistently refine multiple data types simultaneously. The application of

these refinements is controlled by certain domain-specific rules, which dictate under what circumstances the refinements are legal, whether optimizations of the refinements are possible, etc. The abstract interface of a component corresponds to a suite of function and data type declarations; the role of a component is the refinement of each function declaration to a concrete algorithm and each data type declaration to a concrete representation. When several components are combined to form a composite system, that system is a composite refinement corresponding to a sequence of individual component refinements.

To better explain this interpretation of components as refinements, GenVoca offers a concise notation for representing components, realms, and systems. If a component imports another component's interface, it is designated as a parameter. Thus, in the GenVoca notation, a component is denoted by its name, followed by a bracketed list of the names of the realms it imports, followed by a colon and the name of the realm it exports. For example, Figure 2.3 shows a component `c` that imports realm `S` and exports realm `R`.

```
c[S] : R
```

Figure 2.3 Example component c importing realm S and exporting realm R.

A realm is denoted as a set of elements, where each element represents a component belonging to the realm. For example, Figure 2.4 shows three realms: `R`, `S`, and `T`. Realm `R` has three components: `a`, `b`, and `c`; realm `S` has three: `d`, `e`, and `f`; and realm `T` has one: `g`. Component `b` imports realm `R` and component `c` imports realm `S`. Because it has two parameters, component `f` imports the two realm interfaces `S` and `R`. In essence, this notation treats a realm as if it were a type. A component from a realm is simply a function of some type, and a component that imports an interface has a parameter of some type. So, `d` is an object of type `S`, where `d` has a parameter of type `T`.

```
R = { a, b[R], c[S] }
S = { d[T], e, f[S, T] }
T = { g }
```

Figure 2.4 Example realms R, S, and T, and components a through g.

A *type expression* (a.k.a., *equation*) is a named composition of components that form a composite system. For example, Figure 2.5 shows type expression `A` that specifies how the components `c`, `d`, and `g` are combined to form a composite system.

```
A = c[d[g]]
```

Figure 2.5 Example type expression A that specifies a system built from components c, d, and g.

Note that component syntactic compatibility is easily checked by verifying that each parameter's imported interface matches the corresponding component's exported interface. Thus, `A` is syntactically valid, because the `c`'s imported and `d`'s exported interface are both realm `S`, and `d`'s imported and `g`'s exported interface are both realm `T`.

Component semantic compatibility is a more complicated issue. Note that some combinations of components may be syntactically but not semantically correct. That is, each pair of components in the system imports and exports compatible interfaces, but the resulting algorithms may be invalid for some reason. To verify the semantic correctness of a system, each component must supply domain-specific information that describes the assumptions and restrictions on the use of the component. See [Bat96] for details.

Consider the meaning of type expression `A` when components are viewed as refinements. The notation appears to suggest that GenVoca components are combined in a manner analogous to mathematical functions. This is, however, not the case–GenVoca components are relatively sophisticated, which makes a refinement model more appropriate for understanding component combinations than a model based on mathematical functions. When two components are interconnected, they exchange function, data type, and customization information with one another. The semantics of this exchange are more complicated than simple mathematical function composition. A functional interpretation of `A` would have an innermost-to-outermost evaluation semantics. That is, component `g` would be evaluated first, followed by `d`, then `c`. In GenVoca, the refinements of `A` start at the top component `c`, which provides data type information to component `d`; `d`, in turn, provides its own data types to `g`; which then supplies implemented data types and func-

tions back to d; and so on. Note that the refinements start at the top component, work their way down to the bottom component, and then back up to the top component.

In addition to imported and exported realm parameters, components often take additional imported parameters called annotations. *Annotations* (a.k.a., *nonrealm parameters* or *configuration parameters*) are instantiated by key field names, predicates, timestamp field names, file names, and other constants.

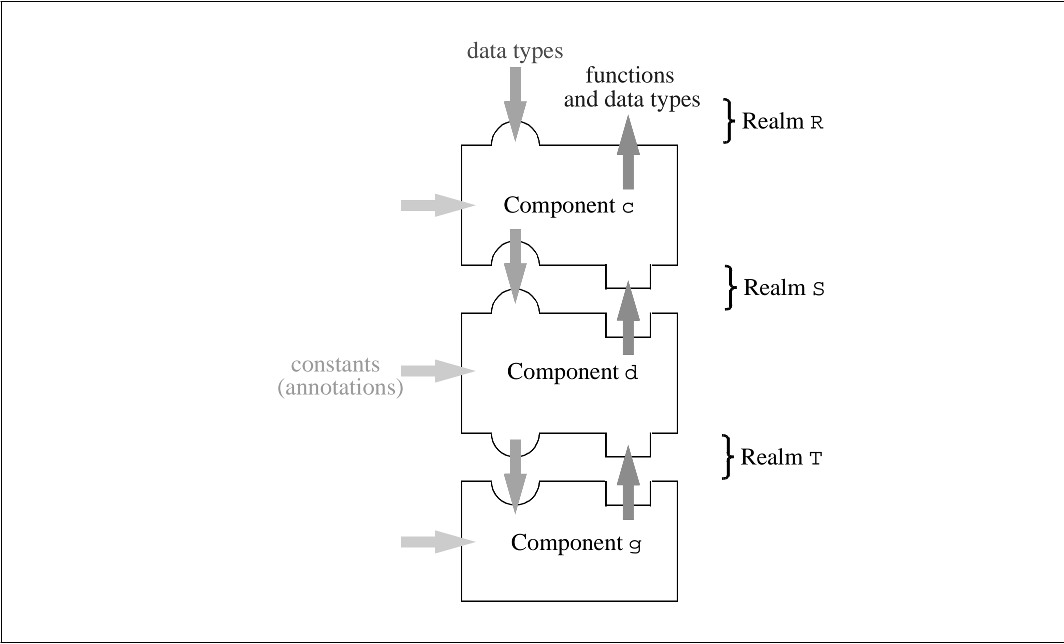

Figure 2.6 The system A built from the type expression A = c[d[g]].

2.3 Lightweight DBMSs

General-purpose DBMSs are *heavyweight*; they are feature-laden systems that are designed to support the data management needs of a broad class of applications. Among the common features of DBMSs are support for databases larger than main memory, client-server architectures, and checkpoints and recovery. A central theme in the history of DBMS development has been to add more features to enlarge the class of applications that can be addressed. As the number of supported features increased, there was sometimes a

concomitant (and possibly substantial) reduction in performance. A hand-written application that does not use a DBMS might access data in main memory in *tens* of machine cycles; a comparable data access through a DBMS may take tens of *thousands* of machine cycles. It is well-known that there are many applications (e. g., LEAPS [Mir90-91]) that, in principle, could use a database system, but are precluded from doing so by performance constraints.

Extensible or *open* database management systems were a major step toward DBMS customization. Early work on extensible DBMS systems includes TI's Open OODB [Wel92], IBM's Starburst [Haa90], Berkeley's Postgres/Miro/Illustra [Sto91-93], Wisconsin's Exodus [Car90], and Texas's Genesis [Bat88a-b]). These systems enable individual features or groups of features to be added or removed from a general-purpose DBMS to produce a database system that more closely matched the needs of target applications. Unfortunately, extensible DBMSs are basically customizable heavyweight DBMSs; their architecture and implementations (e. g., layered designs, interpretive executions of queries) still imposed the onerous overheads of heavyweight DBMSs. While feature customization of DBMSs *can* indeed improve performance, it has been our experience that the gains are rarely sufficient to satisfy the requirements of performance-critical applications.

Extensible DBMSs have grown significantly beyond these early efforts to become universal servers. A *universal server* is a heavyweight extensible DBMS that permits extensions to its type system to support arbitrary data types, such as document, time series, image, and spatial data [Ube94, Nor96]. These extensions (called *DataBlades* by Montage, Illustra, and Informix) are often packaged as modules that users can plug-in to servers at run-time. To support arbitrary data types, universal servers allow the addition of base and composite types, functions, functional indices, access methods, and storage managers. These extensions may permit very special-purpose, efficient algorithms, but universal servers themselves must be very general to support such extensions. For example, Data Blades require virtual dispatch of database functions, which adds overhead to all applications, even those that do not use the extensions [Nor96]. The generative techniques of P2

are general enough to permit similar extensions to the type system, but P2 makes the extensions at compile-time, rather than run-time, which allows P2 to compile away unnecessary overhead.

Lightweight database management systems (LWDBs) appear to be the next major evolutionary trend in DBMS design [Tho95, Bat97b]. A lightweight DBMS is an application-specific, high-performance DBMS that omits one or more features of a heavyweight DBMS *and* specializes the implementations of its features to maximize performance. Examples of LWDBs include main memory DBMSs (e. g., Smallbase [Hey95]), persistent stores (e. g., Texas [Sin92]), and primitive code libraries (e. g., Booch Components [Boo87], Code Farms C++ Data Object Library [Cod95]). Each of these examples strips features from a general-purpose DBMS (e. g., Smallbase removes the databases larger than main memory feature, Texas removes client-server architectures, and the Booch Components further strip checkpoints and recovery) and demonstrates the performance advantages gained by doing so. In principle, an application achieves its best performance when it uses a "lean and mean" LWDB that exactly matches its needs.

There are broad application classes that require lightweight, not heavyweight, DBMSs [Sou92, Bat92b]. Consider the lock manager of a DBMS. It maintains a set of tables whose structures resemble that of relations. Each table stores a different number of tuples (e.g., lock names and lock requests); operations on tables (e.g., set or remove locks) are atomic; and in the case of locks for long transactions, the tables are persistent and recoverable. Multi-table queries (e.g., retrieving identifiers of transactions that are unblocked by an unlock operation) arise frequently. Because performance is critical, the processing of queries is highly optimized. Clearly, a lock manager could use a general-purpose DBMS to manage its data, yet DBMSs are not used because of their inadequate performance and unnecessary generality. Operating systems provide another example. Page, segment, and process tables contain tuples (e.g., page-table entries, segment entries, and process control blocks) that are interrelated. Multi-table queries are common (e.g., return the identifiers of pages that were referenced by a given process in the last time interval) and operations in a multiprocessor environment must be atomic. Although these

tables are generally neither persistent nor recoverable, the basic features that are needed to maintain page, segment, and process data are offered by general-purpose DBMSs. Once again, performance precludes DBMS usage. There are many other similar examples: compilers query and update symbol tables, network managers access and modify routing tables, name servers maintain a horizontally partitioned and distributed table of service names and network addresses, mail systems query address/alias tables, and persistent object stores maintain page translation tables to map virtual memory page slots to pages on disk. Essentially any application fits this paradigm if it includes code for updating and querying data structures and does not use a conventional DBMS.

Because there are no formalizations, tools, or architectural support, LWDBs are hand-crafted monolithic systems that are expensive to build and tune. There is an opportunity for researchers to significantly improve this situation. The database community has solved very general problems of reliable and efficient data management. Clearly, what is needed are lightweight DBMSs that are extensible. The challenge is to focus these general solutions to very specific situations in order to create a technology for constructing high-performance lightweight systems quickly and cheaply.

In this dissertation we describe P2, an extensible lightweight DBMS. P2 provides architectural support and implementation techniques to assemble high-performance lightweight DBMSs from component libraries. P2 users code their applications in a database programming language that is a superset of C. P2 automatically builds (a.k.a., *generates*) a custom LWDB by analyzing the application code and by following user-specified directives that define the database features that are to be supported. P2 performs many optimizations at compile-time: it compiles queries, inlines code to manipulate indices, and partially evaluates code statically, thus enabling the performance of P2-generated LWDBs to be comparable or exceed that of hand-written LWDBs.

Chapter 3

P2

The conventional approach to LWDB construction is fraught with problems. With a partial understanding of the work loads that a LWDB is to support, LWDB designers invent data/storage structures and algorithms that match the perceived need. Implementing the design is tedious, expensive, and time-consuming, as it often involves adapting, coding, and debugging well-known algorithms. Once completed, the LWDB is integrated with the target application to see how well it performs. Without exception, the anticipated work load is different than the actual work load, and thus some of the design decisions/features of the hand-coded LWDB are recognized to be sub-optimal. At this point, designers face two unpleasant options: either leave the LWDB as is, knowing that its performance could be improved, or redesign and recode the LWDB for yet another round of testing. Redesigning has the further unpleasant side-effect that the interface to the LWDB may change, which in turn, would cause parts of the application that use the LWDB to be recoded.

There are two fundamental problems with this approach. First, LWDBs should not have ad hoc interfaces. A LWDB should provide a stable, well-designed interface that would permit applications to be insulated from changes in LWDB implementations. Second, there needs to be a way of reusing well-known algorithms, so that the rote tasks of adapting, coding, etc., can be largely avoided. These are the motivating objectives of P2. To accomplish them, P2 users follow a two-phase approach to the development of LWDBs and their applications.

The first phase of this approach is *application development*. P2 extends ANSI C with *special data types* (e.g., elements, containers, cursors, and schemas). LWDB applica-

tions are coded in terms of these data types without regard to how these types are implemented. This approach radically simplifies programming: application development using high-level database abstractions is substantially easier than using low-level, ad hoc interfaces of hand-crafted LWDB modules. In Section 3.1, we present the data model and embedded language of P2.

The second phase of this approach is LWDB *feature specification*, i.e., how the features of a LWDB are declared and how implementations of the P2 data types are to be generated. For P2, a lightweight database system for an application is the implementation of the P2 data types that it references. We will see in Section 3.2 that both feature specification and data type implementations are accomplished by composing components from the P2 library. An important advantage of this approach is that it is possible to radically alter the implementations of cursors, containers, etc., of an application (via a different combination of components) to improve application efficiency without modifying application code. Thus, tuning P2 LWDBs is considerably simplified.

3.1 Application Development

The P2 data modeling concepts are rather conventional: element, container, cursor, and schema. Our choice of these abstractions was deliberate. We wanted the P2 API to be as familiar and easy to learn as possible for database programmers.

3.1.1 Abstractions

An **`element`**[1] is P2's basic unit of retrieval, update, and insert. Thus, a P2 element is analogous to what other systems call a record, row, tuple, or object. An element is simply a C struct. The fields of the element can be of any number and type, but P2 acts specially on fields of certain recognized data types, currently limited to C integers and null-terminated character strings. In addition to the standard C strings of type array of character (`char[]`)

1. We use boldface for P2 data types.

and pointer to character (`char*`), P2 supports fields of type **`varchar`**. Fields of type **`var-char`** are declared in the same manner as fixed-length arrays, but P2 treats them as variable-length arrays. That is, fields of type **`varchar`** are declared with a length, but P2 ignores this length and treats the field as if it has indeterminate length. An element can have at most one field of type **`varchar`**, and it must be the last field in the element. Figure 3.1 shows an example declaration of `PERSON`, an element with fields of type integer, fixed-length string, and variable-length string.

```
// Element declaration.
typedef struct {
  int group;         // Integer.
  char name[10];     // Fixed-length character string.
  varchar misc[10]; // Variable-length character string.
} PERSON;
```

Figure 3.1 Example element declarations.

The int and character string data types have proven to be sufficient to handle a wide variety of applications. When they are not exactly appropriate, we can usually treat them as if they were. For example, in the log manager, we cast `int` to `unsigned`, and treat **`varchar`** as untyped binary data. In addition, we modularized as much as possible the data type specific portions of P2, so if the need arises in the future, it should be possible to add more data types (for example, `float`, `double`, `long`).

A **`container`** is a collection of elements that are all instances of a single type. Thus, a P2 container is analogous to what other systems call a relation, table, or file; the P2 element type is analogous to what other relational systems call a relational schema. Containers are parameterized by the type of element that is to be stored; several containers may have the same data type. Before an element is inserted into a container, it is merely an instance of a C struct. When an element is inserted into a container, it is copied by value, so it loses its object identity. That is, each insertion of the element into the same or a different container, yields a new element (even though the elements all have the same values in their user-visible fields). After an element is inserted into a container, it may only be referenced via a cursor. Figure 3.2 shows an example declaration of `faculty_container` and `student_container`, which are both containers of PERSON elements.

```
// Container declarations using type expression A.
container <PERSON> stored_as A with { } faculty_container;
container <PERSON> stored_as A with { } student_container;
```

Figure 3.2 Example container declarations using type expression A.

A **cursor** is a placemarker into a container that imposes a one-at-at-time ordering on the elements in a container. Thus, a P2 cursor is analogous to an embedded SQL cursor. Cursors are parameterized by the container to be traversed and optionally by a selection predicate and/or sort criterion. A container *may* be traversed by more than one cursor. Figure 3.3 shows an example declaration of `faculty_cursor`, a cursor over `faculty_container`, with a selection predicate and sort criterion; and `student_cursor`, a cursor over `student_container`. Note that predicates are encoded in P2 in quotes, as if they were strings. Within a predicate, the dollar sign (`$`) denotes the element referenced by the cursor, and string constants are enclosed in single quotes. Thus, in our example, `$.dept` denotes the `dept` field of the element referenced by the cursor, and `'Bob'` represents a string constant.

```
// Cursor declarations.

cursor <faculty_container> // Cursor traversing the faculty container.
  where "$.name != 'Bob'"  // Selection predicate.
  orderby group            // Sort criterion.
faculty_cursor;            // Cursor variable.

cursor <student_container> // Cursor traversing the student container.
student_cursor;            // Cursor variable.
```

Figure 3.3 Example cursor declarations.

A **schema** is a collection of containers. Thus, a P2 schema is analogous to what other systems call a database; the schema *type* is analogous to what other relational systems call a database schema. A P2 application may reference at most one schema. Since not every container need be an element of a schema, schemas may seem unnecessary. This is not the case. Schemas are *useful*, because by combining several containers into a single object, they increase encapsulation; schemas are *necessary* for joins (as we will see in Chapter 4) and transactions (as we will see in Chapter 5). Figure 3.4 shows an example declaration of `s`, a schema with container members named `faculty_container` and

student_container. Note that the schema container is analogous to the containers named faculty_container and student_container declared in Figure 3.2, except that the schema containers are named s.faculty_container and s.student_container, since they are members of the schema s.

```
// Schema declaration.
schema {
  container <PERSON> faculty_container;
  container <PERSON> student_container;
} stored_as A with { } s;
```

Figure 3.4 Example schema declaration using type expression A.

The relationship between the elements, containers, cursors, and schema is depicted in Figure 3.5.

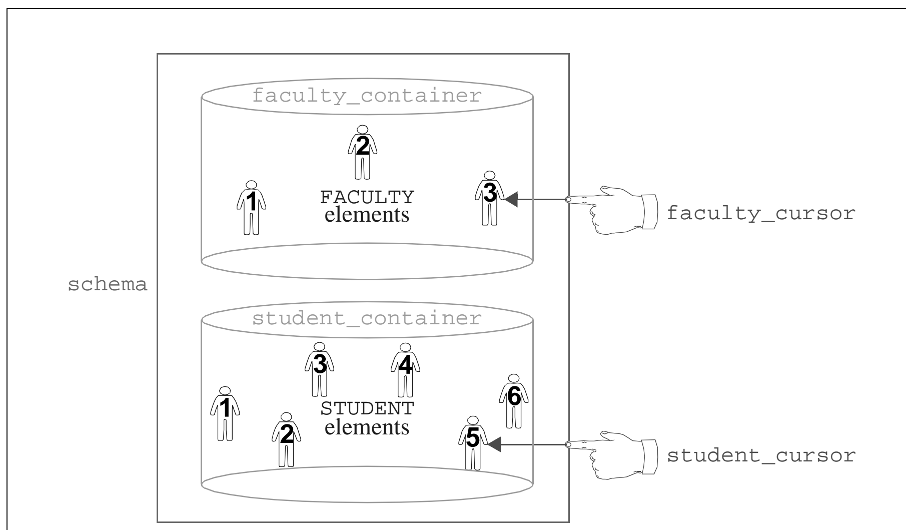

Figure 3.5 Relationships between elements, containers, cursors, and schema.

We chose C as the basis language for P2, because in 1992 when we began our implementation, it was the only language that satisfied the following criteria: (1) compatible with our target applications (written in C), (2) permits both high performance and system level programming, (3) standardized (as ANSI C), (4) easy to parse and extend (C grammars were freely available). P2 extends C in much the same way that C++

extends C. In fact, we observe in retrospect that C++ might have made a better basis for P2 than C, because C++ automatically provides several features (such as objects and polymorphism) that we had to implement manually in P2, and because C++ is backwards-compatible with C, it satisfies criteria (1) and (2). But in 1992, C++ was not very mature, and it did not satisfy criteria (3) and (4).

P2 fully integrates the element, container, cursor, and schema special data types with the C type system. All of these special data types are first-class; they can be used like any C type. Elements are simply C structures, so they are obviously integrated with the C type system. Containers, cursors, and schemas are integrated with the C type system by placing them in the same syntactic category as structures and unions. This is analogous to the way C++ places objects in the same category as structures and unions [Ell90].

Since containers and cursors are first-class, we are able to pass arguments of these types to procedures. But, each container and cursor declaration yields a new type. For example, `faculty_container` and `student_container` have different types. If we implement a procedure to print `faculty_container`, we cannot reuse this procedure to print the `student_container`; we would get a type mismatch error. We would like to be able to write a *polymorphic* procedure that accepts arguments of both types. To this end, P2 provides the **`generic_container`** and **`generic_cursor`** data types. Generics are specific to an element type, but not a container and cursor type. Figure 3.6 shows the declaration of a generic container and cursor for `PERSON` elements, a procedure that prints the cardinality and contents of a generic container and cursor, and a pair of calls to that procedure. Generics are analogous to virtual base types in C++: `GK` is the virtual base type of `faculty_container` and `student_container`, `GC` is the virtual base type of `faculty_cursor` and `student_cursor`. Generics are implemented using *operation vectors*, which are analogous to virtual function tables in C++ [Ell90].

```
// Generic container and cursor declarations.
typedef generic_container <PERSON> GK;
typedef generic_cursor <PERSON> GC;

// Generic procedure declaration.
void print_people (GK gk, GC gc)
{
  printf("cardinality = %d\n", cardinality(gk));
  foreach(gc) { printf("{ %d, \"%s\" }\n", gc.group, gc.name); }
}

// Generic procedure calls.
print_people((GK) &faculty_container, (GC) &faculty_cursor);
print_people((GK) &student_container, (GC) &student_cursor);
```

Figure 3.6 Generics.

3.1.2 Operations

P2 currently supports the realms shown in Figure 3.7. The MEM realm contains the lowest-level memory manager operations, typically operations on elements and containers, but not cursors. The DS realm contains higher level functionality, including cursor operations. The LINK and XACT realms are discussed in Chapters 4 and 5, respectively. The TOP realm contains all the operations that are user-visible. Typically, TOP inherits these operations from lower levels, but the foreach and rofeach operations are unique to the TOP realm. Figure 3.10 shows all the user-visible operations in the DS and MEM realms, as well as the operations unique to the TOP realm.

Realm	Semantics
TOP	Topmost, highest-level. User-callable.
XACT	Resource manager
LINK	Links and joins.
DS	Data structure.
MEM	Memory manager, lowest-level.

Figure 3.7 Realms.

Note that we deliberately specified the semantics of these operations vaguely–that is, informally and at a very high level. A more formal and detailed specification of these semantics would doubtless be useful. But, such a specification is impractical, because the exact semantics of the operations depends on their implementation. The large variety of implementations possible for these operations yields a large variety of interpretations for

these operations, as we will see in Section 3.2. Attempting to enforce a more exact semantics could limit the variety and efficiency of possible implementations. Figure 3.8 shows the user-visible operations of the `DS` and `MEM` realms.

As a convenience for users, P2 also provides shorthands (a.k.a., syntactic sugar) for the `upd()` and `ref()` operations. Figure 3.9 shows these shorthands. We found that these shorthands substantially reduce source code size and increase readability, especially in complicated expressions.

Figure 3.10 shows those operations of the `DS` and `MEM` realms that are not user-visible. These operations are used internally during the code generation process (see Section 3.2).

3.2 Feature Specification

Coding LWDB applications in terms of P2 data types is straightforward. The second phase of P2 application development is to define the features that the application's LWDB is to support and to declare how implementations of the P2 data types are to be generated. The key to our generative approach is to use a domain model of families of P2 data type implementations (i.e., families of LWDBs), where individual members of this family have a precise and unique specification in the model. We used GenVoca to express our domain model of LWDBs.

The motivation for these generators is the feature combinatorics problem outlined in Chapter 2–customized systems implement *m* features out of a possible *n* features. Rather than building an exponential number of monolithic systems that offer unique sets of features, one should build systems by composing primitive components that encapsulate individual features. Thus, by making feature combinatorics explicit, it is possible to describe vast families of systems with a relatively small number of components. In P2, a target LWDB is specified as a composition of P2 components.

DS	MEM	Operation	Semantics
•		void adv(cursor)	Advance cursor to next qualified element.
	•	void alloc(cursor)	Allocate an element of fixed-size.
	•	void allocv(cursor, expr)	Allocate an element of variable-size expr.
•		unsigned cardinality(container)	Return number of elements in container.
•	•	void close_cont(container)	Close connection to container.
•		int cont_id(container)	Return container operation vector identifier.
•		int curs_id(cursor)	Return cursor operation vector identifier.
•		void delete(cursor)	Remove referenced element from container.
•		void delete_curs(cursor)	Finalize cursor (e.g., delete referenced element if its reference count becomes 0). Opposite of init_curs.
•		BOOLEAN deleted(cursor)	Return TRUE iff element currently referenced by cursor has been deleted.
•		BOOLEAN end_adv(cursor)	Return TRUE iff cursor has been advanced past end of container.
•		BOOLEAN end_rev(cursor)	Return TRUE iff cursor has been reversed past beginning of container.

		foreach(cursor)
		{ *statement;* }
			Iterate *forward* using cursor. Execute *statement* for every element. Analogous to:
			reset_start(cursor);
			while (!end_adv(cursor));
			{ *statement;* adv(cursor); }

•		void getrec(cursor, element)	Copy referenced element into element.
•	•	void init_cont(container)	Initialize container to empty. Opposite of delete_curs.
•		void init_curs(cursor)	Initialize cursor (position is undefined).
•		void insert(cursor, element)	Add element to container referenced by cursor.
•		void insertv(cursor, element, expr0, expr1)	Add element of (variable) size expr0 into container referenced by cursor, copy only first expr1 bytes.
•		datatype iref(cursor, field)	Return field of element referenced by *cursor.
•	•	void open_cont(container)	Open container.
•	•	void open_cont_number(container, number)	Open container with file name suffix number (used primarily by log manager, see Chapter 5).
•		BOOLEAN overflow(container)	Return TRUE iff inserting element of fixed-size into container will cause it to overflow.
•		BOOLEAN overflowv(container, expr)	Return TRUE iff inserting element of variable-size expr into container will cause it to overflow.
•		void put_op_vec_c(cursor)	Store cursor operation vector.
•		void put_op_vec_k(container)	Store container operation vector.
•		BOOLEAN query(cursor)	Return value of applying retrieval predicate to referenced element.
•		datatype ref(cursor, field)	Return field of referenced element.
•		unsigned refcount(cursor)	Return number of cursors referencing element.
•		void reset_end(cursor)	Position cursor on *last* element in container.
•		void reset_start(cursor)	Position cursor on *first* element in container.
•		void rev(cursor)	Position cursor on *previous* element in container.
		rofeach(cursor){ *statement* }	Like foreach, but iterate *backwards*.
•		unsigned serial_number(container)	Like cardinality, but return number of elements *ever added* to container.
•		void swap(cursor0, cursor1)	Swap elements referenced by cursor0 and cursor1
•	•	void sync_cont(container, expr0, expr1, BOOLEAN)	Write to disk container memory in range [expr0, expr1). If BOOLEAN is TRUE, wait for operation to complete before returning.
•		unsigned timestamp(cursor)	Return timestamp of referenced element.
•		void upd(cursor, field, expr)	Assign expr to field of referenced element.

Figure 3.8 DS and MEM realm user-visible operations.

Shorthand	Semantics
cursor.field = expr;	This assignment statement is translated to: upd(cursor, field, expr)
cursor.field	Except when it appears as the left hand side of an assignment statement, this is translated to: ref(cursor, field)

Figure 3.9 Shorthands for the upd() and ref() operations.

DS	MEM	Operation	Semantics
•	•	void ddlhint(argc, argv)	Process annotations.
•	•	void optimize(cursor)	Perform static query optimization on cursor.
•	•	void verbatim_c(cursor)	Generate cursor-specific code.
•	•	void verbatim_k(container)	Generate container-specific code.
•	•	void verbatim_l(layer)	Generate layer-specific code.
•	•	void verbatim_s(schema)	Generate schema-specific code.
•	•	void xform(element, container, cursor)	Transform element, container, and cursor.

Figure 3.10 DS and MEM realm internal operations.

3.2.1 Components

Figures 3.11, 3.12, and 3.13 show the components that implement the TOP, DS, and MEM realms respectively. Ideally, every GenVoca component encapsulates a single feature. In general, this is true in P2, but Figure 3.14 shows the few exceptions. The conceptual component encapsulates many features for *convenience*. The user can specify this single layer and get many useful features. The top2ds_qualify component combines two features for *performance*. The code generated by top2ds_qualify can be significantly faster than the code generated by top2ds and qualify separately. This is an example of two features that cannot be fully decomposed (see Chapter 6). It is interesting to note that hash_array and hash_array_overwrite do *not* combine features. Rather, they are specializations of the array component in which the placement of elements in the array is not sequential, but is instead determined by a hash function.

Component	Semantics
conceptual [DS]	Many features. Intended as a convenience for novice users.
top2ds [DS]	Add foreach() and rofeach() operations.
top2ds_qualify [DS]	Like top2ds, but also provide qualification.

Figure 3.11 TOP realm components. See Figure 3.14 for details.

Component	Semantics
array [MEM]	Allocate elements sequentially.
avail [DS]	Free list of deleted elements available for reuse.
bintree [DS]	Binary tree (unbalanced).
cardinality [DS]	Add cardinality() operation.
container_structure [DS]	Add all fields of a given structure to container.
cursor_structure [DS]	Add all fields of a given structure to cursor.
delflag [DS]	Flag deleted elements, but do not reuse them.
dlist [DS]	Doubly-linked list.
dlist_deque [DS]	Double-ended queue implemented by a doubly-linked list.
free [DS]	free() deleted elements with refcount() equal to 0.
generic [DS]	Implement generic_container and generic_cursor by proceduralizing operations on them using operation vector.
generic_funcall [DS]	Proceduralize (a.k.a., *outline*) all operations using operation vector; That is, put code into separate function body. Opposite of inline.
generic_init [DS]	Initialize operation vector.
hash [DS]	Hash table.
hash_array [DS]	Combined hash and array layer. Hash conflicts are errors.
hash_array_overwrite [DS]	Like hash_array, but hash conflicts are *not* errors.
hashcmp [DS]	Speed string comparisons by pre-hashing strings.
hpredindx [DS]	Predicate indexed hash table.
htlist [DS]	Timestamp ordered hash list.
id [DS]	Add cont_id() and curs_id() operations.
inbetween [DS]	After deletion, position cursor on next element.
init_cont_function [DS]	Call given function when container is initialized.
kcur [DS]	Add pointer from member cursors to composite cursor.
lpredindx [DS]	Predicate indexed linked list.
ltlist [DS]	Timestamp ordered linked list.
malloc [MEM]	Allocate elements from a heap using malloc().
malloc_free [MEM]	Like malloc, but free() deleted elements.
malloc_multi [MEM]	Like malloc, but allocate multiple elements per malloc block.
mlist [DS, TOP]	Multi-list indexing.
named_funcall [DS]	Proceduralize operations using named functions.
null [DS]	Do nothing. Intended for debugging.
odlist [DS]	Ordered, doubly-linked list.
orderby [DS, TOP]	Implement the cursor sort criterion (orderby).
part [TOP, TOP]	Partition elements into two separate structures.
predindx [DS]	Predicate index.
qsort [MEM]	Like array, but quicksorted.
qualify [DS]	Implement the cursor selection predicate (where).
red_black_tree [DS]	Red-black tree (balanced binary tree).
refcount [DS]	Adds refcount() operation.
serial_number [DS]	Adds serial_number() operation.
slist [DS]	Singly-linked list.
slist_prev [DS]	Like slist, but cursor maintains pointer to previous element.
slist_queue [DS]	Queue implemented by a slist.
splaytree [DS]	Splaytree (balanced binary tree).
timestamp [DS]	Add timestamp() operation.
tlist [DS]	Timestamp ordered list.
trace [DS]	Print operation names before execution. Intended for debugging.
vtimestamp [DS]	User-specified timestamp.

Figure 3.12 DS realm components.

Component	Semantics
`transient [MEM]`	Transient, non-shared memory.
`mmap_persistent [MEM]`	Persistent, shared memory implemented using UNIX `mmap()`.
`mmap_shared [MEM]`	Like `mmap_persistent`, but transient.

Figure 3.13 MEM realm components.

```
conceptual[DS] = top2ds_qualify[generic_init[generic[generic_funcall[
  orderby[inbetween[qualify[DS]],
    top2ds[inbetween[dlist[malloc[transient]]]]]]]]]

top2ds_qualify[DS] = top2ds[qualify[DS]]
```

Figure 3.14 conceptual and top2ds_qualify component type expressions.

3.2.2 Implementation

P2 is an extensible language. Whenever a new layer is added to P2, new lexical tokens and grammar rules may be needed to parse the layer's annotation. We found the grammar for ANSI C to be too complicated to modify when new annotations were added. Instead, we chose to organize P2 as a pipeline of translators. The `ddl` translator is defined by a simple grammar that parses **`typex`** and **`container`** declarations into an "internal" syntax that can be easily parsed by an ANSI C grammar in the P2 backend (`pb`) translator. If a new layer is added to P2, the `ddl` translator is automatically regenerated; the `pb` compiler remains unchanged.

P2 employs a third translator (`xp`) for transforming high-level layer specifications into ANSI C. One of the lessons learned from the Genesis [Bat88] and Predator [Sir94] projects was that layer implementors had to know far too many details about generator internals to write new layers. A simple specification language was needed to write the translation rules for data types and operations; the compiler for this language would expand these rules and mechanically generate boilerplate information (e.g., standard type declarations, type definitions, code templates, standard error checking) that is common to all components.

The pipeline of `xp`, `ddl`, and P2 backend preprocessors that forms the P2 compiler is show in Figure 3.15.

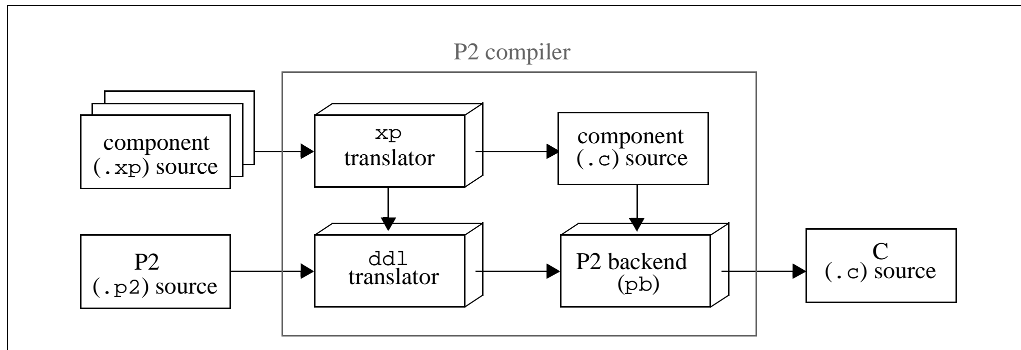

Figure 3.15 P2 architecture.

As an example of the `xp` component specification language, consider the `upd()` operation which updates field `f` of the object referenced by cursor `c` with the value `v`. For the `bintree` layer, if `f` is the sort field, then the object to be updated must be unlinked from the list, updated, and then relinked into its new position. Otherwise, the update is passed directly to the next lower layer, as it would have no effect on this list data structure. This rewrite would be specified as shown in Figure 3.16.

```
// Generate the code necessary to add an element to a binary tree.
upd(cursor, field, expr)
{
  // If f is the sort field.
  if (strcmp (f, %a.sort_field) == 0)
  %{
    unlink(cursor);            // Remove object from binary tree.
    upd(cursor, field, expr); // Call down to lower layers.
    link(cursor);              // Add object to binary tree.
  %}
  else
  %{
    upd(cursor, field, expr); // Call down to lower layers.
  %}
}
```

Figure 3.16 upd() operation of bintree component

Note that `xp` generates all data type declarations for this specification. Furthermore, all text enclosed within `%{...%}` is a code fragment that is to be generated; statements outside of `%{...%}` are to be executed by the P2 compiler. The `%a` symbol refers to a

C structure that contains the information about the layer's annotations. The `%a` symbol is expanded by `xp` into the C expression that references this structure.

The P2 backend (`pb`) recognizes operations on special data types and replaces them with the fragment that is generated for the operation by the first layer of that container's type expression. Calls to lower level operations are replaced, recursively, with their implementing fragments until a terminal layer is reached. Note that data structure specific optimizations in the form of partial evaluations are part of this expansion process. This can be seen in the `upd()` specification above, where depending on the field input to `upd()`, different code fragments are generated. Thus, embodied in layers are domain-specific optimizations that no general-purpose compiler could offer.

P2 also has a query optimizer. Given a retrieval (a.k.a., qualification) predicate, several layers in a data structure could process the query; P2 determines which layer would perform the retrieval most efficiently. P2 associates with each layer a cost function which estimates the cost of processing the query. P2 polls each layer and selects the layer that returns the lowest cost estimate. In this way, the cheapest plan (data structure traversal) for processing a query is selected.

3.2.3 Transformation

All P2 components simultaneously and consistently refine element, container, cursor, and schema data types. For example, the `dlist` component refines a container of elements into a container whose elements are linked together onto a doubly-linked list. That is, `dlist` encapsulates the following data refinements:

- element types are augmented with `next` and `prev` pointer fields
  (for double linking).
- cursor types are unchanged.
- container types are augmented with `first` and `last` pointer fields
  (for head and tail list accessing).
- `schema` type is unchanged.

Figure 3.17 shows how these data refinements are specified using the `xform()` operation of the `dlist` component.

```
// Perform the data refinements specified by the dlist component.
xform(element, container, cursor)
{
  // Add to element, pointers to next and previous elements.
  add element   : struct element *next;
  add element   : struct element *prev;
  // Add to container, pointers to first and last elements.
  add container : struct element *first;
  add container : struct element *last;
  // Call down to lower layers. That is, perform any data refinements
  // specified by components that appear below this one in the type
  // expression.
  xform(element, container, cursor);
}
```

Figure 3.17 xform() operation of dlist component.

Figure 3.18 shows the result of applying the refinements encapsulated by `dlist` to the unrefined element, container, cursor, and schema shown in Figure 3.5.

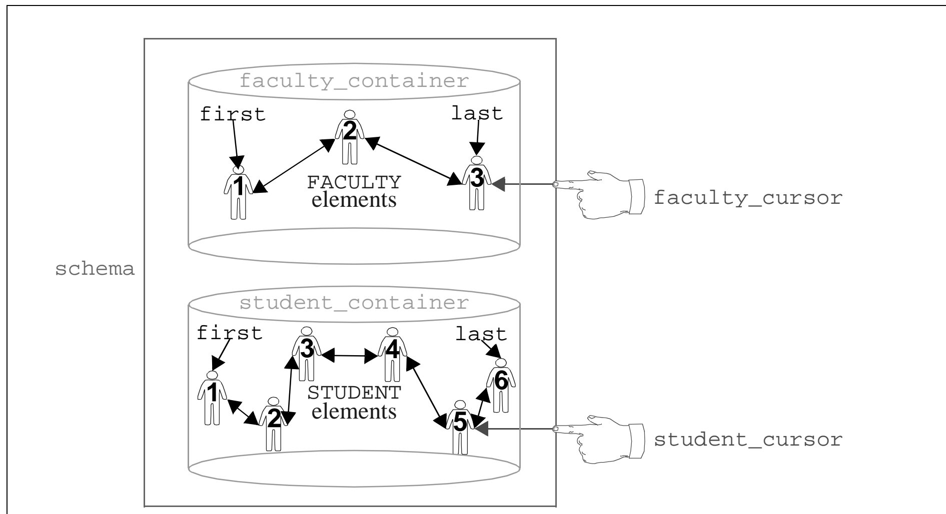

Figure 3.18 dlist refinements. Contrast with Figure 3.5.

Figure 3.19 shows how the insertion algorithm is specified using the `insert()` operation of the `dlist` component.

The `dlist` component is a large-scale transformation because it automatically refines element, container, and cursor data types by adding new data members, algorithms,

```
// Generate the code necessary to add an element to doubly-linked list.
insert(cursor, element)
%{
  // Call down to lower layers. That is, generate any code specified by
  // components that appear below this one in the type expression.
  insert(cursor, element);
  // Insert element into doubly-linked list.
  call link(cursor);
%}
```

Figure 3.19 insert() operation of dlist component.

and optimizations (e.g., query optimization, code inlining, partial evaluation) for the doubly-linked list feature. So, the way to understand the `dlist` transformation is that it takes a P2 program (with cursors, containers, elements) as input, and produces a refined P2 program (with refined cursors, containers, elements) as output. By cascading transformations (i.e., composing components in type expressions), implementation details of the target LWDB are progressively revealed.

Some simple P2 type expressions are shown in Figure 3.18; we will first explain `A`, and then `B`. Type expression `A` means that the elements of the container will be linked together onto a doubly-linked list (by `dlist`). List nodes will be marked deleted, but not reused (by `delflag`). List nodes with the delete flag will be allocated from a heap using `malloc()` (by `malloc`). The heap will be stored in transient, non-shared memory (by `transient`). Note that *none* of the components in type expression `A` require additional information in order to perform their transformations; thus, Figure 3.2 provides complete container declarations.

```
// Declaration of type expressions A and B.
typex {
  A = conceptual[cardinality[dlist[delflag[malloc[transient]]]]];
  B = conceptual[bintree[dlist[avail[malloc[mmap_persistent]]]]];
}
```

Figure 3.20 Simple P2 type expressions.

Figure 3.21 shows the element, container, and cursor types that result when P2 cascades the data refinements specified by the components in type expression `A`; comments indicate which component added each field, and what its purpose is.[2]

Code	Component	Semantics
`struct element {`		
`  int group;`		
`  char name[10];`		
`  struct element *next;`	`// dlist`	Pointer to next element.
`  struct element *prev;`	`// dlist`	Pointer to previous element.
`  int delete_flag;`	`// delflag`	TRUE iff element was deleted.
`  char misc[10];`		
`}`		
`struct container {`		
`  BOOLEAN initialized;`	`// conceptual`	TRUE iff container was initialized.
`  int (**operation_vector);`	`// conceptual`	Array of addresses of container operations.
`  unsigned cardinality;`	`// cardinality`	Number of elements in container.
`  struct element *first;`	`// dlist`	Pointer to first element.
`  struct element *last;`	`// dlist`	Pointer to last element.
`}`		
`struct cursor {`		
`  struct element *obj;`	`// conceptual`	Element referenced by cursor.
`  struct container *con;`	`// conceptual`	Container referenced by cursor.
`  int (**operation_vector);`	`// conceptual`	Array of addresses of cursor operations.
`  BOOLEAN inbetween;`	`// conceptual`	TRUE iff cursor was advanced after delete.
`}`		

Figure 3.21 element, container, and cursor types as refined by type expression A.

Figure 3.22 shows the `insert()` algorithm that results when P2 cascades the data refinements specified by the components in type expression `A`; comments indicate which component added each lines. `cardinality` increments its count of the number of elements in the container. `malloc` allocates memory and copies the new element into it. `delflag` clears the element's delete flag. `dlist` links the element into the doubly linked list. Thus, code from different layers is weaved together.

Code	Component
`insert(cursor, element)`	
`{`	
`  (cursor.con)->cardinality++;`	`// cardinality`
`  (cursor).obj = malloc (sizeof(struct element));`	`// malloc`
`  memcpy(cursor.obj, element, sizeof(struct element));`	`// malloc`
`  (cursor.obj)->delete_flag = FALSE;`	`// delflag`
`  link(cursor);`	`// dlist`
`}`	

Figure 3.22 insert() operation as refined by type expression A.

Type expression `B`, on the other hand, declares a very different storage structure than `A`. `B` means that the elements of the container will be linked together onto a binary tree

2. A *stable* component leaves the cursor referencing an element after it is deleted. An *unstable* component does not; it advances the cursor to the next (non-deleted) element. The `inbetween` field is needed to implement this advance.

(by bintree). Tree nodes will be linked together onto a doubly-linked list (by dlist). List nodes with the delete flag will be allocated from a heap using malloc() (by malloc). The heap will be stored in persistent, shared memory implemented using UNIX mmap() (by mmap_persistent). Note that *some* of the components in type expression B, unlike A, require additional information in order to perform their transformations: a key for bintree, and filename and size (in bytes) for mmap_persistent. This additional information is passed to components in the form of realm parameters called *annotations*; thus, Figure 3.2 must be modified with annotations as shown in Figure 3.23 to provide complete container declarations.

```
// Container declarations using type expression B.

container <PERSON> stored_as B with {
  bintree key is name;
  mmap_persistent file is "/tmp/foo" with size 10000;
} faculty_container;

container <PERSON> stored_as B with {
  bintree key is name;
  mmap_persistent file is "/tmp/foo" with size 10000;
} student_container;
```

Figure 3.23 Example container declarations using type expression B.
Contrast with Figure 3.2.

As these examples suggest, P2 programmers are armed with a small handful of P2 components that can be composed in vast numbers of ways to produce large families of distinct LWDB implementations. This powerful feature allows P2 users to explore different LWDBs's implementations easily by altering just a container's type expression and recompiling; no other source code modifications are needed. Further details about type expressions and P2 components are discussed in [Bat93, Bat94a-c].

Chapter 4

LEAPS

The LEAPS (*Lazy Evaluation Algorithm for Production Systems*) production system compilers, OPS5.c and DATEX, are classical lightweight database applications. The LEAPS compilers produce sequential executables of OPS5 rule sets that are orders of magnitude faster than those produced by previous systems [Mir90-91, Bra91]. A LEAPS executable is a lightweight database application, because it represents its database of assertions as a set of containers, and because it uses unusual search algorithms and novel container implementations to enhance rule processing efficiency; no heavyweight DBMS offers the performance or features needed by LEAPS.

4.1 The LINK Realm

The LEAPS algorithms require multi-container joins. Support for joins requires additional abstractions and operations not provided by the `TOP`, `DS`, or `MEM` realms. P2 uses composite cursors to *abstract* joins, and the `LINK` realm to *implement* operations on them.

A *link* is a relationship between two elements. A *join* of containers C_1, C_2, ..., C_n is a set of n-tuples (e_1, e_2, ..., e_n) where element e_i is a member of container C_i and for each i = 1, 2, ..., n-1, there is a link between e_i and e_{i+1}. A *composite cursor* is a placemarker into a join that imposes a one-at-a-time ordering to the tuples of the join. Composite cursors are constructed from n member cursors, one per container of the join. When

each member cursor i references element e_i, the composite cursor references the tuple of elements (e_1, e_2, ..., e_n).

An example join of `faculty_container` and `student_container` is depicted in Figure 4.1. This join consists of the elements (`f1`, `s1`), (`f1`, `s2`), (`f1`, `s3`), (`f2`, `s4`), (`f2`, `s5`), (`f3`, `s6`). This composite cursor currently references the tuple (`f3`, `s6`).

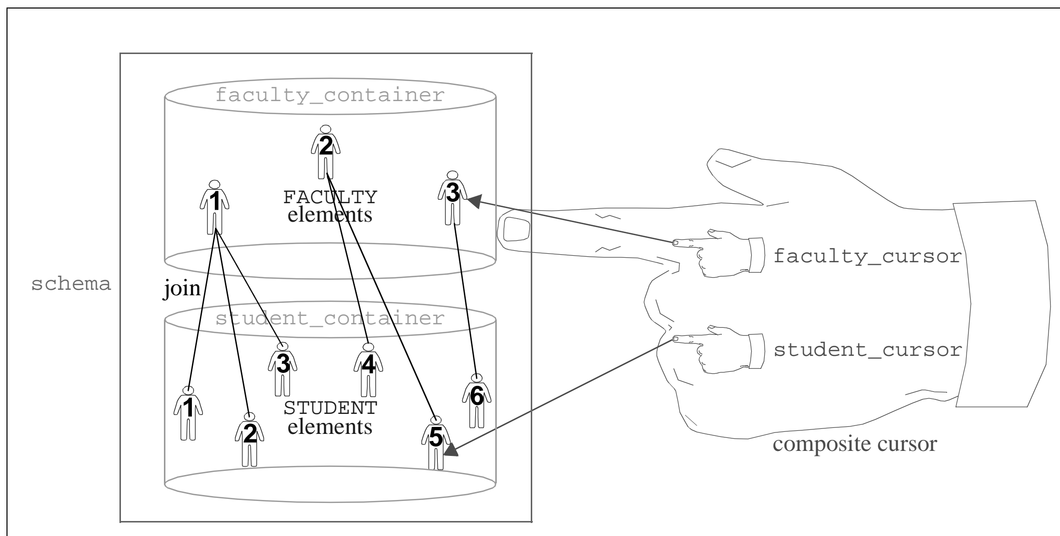

Figure 4.1 Example join.

In P2, composite cursors are declared using the **`compcurs`** data type. Composite cursors are parameterized by the n containers to be joined and optionally by a **`given`** clause, join predicate, and/or **`valid`** predicate. The n containers need *not* be unique; that is, a container *may* be joined with itself (self-join). Figures 4.2 and 4.3 show the declaration of `g` and `c`, composite cursors over the **`schema`** `s` member cursors `faculty_container` and `student_container` (defined in Chapter 3).

```
// Composite cursor declaration with given clause.
compcurs
  <faculty_cursor s.faculty_container, student_cursor s.student_container>
  given < faculty_cursor >                              // Given clause.
  where "$student_cursor.group == $faculty_cursor.group" // Join predicate.
g;
```

Figure 4.2 Example composite cursor declaration with given clause.

```
// Composite cursor declaration with valid predicate.
compcurs
  <faculty_cursor s.faculty_container, student_cursor s.student_container>
  where "$student_cursor.group == $faculty_cursor.group" // Join predicate.
  valid "!deleted($faculty_cursor)"                      // Valid predicate.
v;
```

Figure 4.3 Example composite cursor declaration with valid predicate.

These composite cursors have a join predicate, and **given** clause or **valid** predicate, respectively. Note that the member cursor names are merely aliases, and do not refer to previously declared cursors. The dollar sign syntax of single container cursor declarations is extended for composite cursors such that the dollar sign is followed by the member cursor name. Thus in our example, the join predicate "$student_cursor.group == $faculty_cursor.group" denotes those tuples where the group field of the elements referenced by the student_cursor and faculty_cursor member cursors are equal. Thus these composite cursors specify equijoins on the group field. The **given** clause and **valid** predicate specify subsets of the tuples of a join

The **given** clause specifies a subset of join tuples by fixing the element referenced by one or more member cursors. That is, a composite cursor with a **given** clause will not change the position of the **given** member cursor. Thus, if the **given** clause specifies member cursor i, which currently references (i.e., *seeds*) element e_i, the join will include only those tuples that contain element e_i. For example, suppose that we position the faculty_cursor on element f2 and then during the join we print the faculty and student elements. Figure 4.4 (Full Join) shows how if tuples were computed using a full join, we would print tuples containing faculty elements other than f2–(f1, s1), (f1, s2), and (f1, s3), with element f1; and (f3, s6) with element f3. Figure 4.4 (Given Join) shows how when tuples are computed using the **given** join of Figure 4.2, the tuples containing faculty elements other than f2 are skipped.

The **valid** predicate specifies a subset of join tuples via an arbitrary predicate. Unlike a selection or join predicate, a **valid** predicate is tested for *every* tuple and can *not* be optimized away. Thus, the **valid** predicate permits the elements in the join to be updated or deleted *during* the join. This is perhaps the most novel aspect of composite cur-

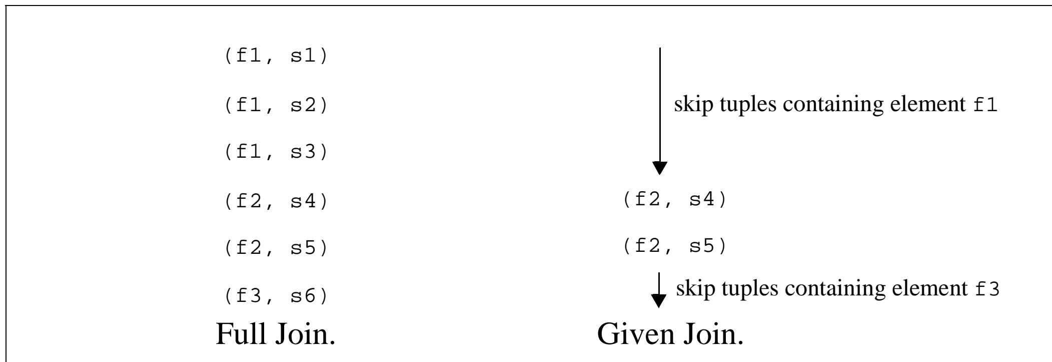

Figure 4.4 Full join vs. join given `faculty_cursor` = `f2`. See Figure 4.2.

sors. Unlike view updates (where changes are restricted [Kel82]), P2 updates are unrestricted and may effect the tuples that are subsequently retrieved. Thus, once an element of a tuple is deleted, that element should not belong to any subsequently retrieved tuple. For example, suppose that during the join we print the faculty and student elements and then delete the faculty element (see Figure 4.8). Figure 4.5 (Eager Join) shows how if tuples were computed using an eager (i.e. blind or set-at-a-time) join, we would print tuples with deleted elements–(`f1`, `s2`) and (`f1`, `s3`) after `f1` has been deleted and (`f2`, `s5`) after `f2` has been deleted. Figure 4.5 (Valid Join) shows how, when tuples are computed using the **`valid`** join of Figure 4.3, deletions are noted and tuples containing deleted elements are skipped.

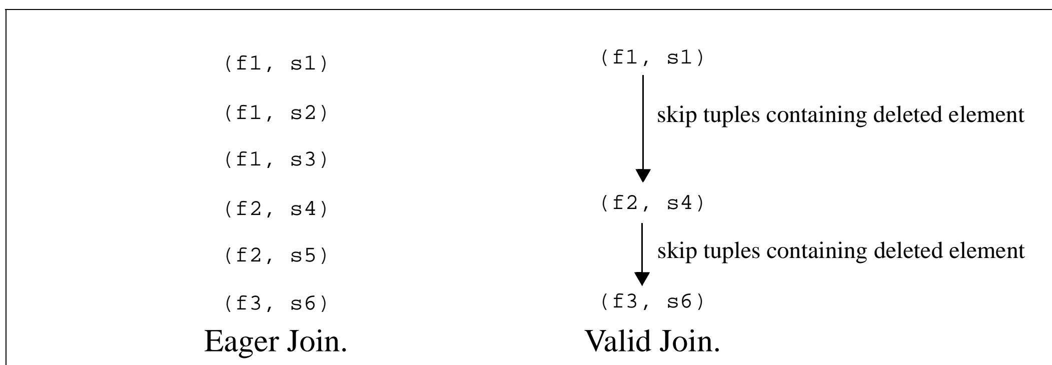

Figure 4.5 Eager join vs. join with valid clause `!deleted($faculty_cursor)`. See Figure 4.3.

Note that tuple validation is more general than merely testing for element deletion. A **valid** join can be used to skip elements satisfying other conditions. For example, the group field of a faculty or student element might be updated within a join. In this case, the element has not been deleted, but its modification may effect the sequence of (valid) tuples that can be produced. Thus tuple validation is a general-purpose feature that is useful, for example, in graph traversal and garbage collection algorithms to ensure correct executions.

The interface of composite cursors is called the LINK realm. This realm extends the DS realm with the additional operations shown in Figure 4.6.

Operation	Semantics
void advk(compcurs)	Advance compcurs to the next tuple in join.
BOOLEAN endk(compcurs)	Return TRUE iff comcpcurs has been advanced beyond the last tuple in join.
void foreachk(compcurs) { *statement*; }	Iterate forward using compcurs. Execute *statement* for every tuple in the join. Analogous to: resetk(compcurs); while (!endk(compcurs)); { statement; adv(compcurs); }
void initk(compcurs)	Initialize compcurs (position is undefined).
void resetk(compcurs)	Position compcurs on the first tuple in join.

Figure 4.6 LINK realm operations.

Figure 4.7 shows an example of code using the foreachk() operation. The code positions the faculty_cursor member cursor of composite cursor g on faculty element f2, and for each tuple in the join, prints the name field of the faculty_container and student_container elements. Thus this code demonstrates the **given** join shown in Figure 4.4.

```
// Position g.faculty_cursor on f2.
reset_start(faculty_cursor);
adv(g.faculty_cursor);
// Print.
foreachk(g)
{
  printf("(%s, %s)\n", g.faculty_cursor.name, g.student_cursor.name);
}
```

Figure 4.7 Example given join.

Figure 4.8 shows another example of code using the `foreachk()` operation. For each tuple in the join, the code prints the `name` field of the `faculty_container` and `student_container` elements, and then deletes the element from the `faculty_container`. Thus this code demonstrates the **`valid`** join shown in Figure 4.5.

```
// Print and delete.
foreachk(c)
{
  printf("(%s, %s)\n", c.faculty_cursor.name, c.student_cursor.name);
  delete(c.faculty_cursor);
}
```

Figure 4.8 Example `valid` join.

Figure 4.9 shows the components that implement the `LINK` realm. Each of these components maps a schema with links to a schema with fewer links. The stacking of `LINK` realm components defines a transformation that progressively removes links. Ultimately, the resulting schema is a collection of non-linked containers, that can be implemented via `DS` and `MEM` realm components. The `link2top` layer transmits `TOP` operations as is; if it receives any `LINK` operations, this means that no layer above it could process the operation. This is a fatal error and should not occur; thus `link2ds` is a safety net.

Component	Semantics
`LINK nloops[LINK]`	Implement links as nested loops joins.
`LINK ringlist[LINK]`	Implement links as pointers.
`LINK link2top[TOP]`	Treat `LINK` operations as errors, translate `TOP` operations as is.

Figure 4.9 `LINK` realm components.

4.2 The LEAPS Algorithms

As mentioned earlier, the LEAPS compilers, OPS5.c and DATEX, produce executables of OPS5 rule sets that are faster than those produced by previous systems, often outperforming OPS5 interpreters that use RETE-match or TREAT-match algorithms by several *orders* of magnitude [Mir90-91, Bra91]. These compilers translate OPS5 programs into C

programs. Besides the expected performance gains made by compilation, LEAPS relies on special algorithms and sophisticated data structures to make rule processing efficient. See [Bat94a].

Figure 4.10 shows the relationship between the LEAPS compilers, OPS5.c and DATEX, and P2. To reengineer LEAPS required us to translate OPS5 rule sets into a P2 program; this translator is called RL (*Reengineered Leaps*). The RL-generated P2 program is then translated into a C program by the P2 compiler, thus effectively accomplishing in two translation steps what the LEAPS compilers do in one. All of the LEAPS algorithms are embedded in the generated P2 program. In this section, we show that the LEAPS algorithms have an elegant specification in P2.

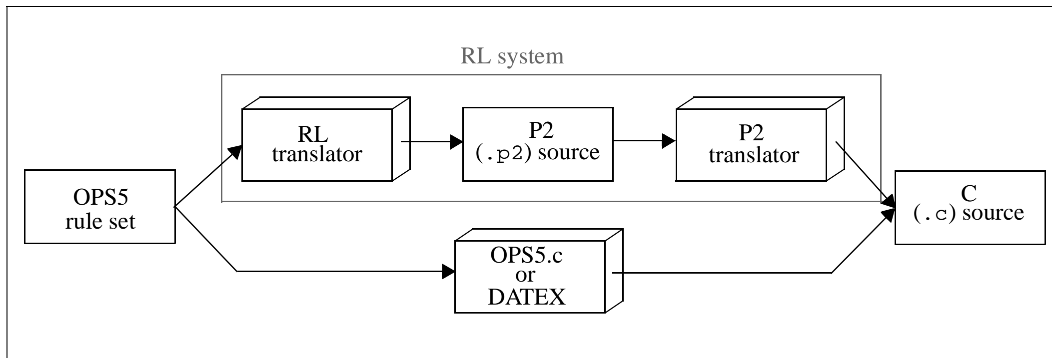

Figure 4.10 Relationship between the LEAPS compilers and RL.

4.2.1 `literalize` Statements and OPS5 Terminology

OPS5 rule sets begin with “container” declarations called `literalize` statements of the form shown in Figure 4.11.

```
(literalize guest name sex hobby)
```

Figure 4.11 Example OPS5 `literalize` statement.

The literalize statement in Figure 4.11 declares a container `guest` whose elements have fields `name`, `sex`, and `hobby`. The LEAPS compilers infer the data types of element fields; we chose to augment literalize statements by supplying the data type for each field.

Although this is a minor difference between the LEAPS compilers and RL, we note that the next generation LEAPS compiler (called VENUS [Bro94]), like RL, uses explicit typing of fields.

An OPS5 rule set is a sequence of rules/productions of the form shown in Figure 4.12.

```
(p make_path
  (context ^value make_path)
  (seating ^id <id> ^pid <pid> ^path_done no)
  (path ^id <pid> ^name <n1> ^seat <s>)
  -(path ^id <id> ^name <n1>)
  -->
  (make path ^id <id> ^name <n1> ^seat <s>))
```

Figure 4.12 Example OPS5 rule.

The name of the above production is `make_path`. Each of the clauses prior to the arrow (`-->`) are called *condition elements* (CEs). The first three are positive; the last (with the minus sign) is negative. Positive condition elements serve two purposes: (1) to express qualifications on containers and (2) to declare variable bindings. For example, the first CE of the `make_path` rule qualifies elements from the `context` container to those whose `value` field is "`make_path`". The second CE qualifies elements from the seating container to those whose `path_done` field is "`no`"; in addition, it sets variable `id` to the value of the `id` field of the qualified element and sets variable `pid` to the value of the `pid` field. The third CE qualifies elements from the path container whose `id` field equals the value of the variable `pid`; in addition, variable `n1` is assigned the value of the `name` field and variable `s` is assigned the value of the `seat` field. In all, the positive CEs of this rule identify 3-tuples (`context` element, `seating` element, `path` element) that satisfy a simple selection predicate.

Negated CEs are disqualification filters. The negated CE above disqualifies selected 3-tuples if there exists a `path` element whose `id` field matches the value of variable `id` and whose `name` field matches the value of variable `n1` and whose `seat` field matches the value of `s`. In general, there can be any number of negated CEs in a rule; however, there must at least one positive CE.

Clauses that follow the arrow (`-->`) are the *actions* of the rule. Once a tuple has been qualified, it fires the actions of the rule. Actions of OPS5 rules include element creation, deletion, and modification; calls to routines external to OPS5 are possible. In the `make_path` rule, the sole action is to insert a `path` tuple whose `id` field equals variable `id`, whose `name` field equals variable `n1`, and whose `seat` field equals variable `s`.

4.2.2 LEAPS Overview

Forward-chaining inference engines, including LEAPS, use a match-select-action cycle. Rules that can be matched (i.e., tuples found to satisfy their predicates) are determined; one tuple is selected and its corresponding rule is fired. This cycle continues until a *fix point* has been reached (i.e., no more rules can be fired). RETE-match [For82] and TREAT-match [Mir91] algorithms are inherently slow, as they *materialize* all tuples that satisfy the predicate of a rule. Materialized tuples are stored in data structures and have a negative impact on performance as they must be updated as a result of executing rule actions. A fundamental contribution of LEAPS is the *lazy* evaluation of tuples; that is, tuples are materialized only when needed. This approach drastically reduces both the space *and* time complexity of forward-chaining inference engines and provides LEAPS with its phenomenal increase in rule execution efficiency.

LEAPS assigns a timestamp to every element to indicate when the element was inserted or deleted. (For reasons that we will explain later, elements are not updated. Instead, the old version is deleted and a new version is inserted). Whenever an element is inserted or deleted, a handle to that element is placed on a stack. In general, the stack maintains a timestamp ordering of elements, where the most recently updated element is at the top of the stack and the least recently updated element is at the bottom.[1]

During a rule execution cycle, the top element of the stack is selected. This element is called the *dominant object* (DO). The DO is used to seed the selection predicates

1. There is an exception: an important LEAPS optimization violates the timestamp ordering. This optimization, called shadow optimization, is discussed in Section 4.2.5.

of all rules. Rules are considered for seeding in a particular order. Rules are sorted by their number of positive condition elements; the more positive CEs, the sooner the rule will be seeded. Many rules have the same number of positive CEs; these rules are seeded in order in which they were defined in the rule set. As soon as it is determined that the DO cannot seed a tuple for a given rule, the next rule is examined.[2] When all rules have been considered, the DO is popped from the stack.

When a DO-seeded tuple is found, the corresponding rule is fired. The actions of the rule may invoke element insertions, deletions, and updates, which in turn will cause more elements to be pushed onto the stack. After a rule is fired, the selection of the next dominant object takes place. This execution cycle repeats; execution terminates when a fix-point is reached. This occurs when the stack is empty.

Note that an element may be pushed onto the stack twice: once when it is inserted and a second time when it is deleted. It is possible that a deleted element may be pushed onto the stack prior to the popping of its inserted element. That is, the stack may contain zero, one, or two references to any given element at any point in time.

The seeding of rule selection predicates by nondeleted elements has its obvious meaning. However, the seeding of rule predicates by deleted elements is not intuitively obvious, and its meaning is closely associated with the semantics of negation. Associated with every container C is a shadow container S. Every element that is deleted from C is inserted in S. The timestamp of an element e in C indicates when e was inserted; the timestamp of an element s in S indicates when s was deleted. Elements in S never undergo changes; they simply define the legacy of elements in C that previously existed. The purpose of shadow containers is to support time travel. The evaluation of negated CEs involves evaluating its predicate P on its container C over a period of time. That is, LEAPS asks questions like: is predicate P true from time t_0 to time t_1? The reason for this will become evident once the evaluation of negation is explained more fully.

---

2. Actually, DOs cannot be used to seed all rules in general. If a DO is from container C and C is not referenced in the selection predicate of rule R, then the DO cannot seed R. Thus, the set of rules that a DO can seed can be pruned at compile time to only those that actually reference the DO's container.

In the following sections, we will give more technical precision to the above description.

4.2.3 Rule Translation

The difficult part of converting OPS5 rules into P2 code is the translation of rule predicates to P2 composite cursor declarations; translating the actions of rules is straightforward. There are six steps in rule predicate translation.

Step 1 is conversion of qualifications of positive CEs to P2 predicates. Figure 4.13 (Original) shows the correspondence of a nonnegated rule predicate with a composite cursor declaration (Transformed). Note that each CE of the rule corresponds directly to a container that is to be joined. Also note that the use of **compcurs** aliases permit containers to be joined with themselves in an unambiguous way (container `cont_edge` is joined with itself using the aliases `c` and `d`).

Original

```
(p rule14
  (stage ^value labeling)
  (junction ^type tee ^base_point <bp> ^p2 <p1> ^p2 <p2> ^p3 <p3>)
  (edge ^p1 <bp> ^p2 <p1>)
  (edge ^p1 <bp> ^p2 <p3> ^label nil)
  -->
```

Transformed

```
#define query14 \
  "$a.value == 'labeling' \
   && $b.type == 'tee' \
   && $c.p1 == $b.base_point && $c.p2 == $b.p1 \
   && $d.p1 == $b.base_point && $d.p2 == $b.p3 && $d.label == 'nil'"

typedef compcurs < a stage, b cont_junction, c cont_edge, d cont_edge >
  where query14
curs14;
```

Figure 4.13 Rule Translation Step 1:
Conversion of Selection Predicates.

Recall that a central concept of rule processing in LEAPS is the seeding of rules by dominant objects. In order to support seeding, multiple copies of an OPS5 rule are spawned, one copy of each different condition element that is being seeded. Step 2 in the rule translation process is to replicate a composite cursor definition, one copy for each possible seed position. Figure 4.14 (Original) shows the format of a cursor declaration

produced in Step 1 and replication of this rule with different seeds (Transformed). Note that the effect of this rewrite is to translate an n-way join to a more efficient (n-1)-way join.

Original

```
typedef compcurs < a ..., b ..., c ..., d ... >
  where query14
curs14;
```

Transformed

```
typedef compcurs < a ..., b ..., c ..., d ... >
  given < a >
  where query14
curs14_a;

typedef compcurs < a ..., b ..., c ..., d ... >
  given < b >
  where query14
curs14_b;

typedef compcurs < a ..., b ..., c ..., d ... >
  given < c >
  where query14
curs14_c;

typedef compcurs < a ..., b ..., c ..., d ... >
  given < d >
  where query14
curs14_d;
```

Figure 4.14 Rule Translation Step 2:
Replication of Composite Cursors by Seeding.

OPS5 semantics imposes a fairness criterion that no tuple can fire a rule more than once. Fairness is achieved in LEAPS through the use of timestamps and temporal qualifications. Every element has a timestamp that indicates when it was last updated (i.e., inserted or deleted). OPS5 semantics are realized by requiring all elements of a tuple to have their timestamps less than or equal to the timestamp of the dominant object that seeded the tuple.[3] Figure 4.15 (Original) shows the format of a Step 2 cursor declaration

---

3. Things are actually a bit more complicated. In the case where a container C is being joined with itself, we want to eliminate pairs of the same object (`c`, `c`) which could be generated more than once. To avoid such duplication, timestamp qualifications on containers "to the left" of the container of the seeding dominant object to have timestamps ≤ the timestamp of the DO and qualifications on containers "to the right" of the seeding container to be < the timestamp of the DO [Bra93b]. The notion of "left" and "right" is determined by the order in which containers are listed to be joined. Note that the LEAPS compilers *did* generate multiple tuples as they did not enforce the ideas outlined in this footnote.

and the addition of temporal predicates to the where clause of the cursor (Transformed). Note that `ts` is the name of the `timestamp` field of every element.

Original

```
typedef compcurs < a ..., b ..., c ..., d ... >
  given < a >
  where query14
curs14_a;
```

Transformed

```
#define temporal_query14 \
  "$b.ts <= dominant_timestamp \
   && $c.ts <= dominant_timestamp \
   && $d.ts <= dominant_timestamp"
typedef compcurs < a ..., b ..., c ..., d ... >
  given < a >
  where query14 "&&" temporal_query14
curs14_a;
```

Figure 4.15 Rule Translation Step 3:
Addition of Temporal Predicates.

Once a rule is fired, the composite cursor is placed on a stack, thereby suspending its execution. At some later time, when the element that seeded the composite cursor again becomes dominant, the composite cursor is popped and advanced to the next tuple. During the time the cursor is on the stack, any or all of the elements of the last tuple it produced could have been modified or deleted. Consequently, advancements of composite cursors must be validated. This is accomplished by adding a **`valid`** predicate to each cursor declaration. Figure 4.16 (Original) shows a Step 3 cursor definition; and the addition of the **`valid`** predicates (Transformed).[4]

Negated CEs are disqualification filters. The LEAPS interpretation of negation is depicted in Figure 4.17. An element e is created at time t_0 and seeds a tuple by advancing a composite cursor at times t_1 ... t_4. Let P be the predicate of a negated CE and t be the time of a composite cursor advancement. LEAPS determines if P is true at time t or at any time since e has been created.

4. Technically, cursors are not popped off the stack; pointers to cursors are overwritten. A stack item contains a pointer to an element, the identifier of the container to which the element belongs, a pointer to a composite cursor whose execution has been suspended, and an identifier of the rule to which the composite cursor belongs. When a cursor is popped, the cursor pointer is set to null. A stack item is popped only when all rules have been seeded.

Original

```
typedef compcurs < a ..., b ..., c ..., d ... >
  given < a >
  where query1 "&&" temporal_query14
curs14_a;
```

Transformed

```
#define valid_query \
  "!deleted($a) \
   && !deleted($b) \
   && !deleted($c) \
   && !deleted($d)"

typedef compcurs < a ..., b ..., c ..., d ... >
  given < a >
  where query14 "&&" temporal_query14
  valid valid_query14
curs14_a;
```

Figure 4.16 Rule Translation Step 4:
Addition of Validation Predicates.

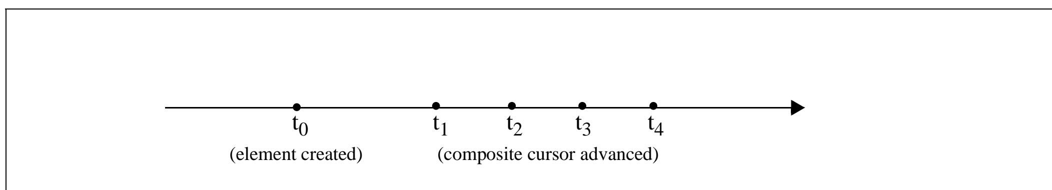

Figure 4.17 Interpretation of Negation.

This interpretation has two significant consequences. First, LEAPS must maintain a history of all container elements so that time can be "rolled back" to evaluate P. This is realized by creating a *shadow container* for each container to be a repository of the versions of elements that have since been modified or deleted. Because shadow elements are tagged with the timestamp of their removal from the primary (nonshadow) container, time travel is possible. Shadow containers are a major source of complexity in LEAPS. Second, because predicate P may be valid at some time t does not mean that P is valid at later times. Consequently, predicate P must be used to filter elements both in the **`where`** clause of a composite cursor and in the **`valid`** clause. Figure 4.18 shows a rule with negation (Original) and its P2 composite cursor counterpart that seeds in position a (Transformed). Note that `N5_4()` is a `BOOLEAN` function (generated by RL) that expresses the filter of the negated CE.[5]

Original

```
(p rule5
  (stage ^value detect_junctions)
  (edge ^p1 <bp> ^p2 <p2> ^joined false)
  (edge ^p1 <bp> ^p2 <p3> ^p2 <> <p2> ^joined false)
  -(edge ^p1 <bp> ^p2 <> <p2> ^p2 <> <p3>)
  -->
```

Transformed

```
#define query5 \
  "$a.value == 'detect_junctions' \
   && $b.joined == 'false' \
   && $c.p1 == $b.p1 && $c.p2 != $b.p2 && $c.joined == 'false' \
   && N5_4(&$b,&$c)"

#define temporal_query5 \
  "$a.ts <= dominant_timestamp \
   && $b.ts <= dominant_timestamp \
   && $c.ts <= dominant_timestamp"

#define valid_query5 \
  "!deleted($a) \
   && !deleted($b) \
   && !deleted($c) \
   && N5_4(&$b,&$c)"

typedef compcurs < a cont_stage, b cont_edge, c cont_edge >
  given < a >
  where query5 && temporal_query5
  valid valid_query5
curs5_a;
```

Figure 4.18 Rule Translation Step 5:
Placement of Negated Predicate Filters.

Finally, it is possible for shadow container elements to become dominant. The idea here is that a container element may block the qualification of tuples because it satisfied a negated CE filter. With the deletion of this element, previously disqualified (or *blocked*) tuples may now be qualified (*unblocked*). Tests for unblocked tuples are created by (a) modifying the original OPS5 rule by replicating the negated CE as a positive CE, (b) converting the resulting rule via the translation steps we have just outlined, and (c) seeding the resultant composite cursor with the shadow object. Figure 4.19 (Original)

5. Negated CE filters, like `N5_4()` have a simple realization in P2. The filter is (1) to test the container of the negated CE for any element that satisfies predicate P of the negated CE, and (2) to examine the corresponding shadow container if any element satisfies P and whose timestamp is greater than the dominant timestamp. If qualified elements are found in either container, `N5_4()` returns FALSE. Both qualifications can be easily expressed using cursors with the obvious selection predicates over the container and shadow container.

shows the result of applying (a) to the rule of Step 5; Figure 4.19 (Transformed) shows the result of applying (b) and (c). For perspicuity, we have underlined the relevant code.

Original

```
(p rule5
  (stage ^value detect_junctions)
  (edge ^p1 <bp> ^p2 <p2> ^joined false)
  (edge ^p1 <bp> ^p2 <p3> ^p2 <> <p2> ^joined false)
  (edge ^p1 <bp> ^p2 <> <p2> ^p2 <> <p3>)
  -(edge ^p1 <bp> ^p2 <> <p2> ^p2 <> <p3>)
  -->
```

Transformed

```
#define query5d \
  "$a.value == 'detect_junctions' \
   && $b.joined == 'false' \
   && $c.p1 == $b.p1 && $c.p2 != $b.p2 && $c.joined == 'false' \
   && $d.p1 == $b.p2 && $d.p2 != $b.p2 && $d.p2 != $c.p2
   && N5_4(&$b,&$c)"

#define temporal_query5d \
  "$a.ts <= dominant_timestamp \
   && $b.ts <= dominant_timestamp \
   && $c.ts <= dominant_timestamp \
   && $d.ts <= dominant_timestamp"

#define valid_query5d \
  "!deleted($a) \
   && !deleted($b) \
   && !deleted($c) \
   && !deleted($d) \
   && N5_4(&$b,&$c)"

typedef compcurs < a cont_stage, b cont_edge, c cont_edge, d shadow edge >
  given < d >
  where query5d && temporal_query5d
  valid valid_query5d
curs5_d;
```

Figure 4.19 Rule Translation Step 6:
Seeding of Shadow Elements.

4.2.4 Other Issues

There are additional issues regarding the translation of OPS5 rule sets into P2 programs that are worth mentioning. First, when an element is inserted in LEAPS, it is pushed onto a wait-list stack for subsequent seeding. Composite cursors, whose execution was suspended, are placed on a join-stack. The stack whose top element has the most recent timestamp is chosen to be the dominant object on the next execution cycle. In RL (and in news versions of LEAPS than OPS5.c and DATEX), the wait-list stack and join stack are uni-

fied. This gives a very compact and elegant representation of the primary cycle loop (see Figure 4.20). Note that the “unified” stack is represented as a container, and top is a cursor that references the top element of the stack.

The procedures for rule firings are also compact (see Figure 4.21). If a cursor has not yet been created (i.e., `fresh` is `FALSE`), one is allocated from the heap, initialized, and positioned on the seeding element. Control then falls to the `foreachk` statement. If a cursor has been created (and whose execution has been suspended), control continues at the end of the `foreach` statement (where validation tests are performed by P2). Once a tuple is generated, the rule is fired and the procedure is exited. After all tuples have been generated, control passes to the next rule for possible firing.

```
void execute_production_system (void)
{
  while(1) {
    // Get the top of the stack.
    reset_start(top);
    if (end_of_container(top))
      // The stack is empty. We're at a fix-point.
      break;
    else {
      // The stack is not empty.
      fresh = !top.curs;
      dominant_timestamp = top.time_stamp;
      (*top.current_rule)();
    }
  }
}
```

Figure 4.20 Execution cycle.

4.2.5 Notes on Data Structures

We stated earlier that elements are not updated, but rather deleted and then reinserted. While this seems odd, it actually plays an integral role in the design of the LEAPS data structures for containers. The basic idea is that composite cursors on the unified stack may point to elements that have been deleted. To advance a composite cursor in such situations, elements can only be logically deleted (i.e., flagged deleted); their storage space cannot be physically reclaimed. (Or more accurately, their storage space cannot be reclaimed until no cursors are referencing them). Hence the need for modeling updates as deletions followed by insertions.

```
void seed_rule14_a (void)
{
  curs14_a *c;

  if (fresh) {
    c = (curs14_a*) malloc(sizeof(curs14_a));
    top.curs = (void*) c;
    initk(*c);
    pos(c->a,top.cursor_position);
  }
  else {
    c = (curs14_a *) top.curs;
    goto valid_tests;
  }

  foreachk(*c) {
    fire_rule14_a( c );
    return;
valid_tests:
    // Perform valid tests here.
  }

  free(c);
  fresh = TRUE;
  top.current_rule = skip_rule;
  // Call next_rule procedure for next rule firing.
  next_rule();
}
```

Figure 4.21 Rule seeding procedure.

The LEAPS compilers performed rudimentary garbage collection, where the physical space of elements is reclaimed. We noted that maintaining reference counts (or whatever the LEAPS compilers actually do) adds considerable run-time overhead. For the applications of LEAPS that we have seen, garbage collection at fix-point time offers a much faster and simpler way to accomplish garbage collection.

Another unusual requirement for a LEAPS container data structure is that elements must be stored in descending timestamp order. One reason is to maintain OPS5 semantics. Another is to be consistent with the general expert-system philosophy that the tuple that is selected for rule firing should have the most recent timestamps. The simplest data structure that LEAPS could use as a container implementation is a doubly-linked list, where deleted elements are still "connected" in that cursors on deleted elements can be advanced to nondeleted elements. Figure 4.22 shows two cursors on a container (imple-

mented by a list). Cursor A points to an element with timestamp 3; cursor B points to a deleted element with timestamp 2. When cursor B is advanced, it will be positioned on the next undeleted element of the container whose timestamp is less than 2 (in this case, the element with timestamp 1). In general, it is possible that B may need to traverse a chain of deleted elements before the first undeleted element is reached.

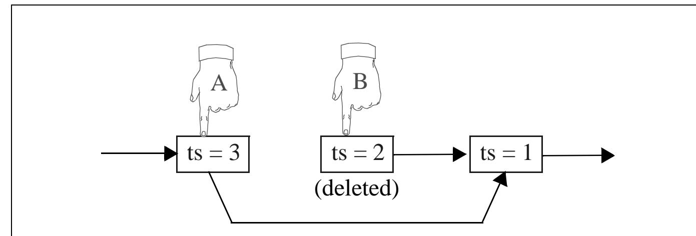

Figure 4.22 Container with deleted element.

4.2.6 Optimizations

The LEAPS compilers (and RL) include a variety of optimizations than enhance the basic algorithms outlined in Section 3. We explain the major optimizations in this section.

Timestamp Ordered Lists. A timestamp ordered list is a doubly-linked list where (undeleted) elements are maintained in descending timestamp order. Unlike "standard" doubly-linked lists, timestamp ordered lists perform query modifications for optimizations. For example, a typical rule selection predicate requires element timestamps to be less than or equal to the dominant timestamp. A timestamp ordered list would use this requirement to optimize the `reset_start` operation, which positions a cursor on the first record that satisfies the selection predicate. What happens is that the timestamp predicate is applied to the first elements of the list until an element qualifies. From that point on, there is no need for qualifying subsequent elements due to timestamp ordering. Thus, predicates applied to subsequent elements do not involve temporal qualification. Other types of query optimizations with timestamp ordered lists are possible; readers are encouraged to see the `tlist` component in the P2 library.

Predicate Indices. A predicate index is a list of elements of a container that satisfy a given predicate. (In the AI literature, predicate indices are called *alpha memories*). Predicate indices are quite useful in LEAPS/RL, as the selection predicates of rules are static. For example, to minimize the search time for finding stage elements whose `value` field is "`labeling`" (in `rule14` of Figures 4.13-4.16), a predicate index for stage using predicate "`$.value == 'labeling'`" is used. In general, a predicate index is created for each positive (and negative) condition element of a rule that references constants. Again, looking at `rule14` as an example, Figure 4.23 shows the predicate indices that would be created. A predicate index component in P2, `predindx`, is a minor modification of `tlist`.

Condition Element	Container	**Predicate to Index**
(stage ^value labeling)	stage	$.value == 'labeling'
(junction ^type tee ...)	junction	$.type == 'tee'
(edge ^p1 <bp> ^p2 <p1>)	edge	(none)
(edge ^p1 <bp> ^p2 <p3> ^label nil	edge	$.label = 'nil'

Figure 4.23 Predicate indices.

Active Rule Optimization. k-way joins can lead to $O(n^k)$ execution times, where n is the number of elements in a container. Eliminating costly searches that are known, a priori, not to yield tuples, often provides great performance advantages. The active rule optimization is the skipping of rules to be seeded because it is known that the rule cannot generate tuples. This optimization requires the presence of predicate indices. When a predicate index is an empty list (i.e., there are no elements in the container that satisfy the given selection predicate), we know that a seeded rule cannot produce tuples. It is a simple matter to augment the definition of the predicate index layer to accept as a further annotation two procedures. One procedure is called when the predicate index becomes empty; another procedure is called when the predicate index becomes nonempty. The procedures themselves merely increment a counter for each rule that uses the predicate index. If the counter for a rule is nonzero (meaning that there are one or more predicate indices that are null), we know that the rule can be skipped for seeding. If the count is zero, seeding the rule may produce tuples. In RL, we examine the counter for every rule that could be seeded by a dominant object. Thus, if there are n rules, there are n tests. In general, the

number of rules that are active at any one time is rather small. Thus, if the list of active rules is maintained dynamically, performance of LEAPS should be enhanced. In particular, we conjecture that as the number of rules per rule set increases, a scheme to maintain dynamically the list of rules may offer significant performance advantages.

Symbol Tables. String comparisons are always costly. A more efficient way to perform string comparisons is to enter strings into a symbol table and to compare handles to strings. Since OPS5 allows only == and != operations on strings, handle comparisons work well. This optimization is also called *string constant enumeration*.

Shadow Stacking. We explained earlier that the unified stack maintains a timestamp ordering of its elements; the top element has the most recent timestamp and the bottom element has the oldest. Deleted objects are called *shadows*. Experience has shown that shadows rarely succeed in seeding rules (i.e., producing tuples to fire). As there can be many shadows on the stack at any given moment, a large fraction of LEAPS run time is consumed processing shadows. A way to minimize the processing time for shadows is to place them at *bottom* of the stack, rather than at the top. This invalidates the property that the stack maintains elements in descending timestamp order, but has the advantage that shadow processing becomes more efficient (i.e., particularly in the presence of active rule optimizations). The increase in LEAPS performance can be dramatic with shadow stacking.

Hashed Timestamp Ordered Lists. The standard LEAPS data structure is a timestamp ordered list, described above. The standard LEAPS join algorithm is nested loops. A way to improve the performance of LEAPS dramatically is to use hashed timestamp ordered lists. The idea is simple: instead of maintaining a single list of timestamp ordered elements, b lists are maintained, one list per bucket. Bucket assignments of elements are based on a hash key, which is also a join key. As a result, search times for elements on inner loops of joins are reduced by a factor of b (i.e., a fraction of (b-1)/b of the elements have been eliminated as they don't hash to the right join key). Hashed timestamp ordered lists (`htlist`) is a simple variation on timestamp ordered lists (`tlist`); hashed timestamp ordered predicate indices (`hpredindx`) is a simple variation on predicate indices (`pred-`

indx). In general, the idea of using "hash" joins to obtain improved performance is an obvious consequence of the work of Brant and Miranker [Bra93a].

Negation Optimization. Experience has shown that the following is not an effective optimization, but it is an interesting idea never-the-less. Figure 4.24 recalls Figure 4.17 which helps illustrate the meaning of negation. A great deal of time is spent evaluating predicates of negated condition elements. In the case that a dominant object can seed multiple tuples (as in Figure 4.24 where object e seeds 4 tuples), it is possible to optimize the processing of predicates of negated CEs. The idea is simple: when the negated CE is applied to tuple at time t_2, the truth value of its predicate must be determined for the interval $[t_0, t_2]$. Note that the predicate must have been true for all previously tested intervals, e.g., $[t_0, t_1]$, since had the predicate failed, it would not be possible to seed further tuples. The optimization is to avoid replicate evaluation of a negated predicate over the same interval. To test the validity of a predicate in interval $[t_0, t_2]$, it is sufficient to test the validity only over the interval $[t_1, t_2]$, as the truth of the predicate in $[t_0, t_1]$ has already been established. Experience has shown that a dominant object typically seeds at most one tuple per rule. Consequently, the conditions for the above optimization don't seem to arise.

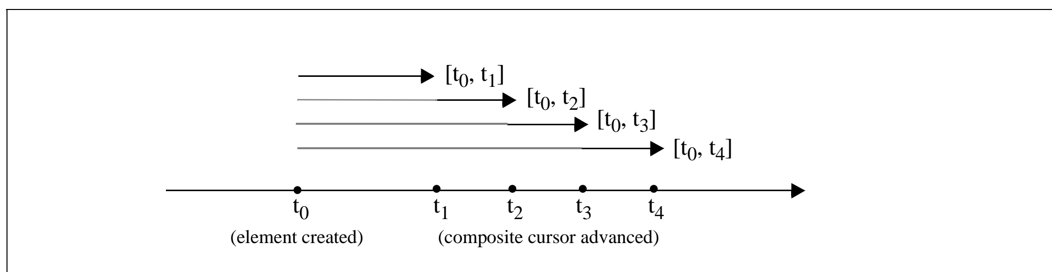

Figure 4.24 Negation Optimization.

Malloc optimization. The Unix malloc operation is very slow. The LEAPS compilers relied on their own memory allocation scheme. We tried to do something similar with a layer in P2 which performs the duties of malloc on our own, home-grown memory

allocation scheme. As it turns out, gnu malloc is more efficient, so we did not pursue malloc optimization further.

4.3 Results

The LEAPS algorithms are notoriously difficult to understand. In interviews with the OPS5.c development team, they felt that their expertise would enable them to rewrite a LEAPS compiler in 2-3 months, whereas novices (us) would take at least twice that long to code (e.g., 6 months). It did take us several months to comprehend the algorithms, but only took us two months to code RL.[6] As supporting evidence, RL is less than 5K lines of C, `lex`, and `yacc`. OPS5.c is four times larger–almost 20K lines: 10K for the basic compiler and another 10K for the run-time system included in all OPS5.c-produced executables. Thus *for the LEAPS application and LWDBs, using P2 reduced the development time and code size by a factor of three*.

We discovered two reasons for this. First, P2 offers substantial leverage in developing LWDBs and their applications. P2 currently consists of over 75K lines of code; it performs general optimizations that LEAPS experts had to hand-code into their compilers. Second, by far the most substantial productivity gain was using P2 data types to express the LEAPS algorithms. Although complicated, the LEAPS algorithms are elegant when expressed in P2. The P2 separation of LWDB implementation details from its client applications significantly reduced the complexity RL's development and the understanding, coding, and debugging of the LEAPS algorithms.

To help us evaluate the performance of RL/P2-generated programs, the LEAPS compilers development team provided us with OPS5 rule set benchmarks:

- `tripl` (3 rules that output 3-tuples of numbers ranked in descending order).
- `manners` (8 rules that find seating arrangements with constraints).

6. The un-optimized RL implementation of the LEAPS algorithms were coded in one week. P2 was being written at the time of our RL work; the remainder of the two months included the time spent waiting for P2 to be debugged and the time needed to add the myriad optimizations to RL that LEAPS uses.

- `waltz` (33 rules that define a 2-D line labeling program).
- `waltzdb` (38 rules that define a more complex version of `waltz`).

Each of these rule sets processed scalable input data sets; programs that generated these data sets were included with each rule set. The LEAPS compilers development team also provided us with two versions of LEAPS:

- OPS5.c (a version that generates programs whose databases are main-memory resident [Mir90-91])
- DATEX (a version that generates programs whose databases are disk-resident [Bra93a]).

DATEX databases are stored by Jupiter, the (heavyweight) Genesis file management system [Bat88]. Thus, OPS5.c and DATEX provided us with an ideal opportunity to evaluate the scalability of P2: we could compare P2-generated LWDBs with both hand-coded main-memory LWDBs and a heavyweight extensible disk-resident DBMS. We accomplished this using the same P2 programs generated by RL, but swapping type expressions.

Figure 4.25 shows the type expressions that we used to store RL databases. $\mathtt{RL}_1$($\mathtt{RL}_3$) differ from $\mathtt{RL}_2$($\mathtt{RL}_4$) only in the transient or persistent storage of containers. $\mathtt{RL}_3$($\mathtt{RL}_4$) differ from $\mathtt{RL}_1$($\mathtt{RL}_2$) only in the use of the hashed timestamp ordered lists optimization. All of the RL databases use the following optimizations: timestamp ordered lists, predicate indices, active rule optimization, symbol tables, and shadow stacking. $\mathtt{RL}_3$ and $\mathtt{RL}_4$ also uses the hashed timestamp ordered lists optimization.

```
#define RL(PREDINDX_LAYER, TLIST_LAYER, MEM_LAYER) \
  nloops[link2top[top2ds_qualify[PREDINDX_LAYER[ \
    TLIST_LAYER[delflag[malloc[MEM_LAYER]]]]]]]

typex {
  RL1 = RL(predindx, tlist, transient);
  RL2 = RL(predindx, tlist, mmap_persistent);
  RL3 = RL(hpredindx, htlist, transient);
  RL4 = RL(hpredindx, htlist, mmap_persistent);
}
```

Figure 4.25 `RL` type expressions.

Prior to benchmarking, it was our goal to have RL/P2 generated programs have a performance within 10% of the LEAPS compilers, OPS5.c and DATEX. We expected RL/

P2 to be slower than the LEAPS compilers, because (1) P2 is a general-purpose tool, whereas LEAPS was hand-coded by experts, and (2) we converted OPS5 programs to C programs in two translation steps, whereas the LEAPS compilers accomplished this in one step (see Figure 4.10).

Figures 4.26, 4.27, 4.28, and 4.29 show the performance results for the `tripl`, `manners`, `waltz`, and `waltzdb` benchmarks respectively. The results presented here were obtained on a SPARCstation 5 with 32 MB of RAM running `SunOS 4.1.3` using the `gcc2.5.8` compiler with the `-O2` option. Similar results have been obtained on other systems.

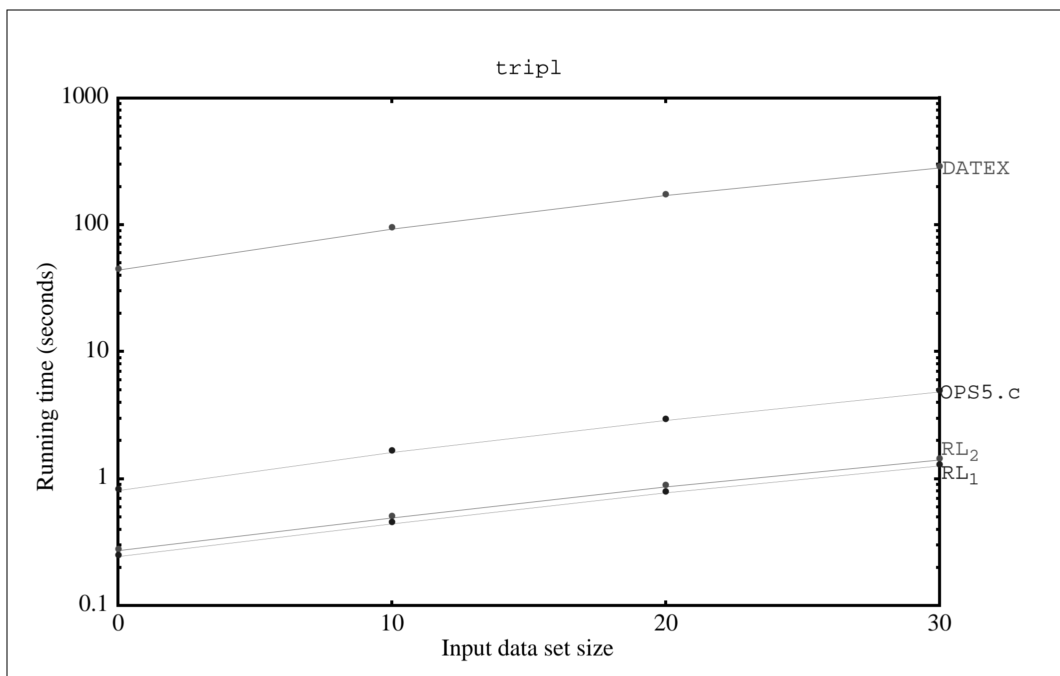

Figure 4.26 `tripl` running time vs. input data set size.

Let's consider first the performance of RL_1 and RL_2. In all cases, their performance exceeded that of OPS5.c and DATEX. RL_1 was typically *two times faster* than OPS5.c, while RL_2 was typically *fifty times faster* than DATEX.

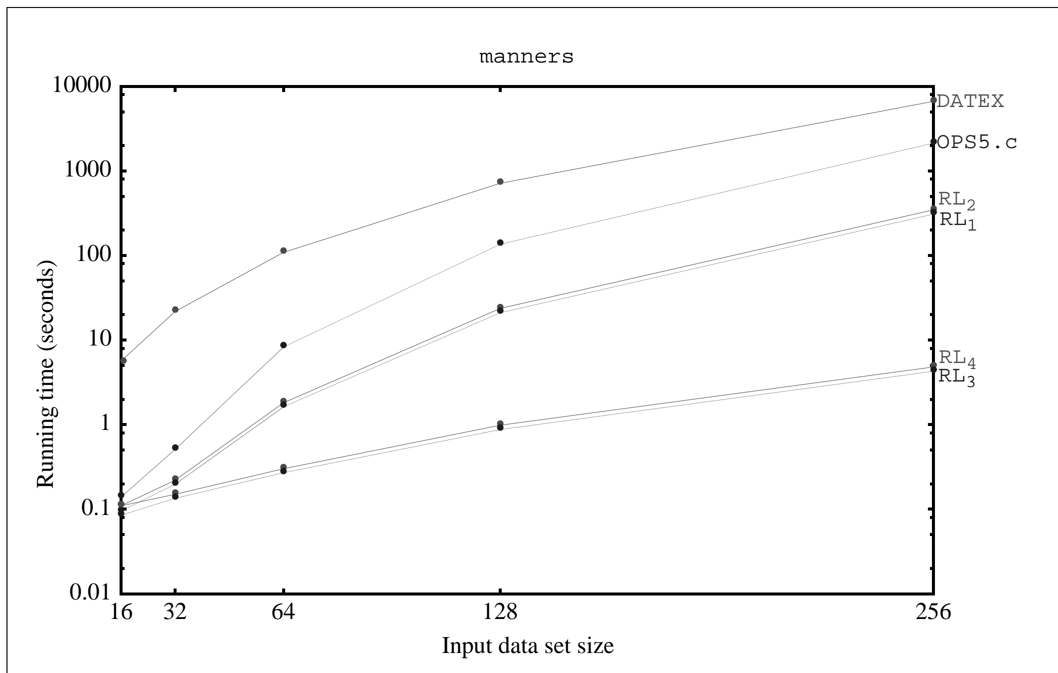

Figure 4.27 `manners` running time vs. input data set size.

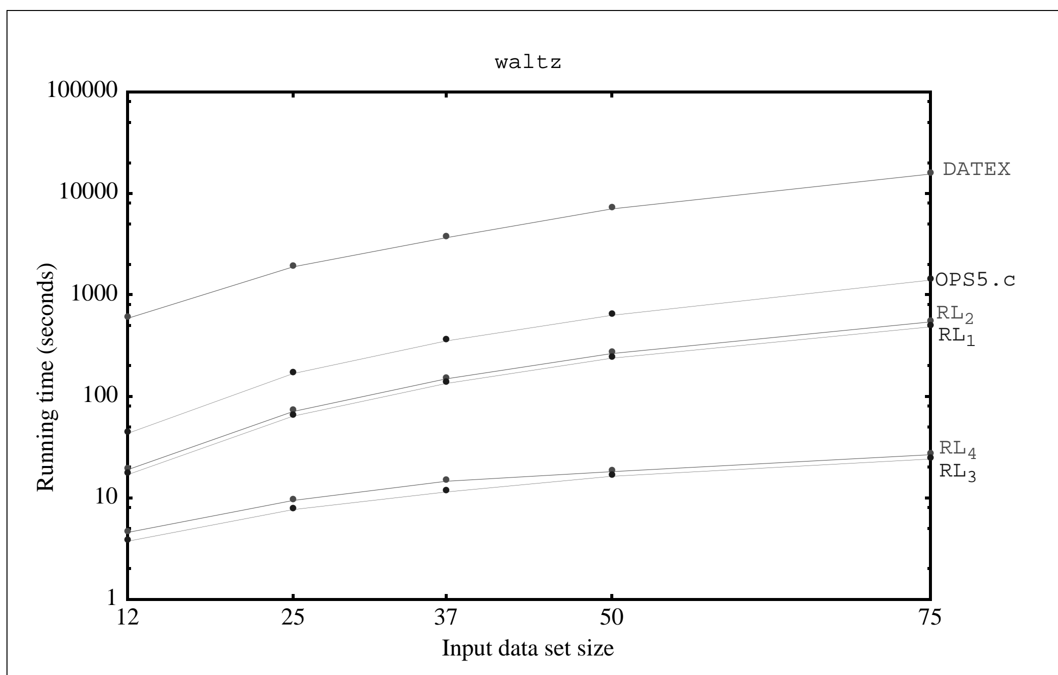

Figure 4.28 `waltz` running time vs. input data set size.

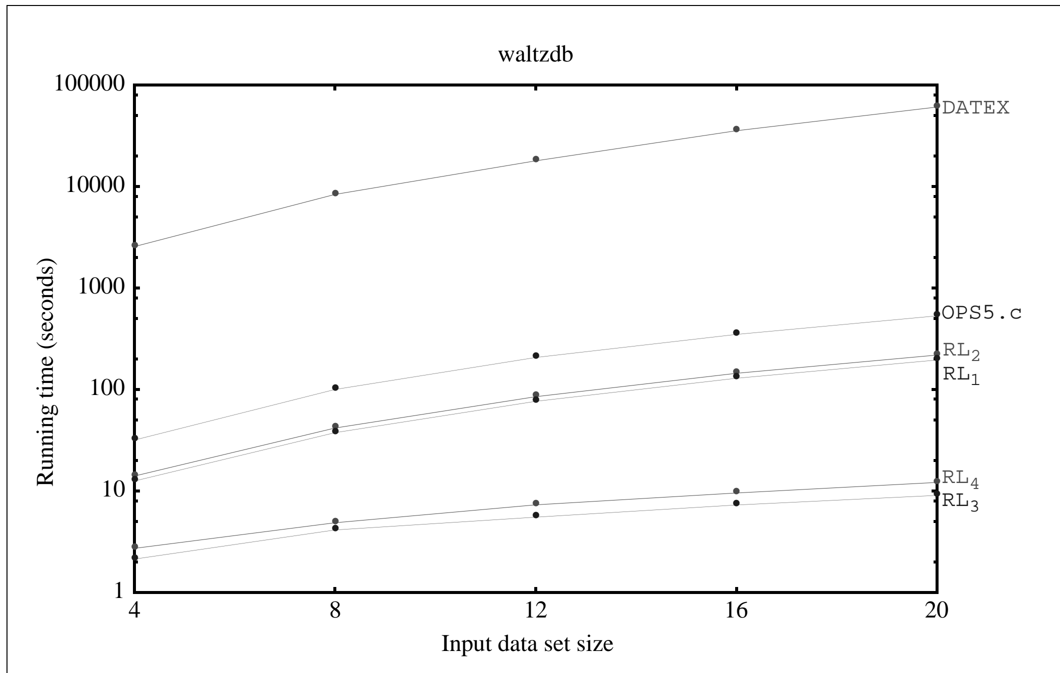

Figure 4.29 `waltzdb` running time vs. input data set size.

RL_1's improved performance over OPS5.c was due to several reasons. First, P2 generated code is more efficient than that of OPS5.c; P2 performs optimizations automatically that are difficult, if not impractical, to do by hand. Second, expressing the LEAPS algorithms in terms of P2 abstractions clearly revealed some simple optimizations that were otherwise obscured. Third, the design and implementation of OPS5.c was so complicated that it was necessary to replicate predicate indices for understandability; eliminating replicated indices was trivial in the RL/P2 version. Fourth, OPS5.c used a tagged type system and performed dynamic garbage collection unnecessarily. (This was an example of a LWDB being designed to meet perceived needs that never arose.) It was through our experiments with RL/P2 that the LEAPS compilers implementors learned that garbage collection was unnecessary.

RL_2's improved performance over DATEX was due in part to the reasons cited over OPS5.c, but by far the most substantial gains came from eliminating the large overheads of heavyweight (extensible) DBMS construction. These included: layered software

designs, interpretive execution of queries, buffer management, and general-purpose storage structures that were not as efficient as the LEAPS-optimized storage structures of OPS5.c. These overheads caused DATEX to be slower than OPS5.c by more than an order of magnitude, whereas swapping the `transient` component in $\mathtt{RL}_1$ with `mmap_persistent` to produce $\mathtt{RL}_2$ (i.e., replacing transient memory with memory-mapped I/O) reduced performance by only a few percent.

When we swapped $\mathtt{RL}_1$($\mathtt{RL}_2$) with $\mathtt{RL}_3$($\mathtt{RL}_4$) (i.e., when we used hashed structures instead of non-hashed), we observed an astounding performance improvement for three of the rule sets. For `manners`, `waltz`, and `waltzdb`, $\mathtt{RL}_3$ executed over *an order of magnitude faster* than OPS5.c and $\mathtt{RL}_1$; $\mathtt{RL}_4$ was *three orders of magnitude faster* than DATEX. The reason is simple: nested loops is an inefficient join algorithm; by emulating hash joins, we obtained big improvements in performance. We did not achieve speedups for `tripl`, as `tripl` has only inequality-joins and thus hash-joins could not improve its performance.

We can make three important observations here. First, experimenting with very different LWDB implementations was effortless: all we needed to do was to alter type expressions and recompile. Second, when needed components were absent from the P2 library (as was the case for `htlist` and `hpredindx`), it took us only a few days to write them. We were able to reuse other components of the P2 library to minimize our coding efforts. In contrast, DATEX was a full rewrite of OPS5.c and took many months to complete. Third, P2 provides a technology by which customized LWDBs (i.e., customized type expressions) can be generated *per rule set* to maximize performance; this capability is impossible with standard LWDB implementation techniques, including those used by LEAPS.

Chapter 5

Smallbase

We used P2 to reengineer the Smallbase LWDB [Hew96] in order to demonstrate that GenVoca can scale to generate complex systems, while at the same time maintaining good generated code performance and high programmer productivity.

Smallbase is a meaningful basis for comparison because it shares many design goals with P2 but differs mainly in construction methodology. Smallbase [Hey95] is a modern LWDB from HP labs. Like P2, the primary design goal of Smallbase is high performance: an order of magnitude performance improvement over traditional DBMSs for transaction processing queries and even more for decision support queries. Unlike P2, Smallbase has been hand-crafted and optimized as a single, monolithic system.

Our goal in these experiments was to reengineer Smallbase as accurately as possible within the framework of P2. We did *not* improve upon the Smallbase algorithms and data structures, as we had done with LEAPS. Rather, we eliminated differences, in order to isolate the effects of construction methodology on generated code performance and programmer productivity.

In order to reproduce Smallbase exactly, we would have to start with the complete Smallbase source code. Unfortunately, this was not possible, both because HP corporate policy forbid the release of the Smallbase source code, and because, when we began our experiment, Smallbase was not yet finished. Thus, we begin with Smallbase design documents and code snippets for only the most important features (e.g., latches). Where the Smallbase design documents were unclear and code snippets were unavailable, we

appealed to freely available DBMS code. We drew, in particular, from [Gra93b, Cla86, Exo94].

Despite the unavailability of the complete source code, we believe we have adequately reproduced the functionality of Smallbase in P2. This belief is reinforced by two observations. First, the implementations of various DBMSs are strikingly similar. In particular, [Gra93b, Cla86, Exo94] have more similarities than differences. This is typical of a mature, well understood domain–the type of domain that is amenable to GenVoca generation [Bat92a]. Second, the finished Smallbase[1] and P2 systems have remarkably similar performance.

5.1 Decomposition

Smallbase consists of a single, monolithic, pre-compiled executable. Smallbase user programs are compiled and linked with this executable. All Smallbase functionality is embedded into the executable, and users extract the behavior they require via run-time arguments. For example, when they open a connection to a database, Smallbase users can opt for single (a.k.a., private) or multiple (a.k.a., shared) user process support; transient (a.k.a., main memory) or persistent (a.k.a., disk) storage of data; logging or no logging; and if logging is selected, synchronous or asynchronous flushing of the log from memory to disk. Thus, Smallbase encapsulates a family of related systems in a single executable.

The primary challenge in reengineering Smallbase was decomposing it into GenVoca components. We chose a decomposition based on resource managers. A *resource manager* is a subsystem (e.g., log manager or lock manager) of a DBMS that encapsulates the data and code that provide access to some shared object (e.g., the log or the lock table); a collection of resource managers *together* provide the ACID properties of transactions

1. Note that HP Labs spun-off the Smallbase project before it was fully completed. For example, the final version of Smallbase provides transactional isolation only by locking the entire database for the duration of the transaction, thus serializing all transactions, whereas the original design for Smallbase included support for finer granularity concurrency control [Hey95]. Since P2 was intended to reengineer this original design, P2 provides several features that were not necessary for the experiments comparing Smallbase and P2 (see Section 5.4). For example, P2 supports fine grain concurrency control (see Section 5.3.2).

[Gra93b]. In particular, we chose a decomposition of one resource manager per component; this decomposition is the most obvious and natural, and it is appropriate for both users and component implementors.[2]

A complete DBMS is specified using the type expression *pattern* shown in Figure 5.1. Note that this is a pattern rather than an actual type expression, because protocol, log, and process represent parameters that may be instantiated by any member of a family of components. A *family* of components is a subset of components from a particular realm. For example, the retrieval family consists of the subset of DS realm components that are capable of processing queries (e.g., bintree, dlist, hash). The *lock protocol manager* family consists of the XACT realm components protocol_coarse, protocol_fine, protocol_xact_mutex, and protocol_smallbase (see Section 5.3.2). The *log manager* family consists of the XACT realm components log, log_async, and log_sync (see Section 5.3.4). The *process manager* family consists of the PROCESS realm components process_uniprocess, process_unix, process_thread, process_pthread, and process_smallbase (see Section 5.2.1).[3]

Any component in the family may be used to instantiate the parameters in the type expression pattern. For example, Figure 5.2 shows an example actual DBMS type expres-

---

2. This encapsulation is appropriate for *users*, so they can easily comprehend and use type expressions. A *smaller* granule would force users to include several components in a type expression to specify a single high-level function (e.g., logging or locking). This would be inconvenient and subject to error. A *larger* granule (e.g., a combined logging plus locking component) would force users to include unnecessary functionality. This, in turn, would necessitate additional runtime overhead and/or additional mechanisms (e.g., annotations or compile-time arguments) to select the desired sub-functionality. This encapsulation is also appropriate for *component implementors*, because it maximizes decomposition while minimizing dependencies between components. Resource managers are, unfortunately, not completely self contained. For example, at restart, the transaction manager gets its anchor from the log, which naturally, is maintained by the log manager. A *smaller* granule would increase decomposition, but would introduce more such dependencies. A *larger* granule would reduce these dependencies, but reduce decomposition. One resource manager per component appears to yield the best compromise between decomposition and dependencies.
3. We use the Courier typeface to distinguish families of components from individual components (e.g., the log manager family of components includes the log component); underlining to distinguish the parameters in the pattern from actual components (e.g., the process parameter may be instantiated by the process component); and uppercase to distinguish realms from families of components and individual components (e.g., the XACT realm includes the transaction manager family of components which includes the xact component)

```
dbms = trace[protocol[log[xact[lock[
  op_vec[xact2process[process[
    process2link[LINK]]]]]]]]]
```

Figure 5.1 Pattern for DBMS type expressions.

sion, instantiating protocol using protocol_fine, log using log_sync, process using process_unix, and the LINK realm parameter using link2top[top2ds_qualify[hash[array[transient]]]]. Since all containers in the schema share the same type expression, they must use the same instantiations. In particular, they share a single log and lock table.

```
dbms = trace[protocol_fine[log_sync[xact[lock[
  op_vec[xact2process[process_unix[
    process2link[link2top[top2ds_qualify[hash[array[transient]]]]]]]]]]]]
```

Figure 5.2 Example DBMS type expression.

This decomposition required us to add the two new realms, PROCESS and XACT, shown in Figure 5.3. PROCESS realm components provide primitive, low-level operating system specific process and thread synchronization and control (see Section 5.2). XACT realm components are high-level resource managers (see Section 5.3). The process2link layer transmits PROCESS operations as is; if it receives any PROCESS operations, this means that no layer above it could process the operation; the xact2process layer provides analogous semantics for XACT and PROCESS operations. These are fatal errors and should not occur; thus process2link and xact2process are safety nets.

A higher performance database can be generated by customizing the type expression: either by specifying a more efficient member of a family of components, or by omitting an unnecessary component. Ideally, a user could independently include or omit *any* component from Figure 5.1. We were not able to achieve this level of independence. But, by careful design of components and interfaces, we were able to substantially reduce dependencies to those shown in Figure 5.4.

```
LINK : process2link[PROCESS] // Treat PROCESS operations as errors,
                             //   translate LINK operations as is.

PROCESS = {
  process[LINK]       // Process and thread synchronization and control.
  xact2process[XACT] // Treat XACT operations as errors,
                      //   translate PROCESS operations as is.
}

XACT = {
  trace[XACT],        // For debugging, print operations executed.
  protocol[XACT],     // Lock protocol for transactional isolation.
  log[XACT],          // Manage the log.
  xact[XACT],         // Manage transaction state and transaction identifiers
  lock[XACT],         // Manage the lock table.
  op_vec[PROCESS]     // Manage operation vectors.
}
```

Figure 5.3 XACT and PROCESS realms.

Manager	**Design Rule**
lock	Requires a process manager.
log	Requires generic_init (or conceptual). Requires the *xact* manager. Must appear above the *xact* manager.
process	A protocol manager is required by process_smallbase.
protocol	The lock manager is required by protocol_coarse, protocol_fine, and protocol_xact_mutex.
trace	The trace and process managers must appear above the log manager.
xact	Requires a process manager.

Figure 5.4 Resource manager design rules.

5.2 The PROCESS Realm

Several resource managers require the ability to synchronize and control execution contexts. For example, the lock and transaction managers need mutual exclusion on their internal data structures, the lock manager needs to explicitly schedule processes, and the protocol_smallbase protocol manager needs to regulate concurrent access to shared data. This functionality is not provided by the TOP, DS, MEM, or LINK realms. P2 uses processes, semaphores, and conditions to *abstract* this functionality, and the PROCESS realm to implement operations on them.

In P2, the **process** data type is an abstract model of an *execution context*; that is, some code and private data; it may be implemented by various mechanisms, for example, Unix processes or POSIX threads. The **semaphore** data type is a mechanism used to

implement mutual exclusion between processes; it may be implemented, for example, by System V IPC semaphores or POSIX semaphores. The **condition** data type is a mechanism used for communication between processes; it may be implemented, for example, by Unix signals or POSIX condition variables. Processes, semaphores, and conditions all have unique names called *identifiers*.

The interface exported by the process manager components is called the PROCESS realm. This realm extends the LINK realm with the additional operations shown in Figure 5.5

Operation	Semantics
int delete_process(void)	Finalize and delete process manager.
int exit_process(int i)	Terminate current process.
process get_process_id(void)	Return process identifier of current process.
int fork_process(**process** *p, void *f(), void *arg)	Create new process with start function f(arg). Return process identifier of new process in p.
int init_process(void)	Create and initialize process manager.
int join_process(**process** p)	Wait for process p to terminate.
int init_semaphore(**semaphore** *s, unsigned c)	Create new semaphore with initial value c. Return semaphore identifier of new semaphore in s. Some process managers (i.e., process_uniprocess, process_unix, and process_smallbase) permit c to be any value greater than or equal to 0 (counting semaphore), while other process managers (i.e., process_pthread and process_thread) restrict c to 0 or 1 (i.e., binary semaphore or mutex).
int lock_semaphore(**semaphore** *s)	P(s): if semaphore s is greater than 0, decrement it by 1. Otherwise wait until s becomes greater than 0.
int trylock_semaphore(**semaphore** *s)	If semaphore s is greater than 0, decrement it by 1. Otherwise, return error.
int unlock_semaphore(**semaphore** *s)	V(s): increment semaphore s by 1.
int delete_semaphore(**semaphore** *s)	Delete semaphore s.
int sleep_process(**condition** *c, **semaphore** *s, unsigned *t)	Suspend current process. Return new condition identifier in c. If condition is not signaled within t seconds, return an error.
int wakeup_process(**condition** *c)	Signal condition c in order to resume process sleeping on condition.

Figure 5.5 PROCESS realm operations.

As described in Chapter 2, construction of a GenVoca generator for a new realm requires a careful domain analysis, which is only possible for well understood domains. We did not fully understand the process domain when we began the implementation of P2. This is clear from the fact that the PROCESS realm interface has changed several times in

the course of out implementation. In particular, when we began, the interface had much more the flavor of Unix processes (e.g., the **condition** data type did not exist, and the sleep_process() operation was based on **process**, and had nearly the same interface as the Unix sleep() operation). Yet, we had to modify the interface substantially to also support threads, and now the interface has much more the flavor of POSIX threads (e.g., the wakeup_process() operation is now based on **condition**, and has nearly the same interface as the POSIX pthread_cond_timedwait() operation). The interface has now evolved to the point where it should support additional process and thread models without modification. For example, although we have not yet implemented such components, we speculate that the interface should now support remote procedure call (a.k.a., RPC) as well as non-Unix process and thread models (e.g., Microsoft Windows Win32 API processes and threads [Ric95]).

5.2.1 Process Manager

The process manager abstracts away the details of various process and thread implementations, and allows any of a set of plug-compatible components to provide the underlying implementation. Each of the components in the process manager family implements a different process, semaphore, and condition mechanism, as shown in Figure 5.6. These components are the only components that *export* the PROCESS realm; xact2process is the only component that *imports* it.

Component	Semantics
PROCESS process_uniprocess [LINK]	Single process, no semaphores or conditions. Procedures degenerate to constants or no-ops.
PROCESS process_unix [LINK]	Unix processes, System V IPC semaphores, Unix signals.
PROCESS process_thread [LINK]	Unix (e.g., UnixWare 2.x and Solaris 2.x) threads, semaphores, and condition variables.
PROCESS process_pthread [LINK]	POSIX threads, semaphores, and condition variables.
PROCESS process_smallbase [LINK]	Unix processes, Smallbase latches, Unix signals.

Figure 5.6 Process manager components.

The process manager components must consistently refine **process**, **semaphore**, and **condition** data types *simultaneously.* Thus, their implementations must be

compatible. For example, a component could not implement Unix processes with POSIX semaphores.

The `process_unix` component, for example, implements standard Unix processes, System V interprocess communication (IPC) semaphores, and Unix signals [Ste90]. The **process** and **condition** data types are refined to `pid_t` (defined in `sys/types.h`) and **semaphore** is refined to `sembuf` (defined in `sys/sem.h`). For example, with these refinements, the `fork_process()` operation calls `fork()`, `lock_semaphore()` calls `semctl()`, and `wakeup_process(c)` calls `kill(c, SIGALRM)`.

The `process_pthread` component, for comparison, implements POSIX threads, semaphores, and condition variables [But97]. The **process** data type is refined to `pthread_t` (defined in `pthread.h`), **semaphore** to `sem_t` (defined in `semaphore.h`), and **condition** to `pthread_cont_t` (defined in `pthread.h`). For example, with these refinements, the `fork_process()` operation calls `pthread_create()`, `lock_semaphore()` calls `sem_wait()`, and `wakeup_process()` calls `pthread_cond_signal()`.

5.3 The XACT Realm

The resource managers require functionality not provided by the TOP, DS, MEM, LINK, or PROCESS realms. This functionality is implemented by the XACT realm, and abstracted using **schema** and the following new data types: transactions, locks, and log sequence numbers.

In P2, the transaction or **xact** data type abstracts an ACID unit of work [Gra93b]. It has a unique name called an *identifier*, and a *status* (e.g., active, committing, aborting). The **lock** data type abstracts transaction isolation (the "I" in ACID). A lock has a unique *name*, a degree of sharing called a *mode*, a duration called a *class*, and possibly an error *status* (e.g., granted, waiting, denied). The log sequence number or **lsn** data type represents the location of a particular element within the log.

The interface exported by resource manager components is called the XACT realm. This realm extends the PROCESS realm with the additional operations shown in Figure 5.7.

5.3.1 Lock Manager

The lock manager provides a generic, high-level, operating system independent mechanism to regulate concurrent access to objects. It schedules processes explicitly using the PROCESS realm provided by the process manager (see Section 5.2).[4] The lock manager consists of the component shown in Figure 5.8.

The design and implementation of the lock manager is based directly on [Gra93b], with reference to [Cla86], and [Exo94] to fill-in some missing details. We have diverged from the lock manager in [Gra93b] primarily in lock naming, granularity of parallelism on the internal lock manager data structures, and deadlock detection.[5]

The lock manager provides the standard set of intention locks shown in Figure 5.9, and lock classes (a.k.a., durations) shown in Figure 5.10.

4. Thus we have the design rule that the lock manager requires the process manager (see Figure 5.4).

The lock manager uses two containers internally: a container of lock requests and a container of lock headers. These containers may have different implementations. Thus, in addition to the XACT realm parameter for the DBMS component beneath it, the lock manager component takes two DS realm parameters to specify the implementation of these containers. Their declarations are shown in Figure 5.11.

The lock manager's first internal container, the lock *header* container, represents all lock names; that is, the (static) set of all possible locks. Since this set is too large to represent explicitly, the container represents explicitly only the (dynamic) set of busy locks; free locks are represented implicitly by their absence. Thus, an element is inserted in the container when a lock becomes busy, and deleted when the lock subsequently becomes free. The lock manager assumes that the space of all possible locks is sparse with respect to the number of locks that are busy at any given time. Thus, we can allocate the container as a hashed array that supports deletion. That is, by the type expression `hash[avail[array[DS]]]`.[6]

---

5. Although [Gra93b] allocates for lock names a structure consisting of a 2 byte integer together with a 14 byte string, we use only a standard size unsigned integer (4 bytes on most machines). We chose this lock naming for simplicity, performance, and because we store the lock names in a P2 container and P2 does not fully support non-standard size integers; if it did, we would have chosen long unsigned integers (8 bytes on most machines) as a better compromise. Although [Gra93b] uses a fine grained parallelism, we use a single exclusive semaphore to protect all of the internal lock manager data structures. We chose this coarser granularity for performance and ease of implementation. Debugging, in particular, is greatly simplified by this coarser granularity. Although [Gra93b] provides a simple local deadlock detector, we rely entirely upon timeouts for deadlock detection.

6. We have made the optimization in the lock header type expression of using the `hash_array` component instead of the type expression `hash[avail[array[DS]]]`. The `hash_array` component has the advantage of handling both deletions and key retrievals very efficiently, but the disadvantage that it must be able to hash its keys perfectly, which is not possible for lock names. The solution to this problem is *not* to use the `hash_array_overwrite` component, because it overwrites the old lock header with a new one, and thus loses the old lock. Instead, the solution to this problem *is* to hash the lock names *before* `hash_array` ever sees them. Since they are pre-hashed, `hash_array` can now very easily hash its keys perfectly (e.g., using the identity function). Unfortunately, now `hash_array` treats the two different lock names that pre-hash to the same value as if they were the same lock name. In the uncommon case when this *does* happen, the lock manager still functions correctly, but possibly with some unnecessary spurious waiting, and a concomitant decrease in throughput and perhaps accidental deadlock. In the common case when this does *not* happen, latency decreases, with a concomitant increase in system throughput. In general, the overall effect should be increased system throughput.

Operation	Semantics
`void abort_xact(void)`	Terminate transaction and undo its state changes.
`void begin_xact(void)`	Initiate transaction.
`void checkpoint_schema(`**`schema`**`)`	Write copy of schema state to log.
`void close_schema(void)`	Finalize and delete connection to schema.
`void commit_xact(void)`	Make transaction's state changes public and durable.
`lsn` `get_max_lsn(void)`	Return maximum log sequence number used by current transaction. This is necessary for `UNDO`, which begins at maximum log sequence number and works backwards.
`OP_VEC get_op_vec(cursor)`	Return operation vector of given cursor.
`OP_VEC get_op_name_vec(cursor)`	Return operation name vector of given cursor.
`XACT_ID get_xact_id(void)`	Get transaction identifier of current transaction.
`XACT_STATUS get_xact_status(void)`	Return status of current transaction.
`void init_schema(void)`	Initialize schema.

`LOCK_REPLY`
`lock(LOCK_NAME name,`
`LOCK_MODE mode,`
`LOCK_CLASS class,`
`unsigned timeout)`
	Attempt to acquire lock `name` in `mode` with duration `class`. If caller successfully acquires lock within timeout seconds, return `LOCK_OK`. If timeout expires, return `LOCK_TIMEOUT`. If lock request container overflows, return `LOCK_REQUEST_OVERFLOW`. If lock header container overflows, return `LOCK_HEADER_OVERFLOW`.

`lsn` `log_insert(LOG_STRUCT *x,`
`unsigned fixed_size,`
`int narg, ...)`
 	
Where variable argument list consists of `narg` of the following pairs:
`char *v,`
`unsigned variable_size`
	Allocate a new log record and copy given data into it. Copy `fixed_size` bytes from `x`, and for each of `narg` (`v`, `variable_size`) pairs in variable argument list, copy `variable_size` bytes from `v`.
	The `narg` (`v`, `variable_size`) pairs in argument list are optimization used, for example, by `insert()`, `delete()`, and `upd()` to reduce unnecessary copying. Without this optimization, these operations would have to pass the entire log record in `x`, which would require copying the data twice: once from its original location into `x`, and once from `x` into the log.

`LOG_STRUCT *log_read_lsn(`**`lsn`**`)`	Return log record with given log sequence number.
`lsn` `log_read_anchor(void)`	Return log sequence number of most recent checkpoint.
`lsn` `log_transaction(`**`lsn`**`)`	Inform transaction manager that new log record has been created. Return log sequence number of previous log record. Necessary because log records are created by log manager, and log sequence numbers (e.g., maximum log sequence number used by each transaction and by DBMS as a whole) are maintained by transaction manager.
`void log_write_anchor(`**`lsn`**`)`	Store given log sequence number as that of most recent checkpoint.
`void open_schema(void)`	Create connection to and initialize schema.
`void put_op_vec(void)`	`Store schema operation vector.`
`void set_xact_status(XACT_STATUS)`	Set status of current transaction.
`LOCK_REPLY unlock(LOCK_NAME name)`	Unlock lock `name`. Return `LOCK_OK`.
`LOCK_REPLY` `unlock_class(LOCK_CLASS class)`	Release all locks of duration less than or equal to `class`. Return `LOCK_OK`. This procedure is called, for example, at end of transaction to release all locks of duration less than or equal to `LOCK_LONG`.
`void warm_restart(void)`	Start DBMS from persistent memory.

Figure 5.7 XACT realm operations.

Component	Semantics
`lock [op_vec : XACT, request : DS, header : DS] : XACT`	The lock manager.

Figure 5.8 Lock manager component.

Mode	Semantics
IS	Intent to request shared
IX	Intent to request exclusive.
S	Shared (read).
SIX	Shared, intent to request exclusive.
U	Update (read followed by write).
X	Exclusive (write)

Figure 5.9 Lock modes.

Class	Semantics
INSTANT	Combined lock/unlock operation.
SHORT	Released at the end of operation.
MEDIUM	Released explicitly.
LONG	Held to end of transaction.
VERY_LONG	Held across transaction boundaries.

Figure 5.10 Lock classes.

```
// Type expressions.
typex {
  request_typex = top2ds_qualify[slist_queue[avail[array[mmap_shared]]]];
  head_typex = top2ds_qualify[hash_array[mmap_shared]];
}

// Lock request.
typedef struct {
LOCK_NAME    name;            // The name of this lock.
LOCK_STATUS  status;          // Granted, waiting, converting, denied.
LOCK_MODE    mode;            // Mode requested (and granted).
LOCK_MODE    convert_mode;    // If in convert wait, mode desired.
int          count;           // The number of times lock was locked.
LOCK_CLASS   class;           // Lock class (a.k.a., duration).
condition    condition_id;    // Condition to signal when lock granted.
XACT_ID      xact_id;         // Transaction identifier.
} REQUEST;

container <REQUEST> stored_as request_typex with {
  array size is MAX_REQUEST;
  mmap_shared file is "/tmp/lock-manager-data" with size MMAP_SHARED_SIZE;
} request_cont

// Lock header.
typedef struct {
LOCK_NAME    name;            // The name of this lock.
LOCK_MODE    granted_mode;    // The mode of the granted group.
BOOLEAN      waiting;         // Flag indicates nonempty wait group.
} HEAD

container <HEAD> stored_as head_typex with {
  hash_array key is name with size MAX_HEAD;
  mmap_shared file is "/tmp/lock-manager-data" with size MMAP_SHARED_SIZE;
} head_cont
```

Figure 5.11 Lock request container and lock header container.

The lock manager's second internal container, the lock *request* container, represents the (dynamic) set of locks that are not free. That is, this container is a list of requests that are either waiting for or that have been granted locks. Under normal circumstances, an element is inserted in this container by a call to `lock()` and deleted by a subsequent call to `unlock()` or `unlock_class()`. Of course, there are many exceptions. For instance, instead of inserting a new element in the container, a call to `lock()` may increment the `count` field of an element previously added by the same transaction; or instead of being deleted by a call to `unlock()` or `unlock_class()`, an element may be deleted from the container if the timeout expires before the request is granted. The only way to fully appreciate the complexity of the lock manager (as well as the elegance of its expression in P2) is to examine the code itself (see Appendix).

5.3.2 Lock Protocol Manager

The lock protocol manager automatically adds to user programs calls to the lock manager to regulate concurrent access to data, providing transactional isolation. The specific calls added by the lock protocol manager are a function of (1) the protocol implemented by the particular lock protocol manager component and (2) the pattern of data accesses in the user program. The lock protocol manager consists of a family of components, each implementing a different protocol, as shown in Figure 5.12.

All of the current lock protocol manager components use strict two-phase locking. In the `protocol_coarse` and `protocol_fine` components, the growing phase consists of acquiring zero or more locks in `X` and `S` mode during the transaction; the shrinking phase consists of a call to `unlock_class()` at the end of the transaction, which releases all of these locks. The `protocol_xact_mutex` and `protocol_smallbase` components use exactly one `X` mode lock per transaction. In these components, the growing phase consists of acquiring a single lock at the beginning of the transaction, and the shrinking phase consists of releasing this lock at the end of the transaction. All of the current lock protocol manager components provide *true isolation* (a.k.a., *degree 3 consistency,* or *repeatable*

Component	Semantics
`protocol_coarse [XACT] : XACT`	Lazily acquire locks as necessary for individual operations that access and/or modify tuples. Use locks of the *largest* possible granularity: always the entire schema. Thus, for the `insert()` operation, lock the entire schema in mode `X` and class `LONG`; and for the `ref()` operation, lock the entire schema in mode `S` and class `LONG`. Release all locks at the end of the transaction. That is, for the `commit_xact()` and `abort_xact()` operations, call `unlock_class(LONG)`.
`protocol_fine [XACT] : XACT`	Like `protocol_coarse`, but use locks of the *smallest* possible granularity: container or element, depending on the particular operation. Thus, for the `insert()` operation lock the container in mode `X`; and for the `ref()` operation, lock the element in mode `S`.
`protocol_xact_mutex [XACT] : XACT`	Eagerly acquire a lock on the entire schema at the beginning of the transaction. Always lock the schema in mode `X` and class `MEDIUM`. Release the lock at the end of the transaction. That is, for the `commit_xact()` and `abort_xact()` operations, call `unlock()`. Thus, this component serializes all transactions.
`protocol_smallbase [XACT] : XACT`	Like `protocol_xact_mutex`, but use process manager semaphores, rather than lock manager locks.

Figure 5.12 Lock protocol manager components.

reads). In particular, they do not allow dirty reads, and thus do not have the problem of *phantoms* [Gra93b].

Only the lock protocol manager components call the lock manager directly. Thus, it was tempting to eliminate the user-visible lock manager component, and encapsulate its functionality in the lock protocol manager components. Doing so would have had the advantage of increasing encapsulation. We chose, however, to make the lock manager a separate component, because doing so yields a generally more elegant decomposition:

- Simplifies unit testing of the lock manager. Making it a separate component allows our regression tests to bypass the lock protocol manager to test it directly without excess baggage. Ease of testing is especially important for lock managers, which are famously difficult to debug.[7]
- Simplifies the implementation of future lock managers, or lock protocol managers. Future lock managers can provide more features (e.g., lock modes, queueing disciplines, and guarantees of service) or better performance; they could implemented

7. “Even well-designed lock managers have some of the most difficult bugs. If the history of lock managers has one lesson, it is this: *don’t get stuck maintaining a lock manager, and if you do, take out all the optimizations and fancy features as a first step*” [Gra93b, p. 485].

either as components or embedded directly in user programs. Future lock protocol managers could be embedded in user programs and call the lock manager directly, in a manner analogous to our regression tests that call the lock manager directly.[8]

- Makes explicit in the type expression the fact that not all protocol managers call the lock manager. For example, the `protocol_smallbase` component, uses process manager semaphores directly, rather than going through the lock manager.

5.3.3 Transaction Manager

Combined with the log manager, the transaction manager provides transactional atomicity, consistency, and durability (the "A", "C", and "D" in ACID). In particular, the transaction manager keeps track of transaction state (i.e., DO, UNDO, or REDO) and maintains transaction identifiers. The basic design and implementation of the transaction manager is based on [Gra93b], but we have made numerous simplifications.[9] The transaction manager consists of the component shown in Figure 5.13.

Component	Semantics
`xact [lock : XACT, xact_cb : DS] : XACT`	The transaction manager.

Figure 5.13 Transaction manager component.

The *transaction anchor* maintains the state of the transaction manager, in particular, the next transaction identifier. The anchor and some per-transaction data are shared by

8. Note that some of the lock modes (see Figure 5.9) and classes (see Figure 5.10) are not used by any of the current lock protocol managers. Nevertheless, we have provided these modes (1) because they are part of standard lock managers [Gra93b] and (2) to support future lock protocol managers.

9. Although [Gra93b] implements UNDO and REDO of transactions in the transaction manager, we implement this functionality in the log manager. Although [Gra93b] provides support for distributed transactions in the transaction manager, we have tried to encapsulate these details in the process manager. Although [Gra93b] provides a framework to support a dynamic set of resource managers, in P2 the set of resource managers is fixed at DBMS compile-time; thus, we have omitted their resource manager control blocks, resource manager transaction control blocks, and the resource manager identifier field of the transaction identifier. Although [Gra93b] allocates for transaction identifiers a structure consisting of a resource manager identifier and long unsigned integer, we use only a standard size unsigned integer. Although [Gra93b] adds a field to the transaction control block for deadlock detection, we omit this field, because our lock manager uses timeouts for deadlock detection (see Section 5.3.1).

all transactions. The anchor is initialized by the `init_xact_cb_cont()` operation which is called by the `open_cont()` operation of the `init_cont_function` component when the container is initialized. The anchor is stored using the `container_structure` component.[10] The per-transaction data is stored as *transaction control blocks* in the as shown in Figure 5.14. Each transaction control block contains the transaction's identifier, and the log sequence numbers of the transaction's first log record and most recent log record.

The transaction manager's internal container may be shared by more than one **`process`**, but need not be persistent, since it can be constructed from the log. Thus, we store the container in shared, but not persistent, memory using the `mmap_shared` component. Access to the container is protected by a single exclusive **`semaphore`**. As an optimization, when we have a single **`process`**, we can disable the declaration and use of the **`semaphore`** (see Section 5.3.1), and reduce the container to a single element, since a **`process`** can not be active in more than one transaction.[11]

5.3.4 Log Manager

Combined with the transaction manager, the log manager provides transactional atomicity, consistency, and durability. There is a single log per schema; that is, all containers share the same log. The design and implementation of the log manager is based directly on

---

10. The transaction anchor can not simply be stored as a normal global variable, because it must be shared by all processes. Data sharing functionality is provided by P2 components such as `mmap_shared`, so we would like to store the anchor in a container. But, since the anchor and per-transaction data have different element types, we would have to store them in separate containers, and the anchor container would have only a single element. When we implemented the anchor as a single-element container, we were struck by the inelegance of this solution both conceptually and practically. Conceptually, the anchor and per-transaction data are very tightly related, we would like to store it in the same container. Practically, a single-element container has some inefficiencies, because the components that implement the container were designed for multiple, rather than single elements, and thus include unnecessary functionality. We could have implemented some specialized components optimized for single-element containers, but this didn't seem like a very general solution. Instead, we decided to store the anchor in the container *structure* (rather than the container elements), using the `container_structure` component.

11. An **`xact`** may have more than one **`process`** active in it, but a **`process`** may not be active in more than one **`xact`**.

```
// Type expression.
typex {
  xact_cb_typex = top2ds_qualify[init_cont_function[container_structure[
                  avail[array[mmap_shared]]]]];
}

// Transaction identifier.
typedef unsigned XACT_ID;

// Log sequence number.
typedef unsigned LSN_INDEX;
typedef void *LSN_RBA;
typedef struct {
  LSN_INDEX index;       // Sequence number of log file.
  LSN_RBA rba;           // relative byte address (offset) in file.
} LSN;

// Transaction control block.
typedef struct {
  XACT_ID xact_id;       // This transaction's id.
  LSN min_lsn;           // LSN of transaction's first log record.
  LSN max_lsn;           // LSN of transaction's most recent record.
} XACT_CB;

// Transaction manager anchor.
typedef struct {
  XACT_ID next_xact_id;
} XACT_ANCHOR;

#if defined(PROCESS_UNIPROCESS)
// Uniprocess: maximum concurrent transactions = 1
XACT_ANCHOR xact_cb_cont;
#else
// Multiprocess: maximum concurrent transactions > 1
container <XACT_CB> stored_as xact_cb_typex with {
  init_cont_function "init_xact_cb_cont"
  container_structure "XACT_ANCHOR";
  array size is MAX_XACT;
  mmap_shared file is "/tmp/xact-manager-data"
    with size XACT_MANAGER_DATA_SIZE;
} xact_cb_cont;
#endif // PROCESS_UNIPROCESS
```

Figure 5.14 Transaction manager internal container.

[Gra93b].[12] The log manager family consists of a family of components, each implementing operation logging, but using a different discipline for flushing the log from memory to disk, as shown in Figure 5.15.

Component	Semantics
`log [xact : XACT, log : DS, log_anchor : DS] : XACT`	The default log manager. Synchronously flush log but not log anchor at `commit_xact()`.
`log_async [xact : XACT, log : DS, log_anchor : DS] : XACT`	The asynchronous log manager. Do not flush log or log anchor at `commit_xact()`.
`log_sync [xact : XACT, log : DS, log_anchor : DS] : XACT`	The synchronous log manager. Synchronously flush log and log anchor at `commit_xact()`.

Figure 5.15 Log manager components.

[Gra93b] identifies three types of logging: physical, operation, and physiological logging. *Physical* (a.k.a., value) logging records the physical address and values of objects. *Operation* (a.k.a., logical) logging records the operations and their parameters. *Physiological* (a.k.a., physical-to-a-page logical-within-a-page) logging is a compromise between physical and operation logging, where each log record applies to a single page, but is logical within the page. Both physical and physiological logging are page-based, while P2 is operation-based. If we implemented a page-based log manager for P2, we would probably simply use an existing object oriented database or persistent store to provide the functionality.[13] Such an implementation, although it would provide a useful tool, would be trivial to implement, and would not be nearly as interesting from a software engineering research perspective as the operation logging implementation that we have provided. And, of course, since the implementation of the log manager is encapsulated in a component, our

12. Although [Gra93b] implements UNDO and REDO of transactions in the transaction manager, we implement this functionality in the log manager. [Gra93b] can not (and does not) implement UNDO and REDO in the log manager, because the interface to the log manager does not contain the operations: `abort_xact()`, `begin_xact()`, `commit_xact()`, and `warm_restart()`. In P2, it is possible to implement UNDO and REDO in the log manger, because the log manager implements the XACT realm, and thus the interface to the log manager includes these operations. It is desirable to implement UNDO and REDO in the log manager to increase encapsulation and minimize dependencies between the log and transaction managers. In fact, it is possible to use the transaction manager without the log manager (although the value of doing so is dubious). Without the log manager, the transaction manager still, for example, maintains the transaction control block, including the transaction identifier (which is useful to the lock manager), but does not UNDO or REDO transactions.

13. P2 has previously been combined with the Texas uniprocess persistent store [Sin92, Jim98].

decision to use operation logging does not exclude the possibility that we may implement page-based logging in a future log manager.

Operation logging has two fundamental problems: partial actions and action consistency [Gra93b]. The *partial action* problem is that individual operations are not atomic and may fail in the middle and need to be undone. This is not a problem in P2, because all such failures are fatal, and thus are not undone, but rather redone at restart. The *action consistency* problem is that at restart we don't know the order in which the state changes made by an individual operation became persistent. This is not a problem in P2, because we can always REDO the entire log from the last `init_cont()` operation.

The declaration of the log manger's internal containers are shown in Figure 5.16.

```
typex {
  log_typex = top2ds[container_structure[
               slist_queue[malloc[mmap_persistent]]]];
  log_anchor_typex = top2ds[init_cont_function[container_structure[
                       mmap_persistent]]];
}

// Log file index.
// This is stored in mmap memory to make sure this process has the
// most recent log file in mmap memory.
typedef struct {
  LSN_INDEX index;
} LOG_CONT_INDEX;

typedef struct {
  semaphore lock; // Log lock semaphore regulating log write access.
  LSN_INDEX index;   // Sequence number of current log file.
  LSN lsn;           // LSN of next record.
  LSN prev_lsn;      // LSN of most recent record.
  LSN xact_manager_anchor_lsn; // xact mgr's most recent checkpoint
  LSN persist_lsn;   // Max LSN recorded in durable storage.
} LOG_ANCHOR;

container <LOG_STRUCT> stored_as log_typex with {
  container_structure "LOG_CONT_INDEX";
  mmap_persistent file is "/tmp/log-data" with size LOG_SIZE;
} log_cont;

container <LOG_ANCHOR> stored_as log_anchor_typex with {
  init_cont_function "init_log_anchor";
  container_structure "LOG_ANCHOR";
  mmap_persistent file is "/tmp/log-anchor-data"
    with size SIZEOF_ELEMENT_LOG_ANCHOR_CONT;
} log_anchor_cont;
```

Figure 5.16 Log manager internal container declarations.

The *log anchor* maintains the state of the log. This state includes the sequence number

(from which we can derive the name) of the current log file, the log sequence numbers of the next and most recent log records, checkpoint information, and the log lock: an exclusive semaphore protecting access to the log. Note that, once written, log records do not change, so only writes to the log anchor and the end of the log need to acquire the log lock. For example, `log_insert()`, which writes log records, needs to acquire the log lock; and `log_read_lsn()`, which reads log records, need not acquire the log lock.

The log is a container of log records. Each log record contains either the information needed to UNDO or REDO a particular operation, or the transaction manager anchor. Except for some standard header fields, each log record has a different length and internal structure, depending on the particular operation or anchor. Thus, a log record is a union. Unfortunately, P2 does not allow elements to have any type other than struct. Thus, we declare a log record to be the struct of the type LOG_STRUCT, which besides the standard header fields, is mainly a large P2 varchar field. When necessary, we cast log records to the union of the type LOG_UNION. Each field of the union is a struct specific to a particular operation or anchor. Figure 5.17 shows the declarations of LOG_STRUCT and LOG_UNION.

```
// Standard header fields.
#define LOG_FIELDS \
  LSN lsn; \
  LSN prev_lsn; \
  XACT_ID xact_id; \
  LSN xact_prev_lsn; \
  OP_CODE op_code

// Struct.
typedef struct {
  LOG_FIELDS
  varchar v[LOG_STRUCT_VARCHAR_SIZE];
} LOG_STRUCT;

// Union
typedef union {
  XACT_LOG_STRUCT x;                 // schema (a.k.a., transaction) ops.
  CONT_LOG_STRUCT k;                 // Container operations.
  INSERT_DELETE_LOG_STRUCT i, d;     // insert(), delete()
  SWAP_LOG_STRUCT s;                 // swap()
  INT_UPD_LOG_STRUCT ui;             // upd() of an int field.
  STR_UPD_LOG_STRUCT us;             // upd() of a str field.
  XACT_MANAGER_ANCHOR_LOG_STRUCT a;  // Transaction manager anchor.
} LOG_UNION;
```

Figure 5.17 Log record.

The log records every operation that changes the schema state: the contents of either individual containers or the schema as a whole. REDO using operation logging is trivial: we just perform the same operation again. Figure 5.18 shows for every logged operation a corresponding UNDO operation (if applicable), and the log record struct used to store the UNDO and REDO information for that operation. The log manager requires the operation vector manager to locate the operations to implement UNDO and REDO. The design rule that the log manager requires the `generic_init` component (see Figure 5.4) is due to the fact that the operation vector manager, in turn, requires the `generic_init` (or `conceptual`) component.

DO	UNDO	Log record type
`init_schema()`		`XACT_LOG_STRUCT`
`open_schema()`		`XACT_LOG_STRUCT`
`close_schema()`		`XACT_LOG_STRUCT`
`checkpoint_schema()`		`XACT_LOG_STRUCT`
`begin_xact()`		`XACT_LOG_STRUCT`
`commit_xact()`		`XACT_LOG_STRUCT`
`abort_xact()`		`XACT_LOG_STRUCT`
`init_cont`		`CONT_LOG_STRUCT`
`open_cont`	`close_cont()`	`CONT_LOG_STRUCT`
`close_cont()`	`open_cont()`	`CONT_LOG_STRUCT`
`delete()`	`insert()`	`DELETE_LOG_STRUCT`
`insert()`	`delete()`	`INSERT_LOG_STRUCT`
`swap()`	`swap()`	`SWAP_LOG_STRUCT`
`upd()`	`upd()`	`INT_UPD_LOG_STRUCT` if the updated field is an `int`. `STR_UPD_LOG_STRUCT` if the updated field is a `str`.

Figure 5.18 Logged operations.

There is only one schema, and schema operations have no other arguments. Thus, for schema (a.k.a., transaction) operations, the log manager needs to record only the fact that the operation was performed. Figure 5.19 shows the declaration of `XACT_LOG_STRUCT` which is used to store this information.

```
typedef struct {
  LOG_FIELDS
} XACT_LOG_STRUCT;
```

Figure 5.19 Schema operation log structure: `XACT_LOG_STRUCT`.

There may be multiple containers, but container operations have no other arguments. Thus, for container operations, the log manager needs to record only the identity of the container. We do this by recording both the operation vector manager's internal identifier for the container, and the object identifier (address) of the container. Note that recording both of these fields is possibly redundant. But container operations are performed so rarely (compared to cursor operations) that the increased log bandwidth and decreased performance created by this redundancy was deemed insignificant. Figure 5.20 shows the declaration of `CONT_LOG_STRUCT` which is used to store this information.

```
typedef struct {
  LOG_FIELDS
  CONT_ID cont_id;    // Operation vector manager's container identifier.
  void *cont_obj_id; // Container object identifier.
} CONT_LOG_STRUCT;
```

Figure 5.20 Container operation log structure: `CONT_LOG_STRUCT`.

The log manager does not need to record every cursor operation, only those that change the schema state. Except for some standard header fields, for each such operation, the log manager records different information. The `insert()` and `delete()` operation are opposites. To REDO an `insert()` or UNDO a `delete()` operation, we re-`insert()` the element; thus we must log the object identifier of the element. To REDO a `delete()` or UNDO an `insert()` operation, we re-`delete()` the element; thus we must log the contents of the element. `INSERT_DELETE_LOG_STRUCT` is used store the information. To REDO a `swap()` operation, we re-`swap()` the elements; to UNDO a `swap()` operation, we un-`swap()` the elements; thus we must log the object identifiers of the elements. `SWAP_LOG_STRUCT` is used to store this information. To REDO an `upd()` operation, we re-`upd()` the element; thus we must log the *new* value of the field. To UNDO an `upd()` operation, re un-`upd()` the element; thus we must log the *old* value of the field. This information depends on the type of the field. `INT_UPD_LOG_STRUCT` and `STR_UPD_LOG_STRUCT` store this information, for fields of type `int` and `str`, respectively. Figure 5.21 shows the declaration of these structs.

```
#define CURS_LOG_FIELDS \
  LOG_FIELDS; \
  CURS_ID curs_id; \      // Operation vector manager's cursor identifier.
  void *obj_id; \         // Object identifier.

typedef struct {
  CURS_LOG_FIELDS;
  varchar v[LOG_STRUCT_VARCHAR_SIZE];
} INSERT_DELETE_LOG_STRUCT;

typedef struct {
  CURS_LOG_FIELDS;
  void *obj_id1;          // Object identifier.
  CURS_ID curs_id1;       // Operation vector manager's cursor identifier.
} SWAP_LOG_STRUCT;

typedef struct {
  CURS_LOG_FIELDS;
  int int0;               // Old value.
  int int1;               // New value.
} INT_UPD_LOG_STRUCT;

typedef struct {
  CURS_LOG_FIELDS;
  varchar v[LOG_STRUCT_VARCHAR_SIZE]; // Old and new values.
} STR_UPD_LOG_STRUCT;
```

Figure 5.21 Cursor operation log structures: `INSERT_DELETE_LOG_STRUCT`, `SWAP_LOG_STRUCT`, `INT_UPD_LOG_STRUCT`, and `STR_UPD_LOG_STRUCT`.

The `insert()` operation assumes that all elements have the same size. This is not the case for log records. We could, of course, have the `insert()` operation assume that all log records have the *maximum* possible size. This would work correctly, but would substantially increase log size and decrease performance. The I/O system bandwidth required by the log manager is often the bottleneck for DBMS performance, especially in systems with high update rates, such as the TPC-B benchmark (see Section 5.4.1). In order to increase efficiency, we added the `insertv()` operation. This operation takes a parameter that specifies the size of the element.

As a further optimization, `insertv()` takes an additional parameter that specifies the size of the prefix of the element to copy upon insert. We use this prefix size parameter as an optimization to avoid unnecessary copying of data. Instead of copying the data into the element to be inserted, only to have the data (automatically) copied again out of the element by the `insert()` operation, the log manager uses the `insertv()` operation, and the log manager (manually) copies the data after calling `insertv()`.

The log may grow too large logically and/or physically to fit into a single file. Thus, the log is spread across several sequentially-numbered log files. When one file becomes full, we begin writing to a new file. The log manager treats files as separate finite-size containers, even though the log is conceptually a single infinite-size container. Much of the functionality of the log manager is dedicated to explicitly managing the files. For example, the log manager calls `init_cont()` to initialize a new, empty file, and `close_cont()` to close an old, full file.

The semantics of P2 containers and cursors are general enough to allow infinite-size containers. Thus, upon first glance, we thought it would be trivial to encapsulate in a P2 component the details of sequentially-numbered log files. Upon closer investigation, this proved to be too difficult to be practical. Much of this difficulty derived from our early decision to equate object identifiers and memory addresses (i.e., `cursor.obj` is a pointer). This is a common assumption for main-memory optimized DBMSs and has improved the performance and simplified the implementation of P2. Unfortunately, memory addresses are inadequate as object identifiers for log records, since the log can be too large to fit into the logical address space (4 bytes on most machines). Log records use log sequence numbers as object identifiers; log sequence numbers consist of a standard size integer (4 bytes on most machines) file index together with a memory address (4 bytes on most machines) within the file (see Section 5.14). Encapsulating sequentially-numbered log files in a P2 component would have required reconciling memory address object identifiers and log sequence numbers. This would have required a major re-write of much of P2.

By explicitly managing the sequentially-numbered log files in the log manager, we were able to continue to use memory addresses as object identifiers. And, the only change necessary was adding the `open_cont_number()` operation to the `MEM` realm. Since the log manager maintains the index number of the file, we had to provide a means for the log manager to pass the index to the `MEM` realm memory manager component (e.g., `transient`, `mmap_persistent`, or `mmap_shared`). Since it is not a constant, we could not pass the index via an annotation. Instead, we added the `open_cont_number()` operation, which takes the index as an argument.

5.3.5 Operation Vector Manager

Together with the `generic_init` (or `conceptual`) component, the operation vector manager persistently maintains the association of cursor identifiers with operation and operation name vectors.

An *operation vector* is an association between operation identifiers and the address of the functions that implement those operations. As explained in Chapter 3, operation vectors are analogous to virtual function tables in C++ [Ell90], and are used to implement the **`generic_container`** and **`generic_cursor`** data types, which permit the use of polymorphic procedures.

An *operation name vector* is analogous to an operation vector, except it associates operation identifiers and the *names* of those functions. Operation *name* vectors are used to print the log in human readable format.[14]

UNDO and REDO are polymorphic; thus they require operation vectors. They may be invoked by different programs or different invocations of the same program (for recovery); thus the operation vectors must be persistent. The operation vector manager automatically initializes and persistently maintains operation vectors.

The operation vector manager consists of the component shown in Figure 5.22.

Component	Semantics
`op_vec [process : PROCESS, op_vec : DS, op_name_vec : DS] : XACT`	Operation vector manager.

Figure 5.22 Operation vector manager component.

The operation vector manager's internal container declarations are shown in Figure 5.23. The `hash_array_overwrite` component is appropriate in this type expression, because a new operation or operation name vector with the same cursor identifier as an old vector always obsoletes the old one, and may thus overwrite it.

14. The compile-time switch `PRINT_LOG` is used to turn on/off the operation *name* vector functionality. Because of the high cost of this functionality, we usually compile P2 with this functionality disabled.

```
typex {
  op_vec_typex = top2ds_qualify[hash_array_overwrite[transient]];
  op_name_vec_typex = top2ds_qualify[hash_array_overwrite[mmap_persistent]];
}

// Operation vector.
typedef int (**OP_VEC)();

typedef struct {
  CURS_ID id;
  OP_VEC op_vec;
} OP_VEC_STRUCT;

container <OP_VEC_STRUCT> stored_as op_vec_typex with {
  hash_array_overwrite key is id with size MAX_OP_VEC;
} head_cont;

#if defined(PRINT_LOG)

// Operation name vector.
typedef char OP_NAME_VEC[MAX_OP_ID][100];

typedef struct {
  CURS_ID id;
  OP_NAME_VEC op_name_vec;
} OP_NAME_VEC_STRUCT;

container <OP_NAME_VEC_STRUCT> stored_as op_name_vec_typex with {
  hash_array_overwrite key is id with size MAX_OP_VEC;
  mmap_persistent file is "/tmp/op-name-vec-data"
    with size MMAP_PERSISTENT_SIZE;
} op_name_vec_cont;

#endif // PRINT_LOG
```

Figure 5.23 Operation vector manager internal container.

5.4 Methodology

We compared Smallbase and P2 using a modified version of the TPC-B benchmark. This version of the benchmark was written by the implementors of Smallbase as the measure of the performance of their system, thus it provides a fair basis for comparison.

5.4.1 Standard TPC-B benchmark

The *standard* TPC-B benchmark [Gra93a] is characterized by significant disk input/output, moderate system and application execution time, and transaction integrity. The benchmark uses an integer update workload. The benchmark is not on line transaction processing (a.k.a., OLTP) because it does not require terminals, network, or think time.

The benchmark is stated in terms of a hypothetical bank with one or more branches, each with 10 tellers, and 100,000 accounts. The number of branches of the bank is known as the *scale factor*, which must be at least as high as the number of transactions per second (TPS) that a system claims to perform. The system maintains the cash balance for each branch, teller, and account. Each branch, teller, and account element must have a size of at least 100 bytes. The system must also maintain a history table which records the transactions performed. Each history element must have a size of at least 50 bytes. Thus, the database must have a size of at least 100 + 10*100 + 100,000*100 + 50 = 10,001,150 bytes or approximately 10 MB per TPS. The schema is shown in Figure 5.24. The logical relationships among the branch, teller, account, and history tables are shown in Figure 5.25. The declaration of the branch, teller, account, and history structures is shown in Figure 5.26. The abbreviated declaration of the TPC-B containers is shown in Figure 5.27; we have omitted the container annotations for perspicuity. Note that the only difference in the type expressions for the multiple and single-user cases is that the former requires protocol and process managers, while the latter doesn't. Since the `protocol_smallbase` protocol manager is used, no lock manager is necessary (see Figure 5.4).

Account	Account_ID	Account_Balance	Branch_ID		
Teller	Teller_ID	Teller_Balance	Branch_ID		
Branch	Branch_ID	Branch_Balance			
History	Account_ID	Teller_ID	Branch_ID	Amount	Time_Stamp

Figure 5.24 TPC-B schema.

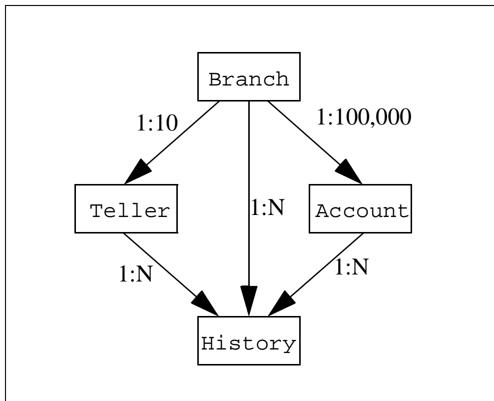

Figure 5.25 TPC-B entity/relationship diagram showing the logical relationships among branch, teller, account, and history tables, where N is the scale factor.

```
typedef struct {
  float         BALANCE;
  int           NUMBER;
  char          FILLER[80];
  unsigned char FILLER_nb;
  unsigned char SYS_fill[3];
} BRANCHES_t;

typedef struct {
  float         BALANCE;
  int           NUMBER;
  int           BRANCHNUM;
  char          FILLER[84];
  unsigned char FILLER_nb;
  unsigned char SYS_fill[3];
} TELLERS_t;

typedef struct {
  float         BALANCE;
  int           NUMBER;
  int           BRANCHNUM;
  char          FILLER[80];
  unsigned char FILLER_nb;
  unsigned char SYS_fill[7];
} ACCOUNTS_t;

typedef struct {
  float         DELTA;
  int           TELLERNUM;
  int           BRANCHNUM;
  int           ACCOUNTNUM;
  int           TIMESTAMP;
  char          FILLER[24];
  unsigned char FILLER_nb;
  unsigned char SYS_fill[7];
} HISTORY_t;
```

Figure 5.26 Branch, teller, account and history structures.

```
// Type expressions.
typex {
#if defined(MULTI_USER)
  indexed_typex =
    protocol_smallbase[LOG_LAYER[xact[op_vec[
      xact2process[process_smallbase[process2link[link2top[
      top2ds[GENERIC_INIT[INDEX_LAYER[ARRAY_LAYER[mmap_persistent]]]]]]]]]]]
#else // MULTI_USER
  indexed_typex =
    LOG_LAYER[xact[op_vec[
      xact2process[process2link[link2top[
      top2ds[GENERIC_INIT[INDEX_LAYER[ARRAY_LAYER[mmap_persistent]]]]]]]]]
#endif // MULTI_USER
  history_typex = top2ds[array[mmap_persistent]];
}

// Schema.
schema {
  container <ACCOUNTS_t> *accountContainer;
  container <BRANCHES_t> *branchContainer;
  container <TELLERS_t>  *tellerContainer;
} stored_as indexed_typex with {...} s;
container <HISTORY_t> stored_as history_typex with {...} historyContainer;
```

Figure 5.27 TPC-B containers.

Each transaction can be represented by the 6 SQL statements: 3 updates, 1 select, 1 insert, and 1 commit as shown in Figure 5.28 [Hey95]. The transaction has four parameters: *Account_ID*, *Teller_ID*, *Branch_ID*, and *Delta*. These parameters are generated randomly with a uniform distribution. The *Teller_ID* must always belong to the *Branch_ID*. The *Account_ID* should belong to the *Branch_ID* with a probability of 0.85 and to a different branch with a probability of 0.15. The *Delta* should be in the range [-999999, +999999]. The ACID properties must be supported. The benchmark must run for at least 15 minutes.

5.4.2 Modified TPC-B Benchmark

The *modified* TPC-B benchmark was adapted to Smallbase by the implementors of Smallbase. It differs from the standard benchmark in two fundamental ways. First, Smallbase is a LWDB, which is able to sacrifice features for performance. As described in 5.1, Smallbase is designed to be able to omit features, including some of the ACID properties, in order to improve the performance of applications that do not require them.[15] Thus, several of these benchmarks do not provide all of the ACID properties. Second, Smallbase is opti-

```
(1) UPDATE Account SET Account_Balance = Account_Balance + :Delta
    WHERE Account_ID = :Account_ID

(2) UPDATE Teller SET Teller_Balance = Teller_Balance + :Delta
    WHERE Teller_ID = :Teller_ID

(3) UPDATE Branch SET Branch_Balance = Branch_Balance + :Delta
    WHERE Branch_ID = :Branch_ID

(4) SELECT Account_Balance
    INTO :Var
    FROM Account
    WHERE Account_ID = :Account_ID

(5) INSERT INTO History
    VALUES (:Account_ID, :Teller_ID, :Branch_ID, :Delta, :Time_Stamp)

(6) COMMIT WORK
```

Figure 5.28 TPC-B transaction profile.

mized for databases that are main-memory resident. Thus, the database can not be scaled by 10 MB per TPS, nor run for the full required 15 minutes, because the database would overflow physical memory.

We repeated our benchmarks for a variety of DBMS configurations: both single and multi user, and with a variety of logging methods: no log, asynchronous log, and synchronous log. The single-user database supports a single thread of execution, and thus does not need to regulate concurrent access to objects. The multiple-user database supports multiple threads of execution, and thus must regulate concurrent access to objects. The no log benchmarks do not maintain a log. The asynchronous log benchmarks maintain a log and write the log to disk at the end of each transaction, but do not wait for this write to finish before committing. The synchronous log benchmarks maintain a log, write the log to disk at the end of each transaction, and wait for this write to finish before committing. Only the synchronous log benchmark implements the full ACID properties. In Smallbase, these configurations are specified by the access and options parameters to the `dbCreate()` operation. In P2, these configurations are specified using the type expressions in Figure 5.27. These configurations are summarized in Figure 5.29.

15. Smallbase can omit concurrency control, persistence, and logging. Even if logging is included, Smallbase can flush it from memory to disk asynchronously. If persistence is omitted, then durability is lost. If logging is either omitted or performed asynchronously, then atomicity, consistency, and durability are lost.

Semantics	Smallbase	P2

Single-user
No log
	`access = Excl`
	`options = DbPrivate|DbNoLogging`
		`#undef MULTI_USER`
		`#define GENERIC_INIT null`
		`#define LOG_LAYER null`

Single-user
Async log
	`access = Excl`
	`options = DbPrivate`
		`#undef MULTI_USER`
		`#define GENERIC_INIT generic_init`
		`#define LOG_COMPONENT log_async`

Single-user
Synch log
	`access = Excl`
	`options =`
	`DbPrivate|DbXactDurable`
		`#undef MULTI_USER`
		`#define GENERIC_INIT generic_init`
		`#define LOG_COMPONENT log_sync`

Multi User
No log
	`access = Share`
	`options = DbNoLogging`
		`#define MULTI_USER`
		`#define GENERIC_INIT null`
		`#define LOG_COMPONENT null`

Figure 5.29 Experimental DBMS configurations.

Smallbase uses a fixed indexing method: hash indexes for exact value searches, and T-trees for range queries. P2 supports a variety of indexing methods. These indexing methods are specified using the type expressions in Figure 5.27. For each of the configurations in Figure 5.29, we used each of the indexing methods shown in Figure 5.30. These indexing methods are named after their retrieval component. The `hash0` and `hash1` indexing methods are identical, except that `hash0` uses the default hash function, and `hash1` uses a hash function that is customized for the TPC-B benchmark: the identity function. Besides being efficient to compute, the identity function is perfect (i.e., has no collisions) for the TPC-B benchmark. Thus, we can (and do) use the identify function for the `hash_array_overwrite` indexing method.

Indexing Method	P2
`red_black_tree`	`#define INDEX_COMPONENT red_black_tree` `#define ARRAY_COMPONENT array`
`hash0`	`#define INDEX_COMPONENT hash` `#define ARRAY_COMPONENT array`
`hash1`	`#define INDEX_COMPONENT hash` `#define ARRAY_COMPONENT array`
`hash_array_overwrite`	`#define INDEX_COMPONENT hash_array_overwrite` `#define ARRAY_COMPONENT null`

Figure 5.30 P2 indexing methods.

5.5 Results

Figures 5.31, 5.32, and 5.33 show the performance results for single-user benchmark with respectively no log, an asynchronous log, and a synchronous log. Figure 5.34 shows the performance results for the multiple-user benchmark with no log.

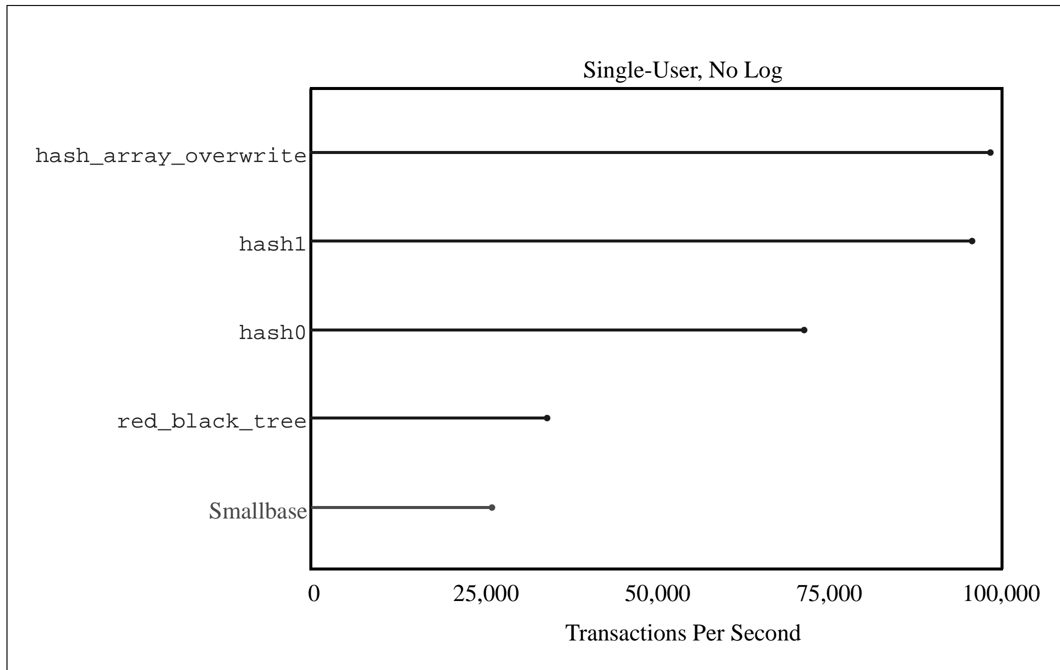

Figure 5.31 Modified TPC-B Benchmark: Single-user, No Log.

The single-user configurations are based on the Smallbase program `tpcbApiCT.c`; the multiple-user configuration is based on the Smallbase program `tpcbApiMU.c`. These programs use the storage manager, rather than the SQL, interface to Smallbase. The storage manager interface is significantly more efficient than the SQL interface, and much more similar to P2's interface. The P2 version of the benchmarks were adapted from the Smallbase programs by replacing the Smallbase statements with analogous P2 statements.

We used Smallbase version 4.5 and P2 version 1.0 for our benchmarks. Our benchmarking platform was a HP 9000/770 (single HP-PA 120 MHz processor) machine with 128 MB of physical memory running HP-UX version 10.20. We compiled the

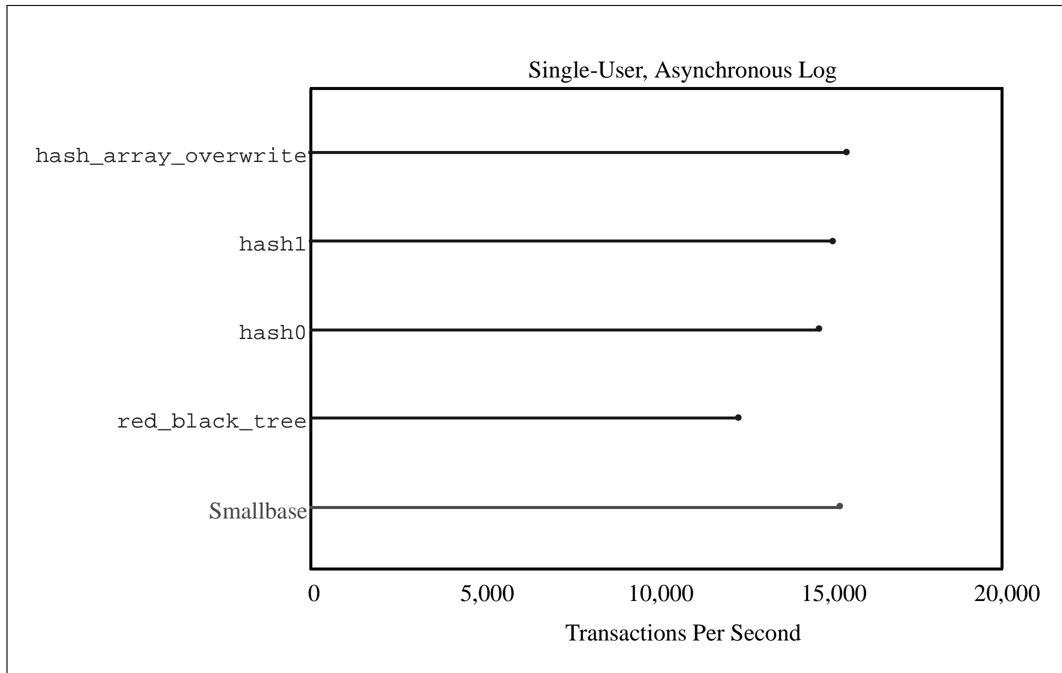

Figure 5.32 Modified TPC-B Benchmark: Single-user, Asynchronous Log.

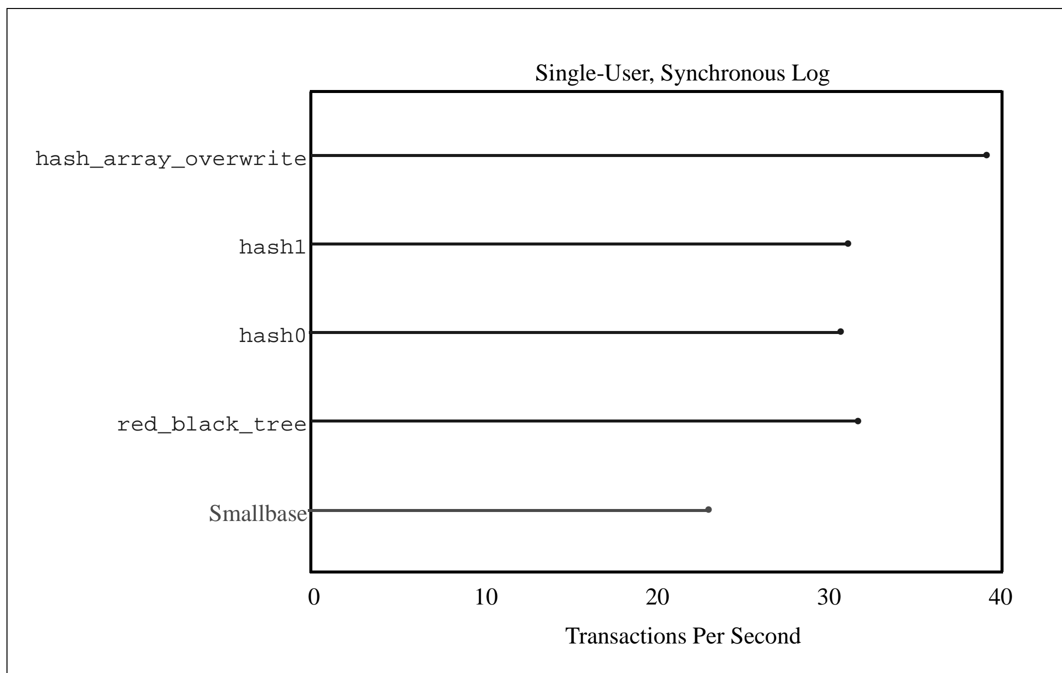

Figure 5.33 Modified TPC-B Benchmark: Single-user, Synchronous Log.

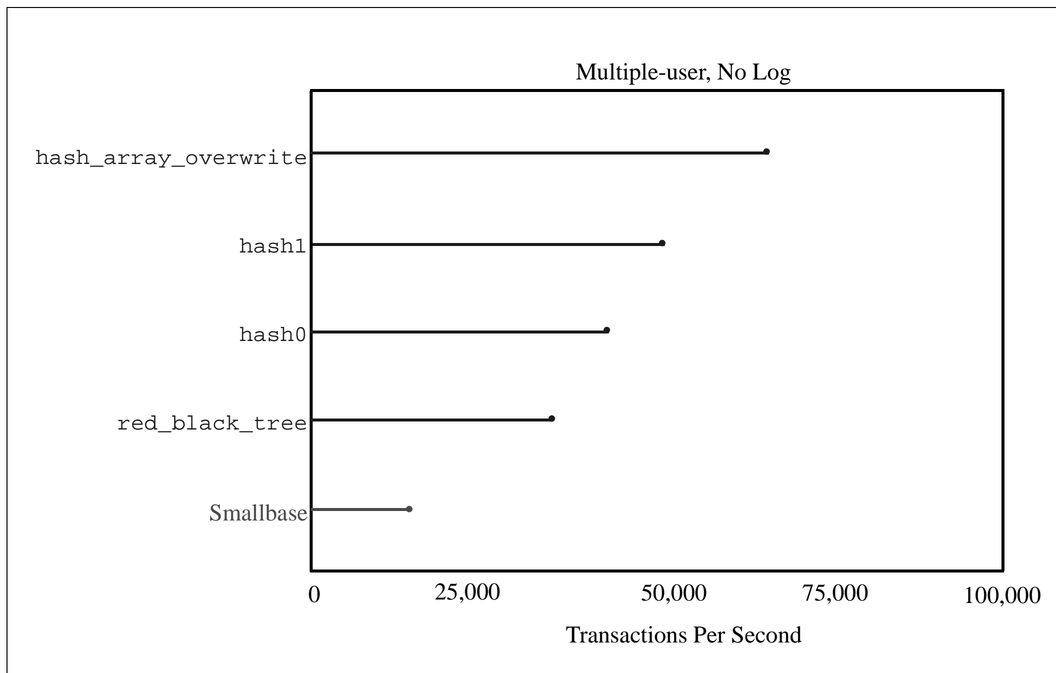

Figure 5.34 Modified TPC-B Benchmark: Multiple-user, No Log.

Smallbase benchmarks using the HP-UX C compiler (`/bin/cc`) version 10.32.15 with the options for unlimited optimization: `-O +Onolimit`, and the P2 benchmarks using the GNU CC compiler (gcc) version 2.8.1 with the option: `-O2`.

This experimental methodology is almost identical to [Hey95]. We ran each benchmark with a scale factor of 3 (the largest supported by Smallbase). The single-user no log and asynchronous log configurations consisted of 300,000 transactions; the synchronous log configuration consisted of 3,000 transactions; the multiple-user configuration consisted of 30,000 transactions, divided evenly between 4 concurrent *processes* (i.e., 7,500 transactions per process, with a maximum of 4 concurrent *transactions*). The TPS for the multiple-user configuration is the sum of the TPS for each individual process. We ran each benchmark 25 times[16], once for each of the random number seeds 1 through 25. Let's consider each benchmark in turn:

16. [Hey95] ran each benchmark only 5 times.

- For the single-user, no log benchmark, P2 displays significantly better performance than Smallbase. This benchmark requires only a very lightweight system. Thus, miscellaneous (e.g., initialization, finalization, updates, transaction management) and retrieval costs dominate. `hash0` is about a factor of 3 faster than Smallbase, indicating that P2 is able to significantly reduce these miscellaneous costs. We speculate that the significantly higher miscellaneous costs incurred by Smallbase are due to vestiges of unnecessary generality. `hash_array_overwrite` and `hash1` are about a factor of 4 faster than Smallbase, indicating that more advanced retrieval algorithms are able to further improve performance.
- For the single-user, asynchronous log benchmark, P2 and Smallbase display very similar performance. This benchmark requires a large amount of system time to write the log to disk, but since the writes are asynchronous, it need not wait for the data to be actually, physically written to disk. Thus, system call costs dominate, and retrieval costs are much less important. `hash_array_overwrite`, `hash1`, `hash0`, and Smallbase all use O(1) hash indices, and have nearly identical performance. `red_black_tree` uses a O(log n) self-balancing tree indices, and has only slightly lower performance. Note that the performance of this benchmark is very sensitive to the size of the log. By increasing or decreasing the amount of data written to the log for each transaction, the performance of this benchmark can be significantly increased or decreased. For example, [Gra93b] writes the log sequence number of every record to the log. Although useful as a consistency check, this information can be recalculated as necessary. Thus, to improve performance, P2 omits this information. In the results presented here, both P2 and Smallbase, write 5 log files, of size about 8 MB each, a total of about 40 MB.
- For the single-user, synchronous log benchmark, P2 displays slightly better performance than Smallbase. This benchmark must wait for the log to be actually, physically written to disk. The mechanical disk latencies are huge, and there is a drastic reduction in performance. Because of these latencies, retrieval costs are largely

irrelevant. `hash1`, `hash0`, and `red_black_tree` all have approximately the same performance. `hash_array_overwrite` has slightly higher performance. Smallbase has slightly lower performance.

- For the multiple-user, no log benchmark, P2 displays slightly better performance than Smallbase. Both P2 and Smallbase use the same lock implementation (Smallbase latches) and locking protocol (serialize all transactions). Thus, this benchmark is very similar to the single-user, no log benchmark, except with the significant overhead due of concurrency control.

Notice that overall, the performance of P2 and Smallbase are remarkably similar. The results for the various benchmarks vary from nearly 100,000 TPS for the P2 `hash_array_overwrite` implementation of the single-user, no log benchmark to less than 25 TPS for the Smallbase implementation of the single-user, synchronous benchmark: a factor of 4000 difference in performance. Yet, for each benchmark, the best and worst performance for P2 and Smallbase differ by less than a factor of 5. Since they both use general-purpose hash indexes, `hash0` is most similar to Smallbase. For each benchmark, the best and worst performance for `hash0` and Smallbase differ by less than a factor of 3; for the benchmarks that perform logging, they differ by only a few percent. Thus, P2 maintains performance.

The programmer productivity provided by P2 is at least as high as that of Smallbase, as measured by source code size, compilation times, or ease of customization. The P2 version of the source code for the modified TPC-B benchmarks is slightly shorter than the Smallbase version. Due to the high cost of the `+Onolimit` option, it takes significantly longer to compile the executables (LWDBs) for the Smallbase benchmarks than for the P2 benchmarks, which do not use this option. The P2 and Smallbase LWDBs are equally easy to customize: in Smallbase, customizations are specified via the access and options parameters to the `dbCreate()` operation; in P2, via type expressions. Thus, P2 maintains productivity.

Yet it is important to remember that while maintaining performance and productivity, P2 offers the advantages of GenVoca generation. The purpose of the experiment

described in this chapter was to test P2's scalability, *not* demonstrate these advantages. Nevertheless, these advantages are readily observable. P2 is significantly more flexible than Smallbase. Due to its conventional parameterization mechanism, Smallbase can provide only a limited, pre-defined set of feature choices, and this customization is not performed until run-time. By exploiting GenVoca generation, P2 provides a much broader, extensible set of feature choices, and performs this customization at compile-time. As the performance improvements of `hash_array_overwrite` and `hash1` versus `hash0` and Smallbase hint at, P2's greater capacity for customization can result in significant performance improvements.[17] As the performance improvements of `hash0` versus Smallbase show, P2's run-time customization is more complete, and better able to eliminate unnecessary generality. This is also evident by looking at the size of the executables. Smallbase executables are approximately 2MB in size; P2 executables 200KB. P2 executables are an *order of magnitude* smaller, because Smallbase executables are linked with a monolithic, pre-compiled system which contains significant un-necessary functionality (e.g., an SQL interpreter and query optimizer).

5.6 Conclusions

The goal of our work is to prove that we can scale a GenVoca generator to produce complex systems, and at the same time maintain good generated code performance and high programmer productivity.

The goal of this experiment was to reengineer Smallbase as accurately as possible within the framework of the GenVoca model. We did *not* attempt to improve upon the Smallbase algorithms and data structures, as we had done with LEAPS. Rather, we wished to eliminate differences, in order to isolate the effects of construction methodology on

17. If the purpose of the experiment described in this chapter was to produce the fastest possible system, we could have introduced algorithmic optimizations, such as group commit [Gra93b] or compression of the log before writing it to disk. We speculate that these optimizations would produce a performance improvement similar to orders of magnitude improvement produced by the LEAPS hash optimization described in Chapter 4.

generated code performance and programmer productivity. If we had achieved our goal fully, Smallbase and P2 would have exactly the same performance as measured by TPS.

As shown in Section 5.5, the finished Smallbase and P2 systems have remarkably similar performance. We believe that the slight performance advantage of P2 can be explained by the differences in construction methodology and how it effects feature customization. Smallbase is a single, monolithic, pre-compiled executable, P2 a GenVoca generator. Customization of features is accomplished at run-time in Smallbase by parameters to the `dbCreate()` operation, at compile-time in P2 by type expressions (see Figure 5.29). Because P2 knows at compile-time what features are needed, it is able to omit unnecessary features and specialize the implementation of the features that are needed to maximize performance.

We believe that this experiment demonstrates that we were able to reengineer Smallbase, and thus scale P2 to generate complex systems, while maintaining performance and productivity.

Chapter 6

Retrospective

In this chapter, we analyze the lessons of P2. We consider the advantages and disadvantages of our implementation, approach, and field; what we did right and what we did wrong; possibilities and limitations. We proceed from specific (low level) details about the P2 tool itself, through GenVoca generators, to general (high level) issues concerning LWDBs.

6.1 P2

P2 was conceived as a "better" Predator (P1) to help us re-engineer LEAPS; we succeeded in this goal.[1] For instance, although we began P2 after P1, we finished re-engineering LEAPS with P2 before P1. Additionally, we were able to add hash optimizations to P2 that we were never able to add to P1. As discussed in Chapter 4, these optimizations resulted in a performance improvement of several orders of magnitude.

Our choice of C as the host language for P2 was the right one at the time, but the wrong one in retrospect. As discussed in Chapter 3, we chose C as the basis language for P2, because in 1992 when we began our implementation, it was the only language that satisfied the following criteria: (1) compatible with our target applications (written in C),

1. We had to abandon the P1 prototype because it did not scale to large systems. Although by the standards of P3 [Bat98b], the component definition facilities of P2 are primitive, P2 represents a significant step forward from P1.

(2) permits high performance, system-level programming, (3) standardized (as ANSI/ISO 9899-1990), and (4) easy to parse and extend (C grammars were freely available).

If we were to re-implement P2, we could possibly use C++, since it now satisfies all of these criteria: (1) C++ is a superset of C, (2) C++, like C, permits high performance, systems level programming, (3) C++ was standardized in 1997 as ISO/IEC FDIS 14882, (4) several C++ source-to-source translators are now available, for example Sage++ from Indiana University [Bod94] and the C++ front end from Edison Design Group [Edi98]. But we would probably use P++ [Sin96], since it is a superset of C++ that satisfies all of these criteria, and in addition provides linguistic features specifically designed to support the GenVoca model. We would not use IP [Sim95], which satisfies criteria (1), (2), and (4), but not (3); nor Java, which satisfies criteria (3) and (4), but not (1) and (2).[2]

6.1.1 ddl

The job of the `ddl` translator is to convert type expressions and annotations from the data definition language (`ddl`) to the internal language of the backend. We chose a very structured DBMS-like language close to English for the `ddl` (see Figure 6.1). This has advantages and disadvantages versus a less structured language closer to a programming language–the approach used for example in DiSTiL [Sma97] (see Figure 6.2) and P3 [Bat98b]. The primary advantage of the P2 `ddl` approach is that it makes it easy for users to *read* the annotations. The disadvantages of the P2 `ddl` approach are (1) that it is difficult for users to *write* annotations, and (2) difficult for implementors to design and maintain the `ddl`.[3]

- *Annotations*. In both `ddl` approaches, the user must remember the order that the parameters are listed in the annotations (e.g., `mmap_persistent` requires the parameters to be in the order file name, size). But, in the P2 `ddl` approach users must also

2. P3 is analogous to P2, but is written in Java using JTS [Bat98b], and therefore does not satisfy criteria (1) or (2). This is not surprising, because P3 was not designed to reengineer LEAPS or Smallbase.
3. The format of the `ddl` actually follows the DBMS approach to annotating schema with pragmas. Thus the format the `ddl` took was not unprecedented (or entirely stupid).

```
// Declaration of type expression C.
typex {
  C = top2ds_qualify[inbetween[dlist[array[mmap_persistent]]]];
}

// Container declaration using type expression C.
container <PERSON> stored_as C with {
  array size is 10;
  mmap_persistent file is "/tmp/foo" with size 10000;
} faculty_container;
```

Figure 6.1 Example P2 type expression and annotations.

```
// Declaration of type expression C.
typeq (PERSON, Dlist(Array(Transient))) C;

// Container declaration using type expression C.
Container (C, (Array(10), Persistent("/tmp/foo", 10000))) faculty_container;
```

Figure 6.2 Example DiSTiL type expression and annotations.

remember what combinations of keywords proceed each parameter (e.g. `mmap_persistent` requires that "`file is`" proceed file name and "`with size`" proceed size). These keywords are extraneous, and have proven to be difficult for users to remember.

- *Design and Maintenance*. The P2 `ddl` is difficult to design and maintain because it is a complete translator with its own lexicographical analyzer, parser, and output language. Thus, every time we add a new type of annotation, we have to add a new reserved word to the `ddl` language, rebuild the entire `ddl` system, and hope that are no conflicts in the grammar. To simplify the design and maintenance of the `ddl` translator we implemented it as macro processor with no symbol table that looks for specific keywords (**`typex`** and **`with`**) and outputs what follows in a fixed format.

Because of the fixed format of the `ddl` output language, the `ddl` translator has difficulty handling components with a variable number of annotations. Thus, in many cases, minor variations of components cannot be specified using annotations (additional parameters to the component). These variations include for example, optimizations for the case that the retrieval key is primary. The limitations of `ddl` have forced us to embed these variations into separate components (specializing the component to reflect fixed values of the addi-

tional parameters). Thus, in addition to the original binary tree component (red_black_tree), P2 also provides a component (red_black_tree_primary) specialized for the case that the retrieval key is primary.

Because ddl has no symbol table, it is unable to provide several features that would have greatly enhanced its usability; for example, static type checking of type expressions (to catch design errors as soon as possible), type expressions that take other type expressions as parameters, and forming new type expressions by combining existing type expressions. For example, see Figure 6.3. In general, we would like the ddl to be much more robust and powerful. Ideally, it would combine design rule checking and a graphical user interface, as do Genesis [Bat88] and P3 [Bat98b].

```
typex {
  // Type expression A with static type DS.
  A : DS = array[mmap_persistent];
  // Type expression B with parameter X.
  B[X : DS] : TOP = top2ds_qualify[inbetween[X]];
  // Type expression C formed by combining A, B, and dlist. Equivalent to
  // C : TOP = top2ds_qualify[inbetween[dlist[array[mmap_persistent]]]]
  C : TOP = B[dlist[A]];
}
```

Figure 6.3 Example of type expression static typing, parameters, and combination.

6.1.2 Design Rule Checking

The job of the design rule checker (DRC) is to identify type expressions that are *semantically* incorrect [Bat97a]. Unfortunately, the P2 DRC has two flaws: (1) the implementation is not properly integrated with the rest of P2 and (2) the paradigm does not allow for consideration of the actual usage of type expressions:

- *Integration with the rest of P2.* The information that the DRC uses is a system of attributes invented and encoded by a domain expert. Since the domain expert is writing rules about *combinations* of components, he or she must be familiar with the semantics of all the components and their interactions. If the semantics of any component changes, or a component is added to or deleted from the library, the DRC attributes may need to be modified. To force the updating of a component's DRC attributes when the component is changed, the DRC attributes should be encapsu-

lated in the component proper. This is not done in P2. The DRC information from the components is stored in a separate text file and no language mechanisms force component implementors to update the DRC information to reflect their changes. Thus, the P2 DRC information became obsolete soon after is was first implemented.

- *Actual usage of type expressions.* P2 DRC does not consider the actual usage of type expressions. For example, consider type expression `C` shown in Figure 6.1. DRC decides that type expression `C` is semantically incorrect only because it does not contain a delete flag component (e.g. the `delflag` or `avail` components). In fact, the correctness of `C` depends on what cursor and container operations will be performed. `C` is correct for some operations, but `C` is incorrect for the `delete()` operation (the only operation that requires a delete flag component), incorrect for the `insertv()` operation (not supported by the `array` component), incorrect for the `cardinality()` operation (only supported by the `cardinality` component), and even with a delete flag component, incorrect if we are going to be performing the `delete()` operation inside of a loop (`red_black_tree` is unstable, instead of it, we need to use the component `red_black_tree_stable`). Somehow, DRC needs to consider what operations will actually be performed. Note that this problem is analogous to the problem `pb` has with increased code size due to proceduralized[4] operations that are never used (see Section 6.1.4). One solution to this problem is to generate DRC errors only when the code for a particular operation is generated. This solution is used in DiSTiL. This solution will not solve this problem entirely, since the code for a particular operation may be generated but never executed. Nevertheless this would make DRC much more accurate. Another solution to this problem would be to allow users to cast the realm of the type expression to an appropriate *super-realm* that reflects the operations that will actually be performed[5]. This solution works well, except that the number of super-realms is exponential in the number of operations, so a more powerful realm notation is required.

---

4. To *proceduralize* (a.k.a., *outline*) is to put code into a separate function body; it is the opposite of inline.

6.1.3 `xp`

The job of the `xp` translator is to convert component definitions from the component definition language (`xp`) to the internal language of the backend. The *concept* of a component definition language is perhaps the most valuable contribution of P2. `xp` helped us define both the problems and promise of component definition languages. `xp` results in an order of magnitude productivity improvement for component definition, but because of its inelegant implementation, still falls short of its potential by an order of magnitude.

Previous GenVoca generators such as Genesis [Bat88] and P1 [Sir94] had no component definition language; that is, components were written by hand, in an ad hoc manner. Lack of a component definition language was not a major problem in Genesis, which is *compositional*; that is components specify application code directly. But lack of a component definition language became a major problem in P1, which is *generational*; that is, components specify the code that *generates* the application code. This extra level of indirection is a major conceptual hurdle for the component implementor. For example, Figure 6.4 shows hypothetical source code for the `adv()` operation of a singly-linked list written in a compositional system. Figure 6.5 shows analogous source code written in a generative system. Notice that the `printf()` obscures the source code that it produces. In addition, none of the type checking or function definition and invocation facilities of the host language can be used until the application code is generated.[6]

A primary goal of `xp` is to make the source code of the generative system at least a simple as that of a compositional system, and allow type checking and function definition/invocation facilities to be used. Figure 6.6 shows the actual `xp` source code for the `adv()`

5. The notion of *sub-realms* is discussed in [Bat97a] as a means to formalize subjectivity. There, by analogy to the object-oriented notion of sub-classes, a sub-realm is a *superset* of an existing realm; that is, a sub-realm *adds* operation(s) to the existing realm. Here, a *super-realm* is a subset of an existing realm; by analogy to the object-oriented notion of super-classes, a super-realm is subset of an existing realm, that is super-realm *removes* operation(s) from the existing realm.

6. The use of code templates in DiSTiL and JTS/P3 provides an elegant means of supporting generation, and allows a single language to support both component definition and use. A *code template* converts a code fragment into an internal representation (e.g., abstract syntax tree). Code templates combine *tree constructors* (analogous to backquote in LISP) with *escapes* (analogous to comma a.k.a., unquote in LISP) [Bat98a].

operation of the singly-linked list component, `slist`. Notice how the actual `xp` source code is very similar to the example compositional source code. The differences in the actual `xp` source code are (1) the function return type (`void`) and layer name (`slist`) can be is omitted, since `xp` knows this information already (they are part of the realm and layer definition), (2) the explicit call down (`type_expression->down[0]::adv(cursor)`) can be omitted (since this is the default action), and (3) the curly braces (`{...}`) enclosing the function definition have been replaced by percent curly braces (`%{...%}`). These percent curly braces indicate to `xp` that the enclosed source code is a template for code to be *generated*, rather than *executed*, at translation time. Thus, `xp` is much better than `printf()`. But `xp` has many problems that prevent it from fully realizing its potential.

```
void slist::adv (type_expression, cursor)
{
  // Application code added by this component.
  cursor.obj = cursor.next;
  // Call the adv operation of the next component down.
  type_expression->down[0]::adv(cursor);
}
```

Figure 6.4 Example compositional singly-linked list adv().

```
void slist::adv (type_expression, cursor)
{
  // Generate the application code added by this component.
  printf("(%s).obj = (%s).next\n", cursor->name, cursor->name);
  // Call the adv operation of the next component down.
  type_expression->down[0]::adv(cursor);
}
```

Figure 6.5 Example generative singly-linked list adv().

```
adv (cursor)
%{
  // Generate the application code added by this component.
  cursor.obj = cursor.next;
%}
```

Figure 6.6 Actual xp singly-linked list adv().

As with the `ddl` translator, to simplify the design and maintenance of the `xp` translator, we implemented it as macro processor with no symbol table that looks for specific keywords (e.g., `%{`, `%}`, **`container`**, **`cursor`**) and outputs what follows in a fixed format. `xp` does not perform type checking, cannot understand function definition and use, and is

confused by complex expressions, but is nevertheless able to process a significant subset of C. When we translated a component (`red_black_tree`) from DiSTiL to `xp`, we found that because of these limitations, `xp` suffers an order of magnitude *decrease* in programmer productivity relative to the more elegant component definition language in DiSTiL. We estimate that `xp` nevertheless yields an order of magnitude *improvement* in programmer productivity relative to `printf()`. This estimate is largely subjective, but empirical evidence comes from the fact that `xp` typically expands code size by well over an order of magnitude. That is, the generator code output by `xp` is typically well over an order of magnitude larger than the component source code input to `xp`.

Overall, `xp` has taught us several important lessons: (1) the component definition language should be easy to use, (2) the component definition language should be easy to modify, (3) component definition and use should be supported in a single, unified language, (4) a facility for specifying default implementations of operations is very useful for component definition languages:

- *Ease of use*. The ease of using `xp` versus `printf()` can be seen in the increased size of the library of P2 versus P1, approximately 75 versus 20 components. But `xp` is more difficult to use than it could be. This difficulty of use can be seen in the fact that the library is not as big as it should be. We were never, for example, able to get the AVL tree component to work correctly, despite the fact that we began with correct source code from the gnu library. Eventually, we gave up on this component and removed it from the P2 library. We also, for example, have numerous components on our “to do” list: tree and list components with dummy head and tail nodes, 2-3 tree, T-tree, B-tree, more sophisticated (e.g. Texas [Sin92]) and less sophisticated (e.g. flatten and write to a file) persistence, compression, encryption, and more `LINK` realm components. Despite the availability of correct source code for all of these components, the difficulty of using `xp` has kept us from translating them to `xp`.

- *Ease of modification*. The difficulty of modifying xp has kept us from making the formal realm interfaces of certain components match their actual semantics. This is particularly apparent for the XACT and PROCESS realms (see Chapter 5), and particularly disappointing because the design and validation of these GenVoca realms is one of the primary research contributions of our work.
- *Single language for component definition and use.* P2 provides separate languages (xp and p2) for component definition and use, and this was a major problem. In many components, we needed to use the p2 language, but could not. In the orderby components, for example, we need to specify and use cursors over an auxiliary, sorted container. We implemented this functionality in the .xp files, but it was very difficult. As explained in Chapter 5, the lock, log, operation vector, trace, and transaction managers also use containers internally. Because of the difficulty of implementing this functionality in xp, we implemented it in .p2 files, which are compiled outside of the xp system. In many circumstances, we found ourselves implementing similar functionality in the xp and backend translators, or wishing that such functionality existed.
- *Default implementation of operations*. xp provides a facility for specifying default implementations of operations, and this facility has proven very helpful. Without such functionality, each component would need to implement all of the operations in its realm, which would greatly expand the size of each component, and necessitate the retrofitting of existing components every time a new operation is added or renamed. In the DS realms, for example, the most common implementation is to do nothing and merely call down. Most operations can be given this default. There are approximately 40 operations in the DS realm, but the average number of operations per DS component is approximately 12. Thus, the fact that most operations are defaulted significantly reduces the size of components. Also, adding new operations and renaming existing operations is very common. In the process of implementing

the resource managers, for example, we added or renamed 12 operations in the `DS` realm alone. The fact that most operations are defaulted in most components saved us from having to retrofit many layers to reflect these new and renamed operations.

6.1.4 `pb`

The job of the predator backend (`pb`) is to integrate the P2 (`.p2`) source translated by `ddl`, with the component (`.xp`) source translated by `xp`, to produce C code. That is, `pb` is P2's code generator. The input to `pb` is C code with extensions for special data types (e.g., cursors and containers). `pb` calls the code generated by `xp` to generate the code to implement operations on these special data types. The output from `pb` is standard C. Thus, `pb` is basically a glorified C source-to-source translator. The primary concerns of `pb` are (1) the extensions to C that it supports as input code, and the (2) correctness and other properties of the generated output code: (3) basic properties, (4) query optimization, and (5) code size:

- *Extensions*. The extensions to C that `pb` supports are fairly cleanly integrated into C. As discussed in Chapter 3, the element, container, cursor, and schema special data types are first-class; they can be used like any C type. Elements are simply C structs. Containers, cursors, and schemas are integrated with the C type system by placing them in the same syntactic category as structures and unions. The only major problem with the extensions is that `pb` does not allow `extern` container, cursor, and schema declarations. This limits pb's ability to support separate compilation–all code that uses a given container, cursor, or schema must be included in the same file. The root of this problem is that container, cursor, and schema declarations may cause `pb` to call code generated by `xp` to generate *other* declarations. For example, when a container has its operations proceduralized (via the `generic_funcall` or `named_funcall` components), the container declarations will generate functions to implement all of the container operations. Or when a schema uses the log component, the schema declaration will call the `verbatim_s()` operation to generate a function to implement warm restart. Neither `pb` or `xp` is smart enough to make the

other declarations `extern` when the original declarations are `extern`. Unfortunately, it is not clear how to make `pb` or `xp` smart enough to handle `extern` container, cursor, and schema declarations without making them much more complicated and difficult to write.

- *Correctness*. `pb` does a fairly good job at generating correct code. `pb` produces correct C source-to-source transformations for almost all "real" programs written entirely in standard C. When `pb` produces incorrect output, it is almost always for programs written using the P2 extensions to standard C (e.g., cursors and containers), or "test" programs designed *specifically* to break `pb`. These "test" programs are typically designed to overflow statically allocated data structures, of which `pb` has many. Correct C source-to-source translation turns out to be a more difficult job than we imagined. Even though we started with a correct C grammar, we wasted a lot of time doing only a fairly good job. If we had to do it again, we would start with a complete source-to-source translator, so that we could concentrate on the P2 extensions rather than the basis language.
- *Basic properties*. `pb` performs some transformations that are correct but not necessarily desirable. Probably the least desirable is that `pb` transforms every declaration with a multiple variable declaring list into multiple declarations with a single variable per declaring list. Figure 6.7 shows an example of such a transformation. This transformation was required in the original specification, but now serves *no* purpose in P2. This transformation is undesirable, because it adds complexity to `pb`, and obscures the output code, which makes debugging more difficult. Other transformations performed by `pb` serve useful purposes, but are undesirable for the same reasons. Name mangling is necessary to avoid conflicts, but makes symbols hard to recognize in the output code. Translation by the C pre-processor is necessary to support macros, separate compilation, etc., but severely obscures the output code by appending `#include` files and eliminating comments. If we were to re-implement `pb` we would try to eliminate or reduce these transformations.

Original
`// Declaration with a multiple variable declaring list.` `int i, j, k`
Transformed

`// Multiple declarations with a single variable per declaring list.`
`int i;`
`int j;`
`int k;`

Figure 6.7 Example unnecessary transformation.

- *Query optimization*. A LWDB achieves high performance because it is specialized to the needs of the target application. Because specialization has a cost, `pb` performs as much specialization as possible at compile time rather than run time. In fact, `pb` even performs query optimization at compile time. Thus, `pb`'s query optimizer can consider only information that is available at compile time, such as type expressions and qualification predicates; `pb` can *not* consider information that is available only at run time, such as container cardinality and predicate selectivity. `pb` optimizes retrievals using only the components specified by the user; `pb` does *not* add or remove components from type expressions to improve application performance[7]. `pb` joins containers in the order they are specified in composite cursors (see Chapter 4), using the `LINK` components specified by the user; `pb` does *not* perform any inter-container query optimization. The savings due to eliminating the cost of run time query optimization, however, can easily be outweighed by the huge penalty due to potentially sub-optimal query plans. Most users of P2 so far (us) have been experts at selecting type expressions and join orders. Thus, we have avoided this penalty. But in the future, it will be necessary to provide more sophisticated query optimization to allow novices to more effectively use P2. The accurate estimate of intermediate results warrants particular attention, because it greatly effects the cost of the query.

7. `xp` can add or remove certain components from type expressions to match the number of annotations given to the component, but *not* to improve performance.

- *Code size*. The code generated by pb is much too large. This is due to three main problems: pb may (i) link-in unnecessary libraries, (ii) generate functions for unused operations, and (iii) generate redundant code, especially for predicate testing:

  Unnecessary Libraries. After application generation and C compilation, pb links in a static library. This library includes functionality written directly in P2 or C, such as the data type (i.e., int and str) specific functions (e.g. int_hash() and str_hash()), and the memory mapped persistence and sharing layer (i.e. mmap_persistent and mmap_shared) specific helper functions (e.g. open_mmap_memory() and new_persistent_var()). Unfortunately, pb links in this library with *all* applications regardless of whether or not they use any of this functionality. This library is fairly large, so it would be preferable to have pb link in only the required functionality, either by putting code from different components into different libraries and keeping track of what components were actually used, or dynamically linking the library.

  Unused Operations. When a container has its operations proceduralized, pb generates functions for all operations, regardless of whether or not these operations are ever used. Note that this problem is analogous to the problem design rule checking has with illegal but unused operations (see Section 6.1.2). The potential solutions to this problem include those for that problem. Alternatively, we could declare these functions as inline static, and hope that the C compiler will be smart enough to optimize-away those functions that are unused.[8]

8. The GNU CC compiler, for example, is smart enough to optimize-away such functions. “When a function is both inline and ‘static’, if all calls to the function are integrated into the caller, and the function’s address is never used, then the function's own assembler code is never referenced. In this case, GNU CC does not actually output assembler code for the function, unless you specify the option ‘-fkeep-inline-functions’.” [GNU98]

Redundant code. When a container uses certain components (i.e., `qualify`, `top2ds_qualify`, or `inbetween`) P2 generates redundant code, especially for predicate testing. For example, consider the source code shown in Figure 6.8 (Original) for a cursor with predicate `"$.name != 'Bob'"` and type expression `C` (see Figure 6.1). The code generated from this consists of many conditional and iterative statements involving boolean expressions as shown in Figure 6.8 (No Optimizations). Upon examination of the generated code, we often find that these expressions can be simplified, or these statements optimized-away using domain-independent optimizations such as constant propagation, double negation elimination, etc. [Har95] as shown in Figure 6.8 (Domain-Independent Optimizations). We hope that the C compiler will be smart enough to perform these optimizations. Other optimizations can only be achieved by application of higher-level, domain-specific (e.g., container and cursor) semantics such as the optimization shown in Figure 6.8 (Domain-Specific Optimizations) that an `adv()` followed by a `rev()` has no net effect. The C compiler will *not* be able to perform these optimizations. Both the domain-independent and domain-specific optimizations would result in significant decreases in code size and increases in application performance.

6.2 GenVoca

6.2.1 Realms

Our experience with P2 has confirmed what we knew already about decomposability and domain analysis. It has also confirmed the usefulness of the established realms `LINK`, `TOP`, `DS`, and `MEM`, and added to our knowledge the new realms `XACT` and `PROCESS`.

Original

```
reset_start(cursor);
adv(cursor);
rev(cursor);
```

No optimizations

```
// reset_start(cursor);
cursor.obj = (faculty_container).first2;
while (!(cursor.obj == 0) && !(strcmp (cursor.obj->name, “Bob”) != 0))
  if (!(cursor.obj == 0))
    if (cursor.inbetween && strcmp (cursor.obj->name, “Bob”) != 0)
      cursor.inbetween = 0;
    else
      cursor.obj = cursor.obj->next2;
// adv(cursor);
if (!(cursor.obj == 0))
  if (cursor.inbetween && strcmp (cursor.obj->name, “Bob”) != 0)
    cursor.inbetween = 0;
  else
    cursor.obj = cursor.obj->next2;
while (!(cursor.obj == 0) && !(strcmp (cursor.obj->name, “Bob”) != 0))
  if (!(cursor.obj == 0))
    if (cursor.inbetween && strcmp (cursor.obj->name, “Bob”) != 0)
      cursor.inbetween = 0;
    else
      cursor.obj = cursor.obj->next2;
// rev(cursor);
if (cursor.inbetween && (cursor.obj == 0))
  cursor.obj = (faculty_container).last2;
else
  cursor.obj = cursor.obj->prev2;
cursor.inbetween = 0;
while (!(cursor.obj == 0) && !(strcmp (cursor.obj->name, “Bob”) != 0)) {
  if (cursor.inbetween && (cursor.obj == 0))
    cursor.obj = (faculty_container).last2;
  else
    cursor.obj = cursor.obj->prev2;
  cursor.inbetween = 0;
}
```

Domain-Independent Optimizations

```
// reset_start(cursor);
cursor.inbetween = 0;
cursor.obj = (faculty_container).first2;
while (cursor.obj != 0 && strcmp(cursor.obj->name, “Bob”) == 0)
  cursor.obj = cursor.obj->next2;
// adv(cursor);
if (cursor.obj != 0)
  cursor.obj = cursor.obj->next2;
while (cursor.obj != 0 && strcmp(cursor.obj->name, “Bob”) == 0)
  cursor.obj = cursor.obj->next2;
// rev(cursor);
cursor.obj = cursor.obj->prev2;
while (cursor.obj != 0 && strcmp (cursor.obj->name, “Bob”) == 0)
  cursor.obj = cursor.obj->prev2;
```

Domain-Specific Optimizations

```
// reset_start(cursor);
cursor.inbetween = 0;
cursor.obj = (faculty_container).first2;
while (cursor.obj != 0 && strcmp(cursor.obj->name, “Bob”) == 0)
  cursor.obj = cursor.obj->next2;
// adv(cursor); rev(cursor); => empty statement
```

Figure 6.8 Example generated code.

Even before we began P2, we knew that GenVoca generators are most successful in mature domains [Bat92]. This is because generators automate software construction, and it is difficult if not impossible, to automate anything that is not already well-understood. Before implementing a GenVoca generator, one must perform an appropriate domain analysis, yielding high-level abstractions, standardized interfaces, and layered decompositions. This domain analysis had already been performed by the implementors of Genesis and P1 for the `LINK`, `TOP`, `DS`, and `MEM` realms. Our experience with P2 further confirmed the appropriateness of these realms.

Our experience with P2 leads us to suggest some minor changes to the `LINK`, `TOP`, `DS`, and `MEM` realms. For example, the name of the `delete()` operation is poorly chosen, since it conflicts with the delete operator in C++; if we were to re-implement P2, we would rename it, possibly to `remove()`. But, for the most part, with the addition of only a few new operations, these realms proved appropriate for hundreds of applications including LEAPS and Smallbase. These additional operations were mostly for performance or convenience. For example, the `insertv()` and `allocv()` operations are new to P2, having been added to support the **`varchar`** data type, which is also new to P2. But the **`varchar`** data type is really just an optimization; we could simulate the functionality of **`varchar`** by always inserting a record of the largest possible size, although this would impose a significant performance penalty. Likewise, the `cardinality()` operation is new to P2, but is also just an optimization; we could simulate the functionality of `cardinality()` by manually maintaining a count of the number of elements in the container, although this would be much less convenient.

The only significant change to the realms that we can suggest is that operations should be capable of taking a variable number of arguments. This capability is needed to conveniently and efficiently support multiple versions of an operation. Multiple versions of an operation are needed when additional features are added. P2 currently provides multiple versions of several operations. For example, consider the insertion operation. In addition to the standard `insert()` operation, P2 currently provides the `insertv()` operation to support the varchar data type feature. And, we can easily envision many more features for

the insertion operation. For example, P1 provided a version of the insertion operation that allowed new elements to be positioned at the beginning of the container, end of the container, before the cursor, or after the cursor. There are three ways to provide multiple versions of an operation. (1) Provide a different operation for each version. This is the solution P2 currently uses. The problem with this approach is that the number of operations is exponential in the number of features (the feature combinatorics). It is inconvenient for component implementors to maintain more than one version of an operation, especially when the operation appears in a large number of components. We already experienced this problem with the `insert()` and `insertv()` operation. (2) Have the operation take a large number of arguments, and have the user select different versions via these arguments. The problems with this approach are that it is inconvenient for users to specify a large number of arguments, and decoding the arguments incurs increased runtime overhead. (3) Allow operations to take a variable number of arguments. This is convenient for both component implementors and users, and does not incur any runtime overhead.

The design and validation of the new realms XACT and PROCESS is one of the primary research contributions of our work; these realms were essential for successfully reengineering Smallbase. Unfortunately, due to limitations of `xp` (see Section 6.1.3), P2 does not implement either of these realms as cleanly as they should be implemented–the entire PROCESS realm and several operations in XACT were implemented by hand, without using xp. But ignoring the limitations of `xp`, we believe that our analysis of the XACT and PROCESS domains was done correctly, and our experiments involving TPC-B have validated this result. Future GenVoca generators with component specification facilities that are more powerful than `xp` should be able to incorporate the XACT and PROCESS realms with little trouble.

6.2.2 Components

Because P2 provides so many components, it gave us a unique opportunity to observe the software engineering costs and benefits of GenVoca.[9] One of the most interesting observations we made is that components seem easier to validate than non-componentized soft-

ware. We observed that, ignoring errors introduced by bugs in `xp`, the density of errors in component code is roughly the same as that in hand-written application. Furthermore, we found that components can be more easily and thoroughly unit-tested than non-componentized software. Plug-compatibility, the same feature that makes components ease to reuse, also makes them easy to validate. For example, P2 provides over 100 regression tests which new components can be plugged-into merely by changing a type expression in a header file. This allows new components to be easily tested in variety of situations, thoroughly exercising the component code, and revealing errors that might otherwise remain latent. For example, these regression tests found an error in the `red_black_tree` component that had remained latent in the analogous component in DiSTiL, which has a less extensive set of regression tests.

6.2.3 Generators

P2 is not the largest GenVoca generator, even in the domain of databases (Genesis is about the same size as P2). But P2 does produce the fastest executables of any generator in this domain. In fact, P2 produces executables that are several orders of magnitude faster than heavyweight systems such as Genesis (see Chapter 4) and at least as fast as hand-coded lightweight systems such as Smallbase (see Chapter 5). Thus, P2 proves that generators can scale to generate complex, high performance systems while preserving the reuse benefits of increased programmer productivity.

That it would be possible to achieve such high performance was not at all obvious when we began our work. A key concern was that composing locally optimal code fragments does not imply that their composition will be optimal. Even before we began, we knew that some things would necessarily be sub-optimal versus hand-coding. For example, we knew that P2 would generate unnecessary conditional and iterative statements with boolean expressions involving predicates, as discussed in Section 6.1.4. Despite these potential performance problems, however, the actual performance delivered by P2 is very

---

9. Of course, more rigorous testing is needed to quantify and validate these observations.

good. Thus the performance of P2 versus hand-coded systems is analogous to the performance of high-level languages (such as C) versus assembly language. In high level languages, it is known that some things are necessarily sub-optimal versus assembly language. Despite these potential performance problems, however, the actual performance delivered by high level languages is very good. The reason for this is that these potential performance rarely arise in practice. In fact, the large productivity gains afforded by high level languages allow the programmer to worry about high level (i.e. global, algorithmic) rather than low level (i.e. local, instruction choice) optimizations. The end result is that programs written in a high level language often use more sophisticated algorithms and thus have *better* performance than analogous programs written in assembly language. This is analogous to how programs written in P2 (e.g. RL using hash-joins) often use more sophisticated algorithms and thus have better performance than analogous programs written by hand (e.g. LEAPS).

6.3 Lightweight DBMSs

LWDBs are clearly an idea whose time has come. The interest in lightweight systems is evident in the buzz surrounding data structure template libraries, persistent stores, object oriented DBMSs, main-memory optimized DBMSs, extensible (or open) DBMSs, and universal servers (see Chapter 2). Lightweight systems can offer huge performance advantages versus heavyweight systems, and huge productivity and performance advantages versus hand-coded data manipulation functionality. In particular, P2 has shown orders of magnitude performance improvements versus heavyweight systems, and a factor of three productivity and orders of magnitude performance improvement versus hand coded data manipulation functionality. There are, however, problematic issues with lightweight systems that are currently inhibiting their widespread adoption. These include limits to the performance improvement that their use can offer and software engineering challenges inherent to their construction and use. Fortunately, GenVoca techniques give us leverage to overcome these difficulties, and P2 is a demonstration of a system that does so.

6.3.1 Performance Improvement Limits

Two factors limit the performance improvement that an application will realize from using a lightweight system: (1) the fraction of the application that can be enhanced, and (2) of that fraction, how much it can be enhanced. Both of these limiting factors are best understood in terms of Amdahl's law, which is shown in Figure 6.9.

$$\mathtt{Speedup}_{\mathtt{overall}} = \frac{1}{(1 - \mathtt{Fraction}_{\mathtt{enhanced}}) + \dfrac{\mathtt{Fraction}_{\mathtt{enhanced}}}{\mathtt{Speedup}_{\mathtt{enhanced}}}}$$

Figure 6.9 Amdahl's Law.

Fraction that can be enhanced. A DBMS application sees a performance benefit from a lightweight (or any enhanced) system in proportion to the fraction of time it spends in data manipulation, $\mathtt{Fraction}_{\mathtt{enhanced}}$. An application that spends only a small fraction of its time in data manipulation code will realize only a small performance benefit, no matter how much the enhanced system can speed-up that code, $\mathtt{Speedup}_{\mathtt{enhanced}}$. An application such as LEAPS that spends a large fraction of its time in data manipulation code, on the other hand, will realize a large performance benefit.

How much that fraction can be enhanced. A lightweight system achieves high performance by omitting and/or specializing features. An application that uses many features and does not allow many features to be specialized will not realize much performance benefit, $\mathtt{Speedup}_{\mathtt{enhanced}}$, from using a lightweight system. Some features are particularly expensive. For example, consider the TPC-B benchmark of Chapter 5. When the benchmark was implemented using no logging, the indexing strategy was the major factor in execution time. When this benchmark was implemented with the synchronous logging feature, this cost of synchronous logging dominated the cost of execution to a very large extent, and specializations to the indexing strategy were largely irrelevant.

6.3.2 Software Engineering Challenges

There are several software engineering challenges inherent to the construction and use of lightweight systems: (1) feature combinatorics, (2) difficulty of customization, (3) non-standard interfaces, and (4) economic and societal issues. These difficulties are in addition to the quite formidable difficulties inherent in to the construction of DBMSs in general:

- *Feature combinatorics*. The feature combinatorics problem for lightweight systems is the same as the feature combinatorics problem for software libraries: different systems embody different combinations of features. An exponential number of different systems must be available in order to exactly satisfy the needs of every possible application.
- *Difficulty of customization.* Extensible (or open) DBMSs, and universal servers attempt to solve the feature combinatorics problem by allowing users to add extensions as required. For the highest possible performance, however, an LWDB must be exactly matched to the needs of the target application. Often, the only person who know these needs is the implementor of the application. Thus, LWDBs must be customizable by application programmers. Application programmers will customize an LWDB only if it is very easy to do so. They are not often prepared for the challenge of implementing an extension. A typical extension may consist of hundreds or thousands of lines of code. And, of course, even when this work is done, the result of extending a heavyweight system is still a heavyweight system, and it still suffers the performance problems of generality.
- *Non-standard interfaces*. Implementors of any system, including data management systems, tend to create an API that is specific to the particular functionality provided by the system. This is very natural, and everyone does it. But this has disastrous consequences for the user–switching to a different system requires re-writing the application in terms of the new interface. Users often get stuck with the first system they choose, rather than being able to experiment with various systems in order to find the one that offers the best performance. In the data management domain, users are aware of the non-standard interfaces problem and other non-compatibility

problems and thus tend to choose the system with the most *features*, rather than the best *performance*, in order to avoid the problem of choosing another system and rewriting their application in case it "outgrows" their original system. Universal servers are the ultimate expression of this problem, providing systems that can grow with the application.

- *Economic and societal issues*. The greatest difficulties with the construction and use of lightweight systems are economic and societal. The construction of a DBMS requires a huge investment. For example, P2 represents approximately ten years of programmer time, yet it is a research system, consisting of only one hundred thousand lines of code. Commercial DBMSs comprise millions of lines of code, and hundreds of years of programmer time. Since a DBMS represents such a huge investment, its cost must be amortized by marketing to as many customers and for as many applications as possible. Each potential customer or application demands a single DBMS that can provide all these features, and the union of all these sets of features is a very heavyweight system. Thus, these economic and social issues cause the construction and use of increasingly heavyweight systems.

6.3.3 GenVoca and P2 Solutions

The LWDB domain is ideally suited to GenVoca techniques[10]. The DBMS domain is very mature and inherently yields the high-level abstractions, standardized interfaces, and layered decompositions necessary for GenVoca generator construction. And GenVoca techniques are ideally suited to the LWDB domain. GenVoca techniques give us leverage to overcome the software engineering challenges inherent to the construction and use of lightweight systems. P2 serves as a demonstration of a system that does so:

10. Some researchers have suggested that, other than network protocols (the Voca in GenVoca), their are precious few domains to which GenVoca is so ideally suited. But, this research is solely concerned with the LWDB domain, so we do not attempt to refute them here.

- *Feature combinatorics*. The GenVoca solution to the feature combinatorics problem for lightweight systems is the same as the solution for software libraries: encapsulate exactly one feature per component and combine components to generate systems with the desired features. In this way, a linear number of components is all that is needed to generate an exponential number of systems.
- *Difficulty of customization.* The GenVoca mechanism of feature specification via type expressions greatly ameliorates the difficulty of customization problem. Type expressions permit the extreme ease of customization needed by application programmers. A typical type expression consists of a single line of code. The supporting tools that we developed, such as design rule checkers, graphical user interfaces, and design wizards [Bat98b] make customization even easier. Of course, the only customizations that can be accomplished via type equations are those that use existing components, so GenVoca techniques do not completely eliminate the difficulty of customization problem. But P2 provides a fairly large library of existing components, and this greatly reduces the chances that users will have to write new components. In the eventuality that users will have to write new components, P2 provides mechanisms (`xp`) that greatly simplify component construction.
- *Non-standard interfaces*. GenVoca components have standardized interfaces (that is, the realms `XACT`, `PROCESS`, `LINK`, `TOP`, `DS`, and `MEM`). All the lightweight systems that P2 can generate use these standard interfaces. All the lightweight systems that P2 can generate comprise a very large family of systems. Thus, P2 goes a long way toward ameliorating the non-standard interfaces problem, but it does not solve it. P2 cannot generate *all* systems. Other lightweight systems use different interfaces and P2 is introducing yet another interface. Important future work is to attempt to provide for P2 a more industry standard interface, such as SQL. Since SQL is such a large language, implementing a full SQL to P2 translator was beyond our research interests.

- *Economic and societal issues*. P2 is a significant advance in understanding of light-weight systems and their construction. Unfortunately, P2 is a research prototype, not a product. P2 has the problems discussed in this chapter and many known (and, no doubt, unknown) bugs. Few programmers are prepared to implement a GenVoca LWDB product, since it requires the rare combination of knowledge of both GenVoca techniques and DBMS internals. Most programmers with knowledge of GenVoca techniques are software engineering researchers; and most DBMS researchers are concerned with implementing new features (parallel, distributed, multimedia), not software engineering techniques for re-packaging old ones. Thus, a GenVoca LWDB product is not likely to be developed anytime soon. And few DBMS application programmers are brave enough to use a research prototype. But as shown in Chapters 4 and 5, P2 offers compelling performance and productivity advantages for those users brave enough to try it.

Chapter 7

Conclusions

This dissertation described the P2 lightweight DBMS generator. This chapter reviews the central results of our experimental work, summarizes the primary contributions of our research, and discusses a few areas of future research and enhancement to P2.

7.1 Results

We draw the following conclusions from the results of our experiments building P2 and comparing it to the LEAPS compilers and Smallbase LWDB:

- *Performance*. GenVoca generators can provide better runtime performance than hand-coding. Our experiments with LEAPS and Smallbase showed that P2 produced code with runtime performance at least as good as hand-written code when using analogous algorithms and data structures, and up to several orders of magnitude faster when using more advanced algorithms and data structures.
- *Productivity*. GenVoca generators can provide better programmer productivity than hand-coding. Our experiments with LEAPS showed that P2 reduced development time and code size by a factor of three, relative to hand-coding.
- *Scale*. GenVoca generators can scale to very large systems without sacrificing performance or productivity. In our experiments with Smallbase, we were able to generate all the features needed by the modified TPC-B benchmark without suffering a decrease in runtime performance or programmer productivity.

7.2 Contributions

Our work makes the following contributions:

- *LWDB Principles.* Developed a comprehensive theory of LWDBs. We unified a class of disparate systems that are related by the fact that they all achieve high-performance in an analogous manner, and coined the term *lightweight DBMS* to describe this class of systems. See Section 2.3.
- *LWDB Construction.* Demonstrated that GenVoca generation is well suited to LWDB construction. Before P2, there were no formalizations, tools, or architectural support for LWDB construction and thus LWDBs were hand-crafted monolithic systems that were expensive to build and tune. With P2, LWDBs are much easier to build and tune. See Section 6.3.
- *XACT and PROCESS.* Carefully analyzed the LWDB domain, developed the XACT and PROCESS realms. See Sections 5.2 and 5.3.
- *TOP, DS, and MEM.* Exercised the data structures and link domains, validated the LINK, TOP, DS, and MEM realms. See Sections 3.1, 3.2, and 4.1.
- *Component Definition Language.* Demonstrated advantages of and requirements for component definition languages: ease of use and modification; single, unified language for component definition and use; facility for specifying default implementations of operations. See Section 6.1.3.
- *DRC.* Provided a vehicle for design rule checking (DRC) research that researchers [Bat97a] could use for testing concepts and implementations of DRC. Learned that DRC should be tightly integrated with the component definition facilities and that DRC cannot determine the correctness of type expressions without considering their actual usage. See Section 6.1.2.
- *Lock Manager.* Implemented a lock manager using the P2 data language. This permitted the lock manager described in Gray and Reuter [Gra93b] to be specified cleanly and concisely; and the P2 LWDB allowed efficient code to be generated from this specification. The code given in [Gra93b], combines specification (con-

tainer of lock requests and a container of lock headers) and implementation (hash table, singly-linked list). P2 allowed the specification and implementation to be cleanly separated (the specification via cursor and container operations, and implementation via type expressions). See Section 5.3.1 and the Appendix.

7.3 Future Research

Future work will look at how GenVoca generators can be scaled to even larger sizes, while further increasing programmer productivity and generated code performance.

- *Components.* Since LWDBs achieve high performance by specializing the implementation of features to meet the needs of individual applications, building additional components will yield systems with even higher performance. Building more components will also allow us to determine if the existing P2 interfaces are general enough to allow a variety of feature implementations.
- *Component Definition Language.* The advantages of and requirements for component definition languages that P2 has demonstrated have already lead to work on generation scoping in DiSTiL [Sma96-97] and JTS [Bat98a].
- *Formal Semantics.* We have not attempted a formal specification of operation semantics, since we have an informal idea of operation semantics that was adequate for our purposes. But, a formal specification would be useful for future P2 users and implementors, and invaluable for further automation such as improved DRC and automatic selection of type expressions.
- *DRC.* More work is needed to more tightly integrate DRC with component definition facilities and to determine the correctness of type expressions from actual usage.
- *Type Expressions.* A primary advantage of GenVoca generators relative to other approaches to reducing the difficulty of software construction is their high degree of automation. One aspect that has not yet been automated, but seems amenable to

automation is the selection of type expressions. It seems likely that by combining a cost model with improved DRC, automated selection of type expressions would be straightforward [Bat98b].

- *Code Size*. The code generated by P2 is much too large. Our work has concentrated almost entirely on optimizing generated code performance. Nevertheless, other properties of generated code also are important, particularly with the proliferation of embedded devices. Despite the fact that P2 generated LWDBs are typically smaller than comparable hand-written systems (e.g. LEAPS and Smallbase), we feel that the code generated by P2 could be made much more compact.
- *Experiments*. Many more experiments are needed. We have no illusions that our experiments with LEAPS and Smallbase are exhaustive. The problem is that reengineering systems the size of LEAPS and Smallbase requires a great deal of work, both to develop applications using cursor and container operations and to implement any necessary components.
- *Dynamic Optimization*. P2 currently does as much as possible statically, at compile-time. This has the advantages of reducing run-time overhead and being easier to implement, but the disadvantages of requiring all queries to be known statically, and not being able to use dynamic information such as relation size or predicate selectivity. An interesting experiment would be to see if run-time query optimization, perhaps even with run-time selection of components, could be added to P2 for those applications that require this functionality without adding run-time overhead for those that don't.
- *SQL*. Adding an SQL front-end to P2 would provide a useful tool, but poses significant research and engineering challenges. Perhaps the best advantage to having such a tool would be that it would allow P2 to support many existing applications without a large reengineering effort per application. The biggest research challenge might be to preserve the performance advantages of P2 in the face of the generality entailed by SQL (e.g., atomic retrieval semantics don't allow lazy retrieval implementation). The biggest engineering challenge would be the sheer size of the SQL language.

Appendix

The P2 Log Manager

Following is the P2 log manager, expressed as a P2 program adapted from [Gra93b]. Note that we have omitted the symbol name prefix P2_ from the discussion in the previous chapters (for perspicuity), but have included it here (for completeness).

```
/* $Id: P2_log-manager.p2,v 45.46 1998/01/20 08:37:00 jthomas Exp $ */
/* Copyright (C) 1998, The University of Texas at Austin. */

/* Recovery log table manager. */

/************************************************************************/
/*                  Includes.                                           */
/************************************************************************/

#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#include <assert.h>    /* assert() */
#include <stdarg.h>    /* va_list */

#include "op-id.h"     /* P2_LOG_BEGIN_XACT_OP */
#include "print-log.h" /* P2_PRINT_LOG */
#include "util.h"      /* P2_MIN */
#if 1
#include "round.h"     /* P2_ROUND_UP */
#endif

#include "P2_paces.h"  /* BOOLEAN */

/************************************************************************/
/*                  Defines.                                            */
/************************************************************************/
```

```
/* To store the log anchor, use a container structure (preferred), rather
   than a container with a single element (useful for debugging). */
#if 1
#if !defined(LOG_ANCHOR_CONTAINER_STRUCTURE)
#define LOG_ANCHOR_CONTAINER_STRUCTURE
#endif /* LOG_ANCHOR_CONTAINER_STRUCTURE */
#endif

/* P2_TRACE. */
#if 0
#if !defined(P2_TRACE)
#define P2_TRACE
#endif /* P2_TRACE */
#endif

/**********************************************************************/
/*                  Log and Anchor.                                   */
/**********************************************************************/

#if 1
/* Normal case: use large log_size, for efficiency. */
/* For release/distribution: ~8 MB (1 MB = 1048576) */
#define LOG_SIZE 8000000
#else
/* For debugging: use small log_size, so boundary conditions get
   exercised. */
#define LOG_SIZE 10000
#endif

/* sizeof(element<log_anchor{,_cont})):
     Size of (transformed) log anchor {,container} element. */
#define SIZEOF_ELEMENT_LOG_ANCHOR_CONT sizeof(P2_LOG_ANCHOR)

typex {
  /* If you change log_typex, make sure
     paces/P2_log-manager.h:P2_LOG_STRUCT_DUMMY_SIZE represents the
     total size of fields added by xform. */
  /* The queue layer guarantees that elements will be ordered from
     oldest to newest.  This is useful for printing the log, or
     performing a system restart. */
  log_typex = top2ds[container_structure[
                      slist_queue[malloc[mmap_persistent]]]];

#if defined(LOG_ANCHOR_CONTAINER_STRUCTURE)
  log_anchor_typex = top2ds[init_cont_function[container_structure[
                      mmap_persistent]]];
#else
  log_anchor_typex = top2ds[init_cont_function[array[
                      mmap_persistent]]];
#endif /* LOG_ANCHOR_CONTAINER_STRUCTURE */
}
```

```
/* Log file index. */

/* This is stored in mmap memory to make sure this process has the
   most recent log file in mmap memory. */
typedef struct {
  P2_LSN_INDEX index;
} LOG_CONT_INDEX;

container <P2_LOG_STRUCT> stored_as log_typex with {
  container_structure "LOG_CONT_INDEX";
  mmap_persistent file is "/tmp/P2_log-data"
    with size LOG_SIZE;
} log_cont;

container <P2_LOG_ANCHOR> stored_as log_anchor_typex with {
  init_cont_function "P2_init_log_anchor";
#if defined(LOG_ANCHOR_CONTAINER_STRUCTURE)
  container_structure "P2_LOG_ANCHOR";
#else
  array size is 1;
#endif /* LOG_ANCHOR_CONTAINER_STRUCTURE */
  mmap_persistent file is "/tmp/P2_log-anchor-data"
    with size SIZEOF_ELEMENT_LOG_ANCHOR_CONT;
} log_anchor_cont;

#if defined(LOG_ANCHOR_CONTAINER_STRUCTURE)
#define log_anchor log_anchor_cont
#endif /* LOG_ANCHOR_CONTAINER_STRUCTURE */

cursor <log_cont> log_curs;
#if !defined(LOG_ANCHOR_CONTAINER_STRUCTURE)
cursor <log_anchor_cont> log_anchor;
#endif /* LOG_ANCHOR_CONTAINER_STRUCTURE */

/************************************************************************/
/*                  Global Variables.                                   */
/************************************************************************/

/*** These declarations are examples of overhead added by generality. ***/
/*** These are initialized in x/log.xph:init_schema() ***/

/* TRUE iff the log manger layer hash been initialized. */
int P2_log_manager_initialized = 0;
/* Durable or non-durable transaction log manager.
/* (Do or do not flush LOG at commit). */
int P2_log_manager_sync;
/* (Do or do not flush log HEADER at commit). */
int P2_log_manager_anchor_sync;

/************************************************************************/
```

```
/*                   P2_lsncmp.                                          */
/***********************************************************************/

#ifndef NDEBUG

/* HACK!!! */
/* Analogous to intcmp and strcmp (see paces/P2.h). */
/* Only used in assertions. */

int
P2_lsncmp (P2_LSN lsn1, P2_LSN lsn2)
{
#if 0
  /* Faster?  gcc won't let us cast a struct to a long. */
  assert(sizeof(P2_LSN) == sizeof(long));
  if ((long) lsn1 < (long) lsn2)
    return(-1);
  else if ((long) lsn1 > (long) lsn2)
    return(1);
  return(0);
#else
  /* Slower?  Field-by-field comparison. */
  if (lsn1.index < lsn2.index
      || (lsn1.index == lsn2.index && lsn1.rba < lsn2.rba))
    return(-1);
  else if (lsn1.index > lsn2.index
      || (lsn1.index == lsn2.index && lsn1.rba > lsn2.rba))
    return(1);
  assert(lsn1.index == lsn2.index && lsn1.rba == lsn2.rba);
  return(0);
#endif
}

#endif

/***********************************************************************/
/*                   Miscellaneous (Local) Procedures.                   */
/***********************************************************************/

/* Operation identifier validity test. */

static BOOLEAN
op_id_valid (P2_OP_ID op_id)
{
  /* Make sure the operation identifier is in the range 0 ... MAX_OP_ID */
  if (op_id < 0 || op_id > P2_MAX_OP_ID)
    return(FALSE);
  return(TRUE);
}

/* Maximum LSN. */
```

```
static P2_LSN
max_lsn (P2_LSN lsn1, P2_LSN lsn2)
{
  if (lsn1.index > lsn2.index)
    return(lsn1);
  else if (lsn2.index > lsn1.index)
    return(lsn2);
  else if (lsn1.rba > lsn2.rba)
    return(lsn1);
  return(lsn2);
}

#ifndef NDEBUG

/* LSN validity test. */

static BOOLEAN
lsn_valid (P2_LSN lsn)
{
  /* Make sure that the given lsn is less than or equal to the LSN of
     the next record. That is, (lsn <= log_anchor.prev_lsn) */
  /* Can we have a stronger test? */
  return(P2_lsncmp(lsn, log_anchor.lsn) <= 0);
}

/* Log record validity test. */

BOOLEAN
P2_log_struct_valid (P2_LOG_STRUCT *x)
{
  /* Make sure x looks like a valid pointer. */
  if (x == NULL || ((unsigned) x) < 8)
    return(FALSE);

  /*** All operations. ***/

  /* Make sure the LSNs are valid. */
#if defined(P2_LOG_STRUCT_LSN)
  if (!lsn_valid(x->lsn))
    return(FALSE);
#endif /* P2_LOG_STRUCT_LSN */
#if defined(P2_LOG_STRUCT_PREV_LSN)
  if (!lsn_valid(x->prev_lsn))
    return(FALSE);
#endif /* P2_LOG_STRUCT_PREV_LSN */
  if (!lsn_valid(x->xact_prev_lsn))
    return(FALSE);

#if defined(P2_LOG_STRUCT_XACT_ID)
  /* If the operation is a cursor operation, then make sure xact_id
```

```
     is valid */
  if ((x->op_code & P2_LOG_NON_CURS_OP) == 0)
    if (!P2_xact_id_valid(x->xact_id))
      return(FALSE);
#endif /* P2_LOG_STRUCT_XACT_ID */

  /* Make sure op_id is valid) */
  if (!op_id_valid(x->op_code & P2_LOG_OP))
    return(FALSE);

  /*** Update operation. ***/

  /* If the operation is an update operation, then make sure it's also
     a cursor operation. */
  if ((x->op_code & P2_LOG_STR_UPD_OP) != 0)
    if ((x->op_code & P2_LOG_NON_CURS_OP) != 0)
      return(FALSE);

  /*** Return TRUE. ***/

  return(TRUE);
}

int
P2_curs_id_valid (P2_CURS_ID curs_id)
{
  return(curs_id > 0 && curs_id < 10000);
}

int
P2_cont_id_valid (P2_CONT_ID cont_id)
{
  return(cont_id > 0 && cont_id < 10000);
}

#endif /* NDEBUG */

/************************************************************************/
/*                  Initialize Relative Byte Address.                   */
/************************************************************************/

/* Return the relative byte address of the 1st log record. */
/* Initialize x and call insertv() explicitly, rather than have
   P2_log_insert() do it for us implicitly. */

static P2_LSN_RBA
init_rba (void)
{
  P2_LSN_RBA r;
  P2_XACT_LOG_STRUCT x;
  int total_size = sizeof(x);
```

```
  assert(total_size >= 0);

  /* Initialize x. */
  /* Is there a more efficient way of doing this, for example,
     make x static, and use an initializer??? (JAT) */
  /* Is this necessary???  Doesn't P2_log_insert() initialize x? (JAT) */
  /* See analogous code in P2_xact_manager_checkpoint_schema() */
#if 1
  assert(((int) sizeof(P2_XACT_LOG_STRUCT)) <= total_size);
  memset(&x, 0, sizeof(P2_XACT_LOG_STRUCT));
#else
#if defined(P2_LOG_STRUCT_LSN)
  x.lsn = P2_NULL_LSN;
#endif /* P2_LOG_STRUCT_LSN */
#if defined(P2_LOG_STRUCT_PREV_LSN)
  x.prev_lsn = P2_NULL_LSN;
#endif /* P2_LOG_STRUCT_PREV_LSN */
#if defined(P2_LOG_STRUCT_XACT_ID)
  x.xact_id = 0;
#endif /* P2_LOG_STRUCT_XACT_ID */
  x.xact_prev_lsn = P2_NULL_LSN;
#endif
  x.op_code = P2_LOG_INIT_RBA_OP | P2_LOG_OTHER_OP;
  /* Insert x. */
  assert(!overflowv(log_cont, total_size));
  insertv(log_curs, x, total_size, total_size);
  /* Compute RBA. */
  /* Force integer, rather than pointer (aka address), arithmetic. */
  r = (P2_LSN_RBA) (((unsigned) log_curs.obj) + total_size);
  return(r);
}

/**********************************************************************/
/*                   Open log file.                                   */
/**********************************************************************/

static void
open_log_cont (P2_LSN_INDEX index)
{
#if defined(P2_TRACE)
  P2_trace("*** open log file ***");
#endif /* P2_TRACE */

  /* Open new log file.  Must do this before we initialize log_curs. */
  open_cont_number(log_cont, index);
  /* Set log file index. */
  log_cont.index = index;
  /* Initialize log_curs. */
  init_curs(log_curs);
}
```

```
static void
close_and_open_log_cont (P2_LSN_INDEX index)
{
  /* Close old log file. */
  close_cont(log_cont);
  /* Open new log file. */
  open_log_cont(index);
}

/**********************************************************************/
/*                  Sync Log and Checkpoint.                          */
/**********************************************************************/

/* Checkpoint. */

void
P2_log_manager_checkpoint_schema (void)
{
  P2_LSN base;
  P2_LSN top;

  base.index = log_anchor.prev_lsn.index;
  if (log_anchor.prev_lsn.index != log_anchor.persist_lsn.index)
  {
    /* Special case: previous checkpoint was a different container. */
    /* base is the beginning of the container. */
    base.rba = 0;
  }
  else
    /* Normal case: previous checkpoint was the current container.  */
    /* base is the previous checkpoint. */
    base.rba = log_anchor.persist_lsn.rba;

  /* top is the end of the log. */
  top = log_anchor.prev_lsn;
  /* Make sure top RBA is zero, or top RBA is base RBA or higher. */
  assert(top.rba == 0 || top.rba >= base.rba);

  /* Synchronize log file. */
  sync_cont(log_cont, base.rba, top.rba, P2_log_manager_sync);

  /* Synchronize log anchor file. */
  if (P2_log_manager_anchor_sync)
    sync_cont(log_anchor_cont, 0, 0, 1);

  /* Start with current durable log as best lsn.  Gray & Reuter, p. 608 */
  P2_low_water = max_lsn(P2_low_water, top);

#if defined(P2_LOG_ANCHOR_STRUCT_PERSIST_LSN)
  /* Update LSN of max (most recent) log record in persistent store. */
```

```
  log_anchor.persist_lsn = P2_low_water;
#endif /* P2_LOG_ANCHOR_STRUCT_PERSIST_LSN */
}

/* Synchronize log file. */

/* Note that P2_log_manager_sync_log() is currently equivalent to
   P2_log_manager_checkpint_schema(), but this is not guaranteed to
   be the case in the future. (JAT) */

void
P2_log_manager_sync_log (void)
{
  P2_log_manager_checkpoint_schema();
}

/***********************************************************************/
/*                  Log insert.                                        */
/***********************************************************************/

/* Return the log record with the given LSN. */
/* Gray & Reuter, p. 501 */
/* Called by paces/P2_xact-manager.p2:P2_init_xact_cb_cont() */

P2_LOG_STRUCT *
P2_log_read_lsn (P2_LSN lsn)
{
  assert(log_cont.index == log_anchor.index);
  if (log_anchor.index != lsn.index)
    close_and_open_log_cont(lsn.index);
#ifndef NDEBUG
  assert(P2_log_struct_valid((P2_LOG_STRUCT *) lsn.rba));
#endif /* NDEBUG */
  return((P2_LOG_STRUCT *) lsn.rba);
}

/* Allocate a new log record and copy the given data into it.
   Copy fixed_size bytes from x, and for each of the narg
   (v, variable_size) pairs in the argument list, copy variable_size
   bytes from v. */

/* The narg (v, variable_size) pairs in the argument list are an
   optimization used, for example, by the insert(), delete(), and
   upd() operations.  Without this optimization, we would pass the
   entire log record in x. The problem with doing so, however, is that
   these operations would then have to copy data twice: once from its
   original location into x, and once from x into the log. */

/* Gray & Reuter, pp. 506-507 */

P2_LSN
```

```
P2_log_insert (P2_LOG_STRUCT *x, unsigned fixed_size, int narg, ...)
{
  char *v;
  unsigned variable_size;
  unsigned total_size = fixed_size;
  P2_LSN lsn;

  assert(narg >= 0);
  if (narg != 0)
  {
    va_list ap;
    int i;

    va_start(ap, narg);
    for (i = 0; i < narg; i++)
    {
       v = va_arg(ap, char *);
       variable_size = va_arg(ap, int);
       total_size += variable_size;
    }
    va_end(ap);
#if 1
    total_size = P2_ROUND_UP(total_size, 4);
#endif
  }

  /* Assertions. */
  /* Make sure the log record is not too small. */
  assert(total_size >= (int) sizeof(P2_XACT_LOG_STRUCT));
  /* Make sure the log record is not too big to fit into ANY log file. */
  assert(total_size <= LOG_SIZE);

#if defined(P2_LOG_MANAGER_LOCK)
  /* Aquire the log lock. */
  /* Lock the log end in exclusive mode. */
  if (P2_lock_semaphore(&(log_anchor.lock)) != 0)
    P2_runtime_error(
      "P2_log-manager: P2_log_insert:"
      " Could not lock log manager mutex lock");
#endif /* P2_LOG_MANAGER_LOCK */

#if defined(P2_PROCESS_UNIPROCESS)
#ifndef NDEBUG

  /*** Uniprocess: log_cont.index should always == log_anchor.index. ***/

  if (log_cont.index != log_anchor.index)
    P2_runtime_error("log_cont.index = %d != log_anchor.index = %d",
                     log_cont.index, log_anchor.index);
#endif /* NDEBUG */
#else
```

```
  /*** Multiprocess: Make sure this process has correct log file. ***/

  /* When another process closes an old log file and opens a new log file,
     this process (sometimes?) retains mappings from the old log file. */
  if (log_cont.index != log_anchor.index)
  {
#ifndef NDEBUG
    /* I included this warning only to see if this code is EVER
       executed.  This condition is in no way anomalous or erroneous,
       it is just rare. (JAT) */
    P2_runtime_warning(
      "P2_log-manager: P2_log_insert:"
      " log_cont.index = %d != log_anchor.index = %d",
      log_cont.index, log_anchor.index);
#endif /* NDEBUG */
    close_and_open_log_cont(log_anchor.index);
  }
#endif /* P2_PROCESS_UNIPROCESS */

  /* Make sure the log record is not too big to fit into THIS log
     file, otherwise, close the old log file and open a new one.
     Since this might change the record's LSN, we must do it before we
     copy the LSN. */
  if (overflowv(log_cont, total_size))
  {
    /* Synchronize log file. */
    P2_log_manager_sync_log();

    /* Increment sequence number of new log file. */
    log_anchor.index++;

    /* Close old log file and open new log file. */
    close_and_open_log_cont(log_anchor.index);

#if defined(P2_LOG_ANCHOR_STRUCT_PERSIST_LSN)
    /* Make sure that (log_anchor.persist_lsn <= log_anchor.lsn) */
#ifndef NDEBUG
    assert(P2_lsncmp(log_anchor.persist_lsn, log_anchor.prev_lsn) <= 0);
#endif /* NDEBUG */
    /* Update LSN of max (most recent) log record in persistent store. */
    log_anchor.persist_lsn = log_anchor.prev_lsn;
#endif /* P2_LOG_ANCHOR_STRUCT_PERSIST_LSN */

    /* Update LSN of next record. */
    log_anchor.lsn.index = log_anchor.index;
    log_anchor.lsn.rba = init_rba();
  }

  /* Make a copy of the record's LSN. */
  lsn = log_anchor.lsn;
```

```
  assert(lsn_valid(lsn));

  /* Fill-in log record header. */
#if defined(P2_LOG_STRUCT_LSN)
  x->lsn = lsn;
#endif /* P2_LOG_STRUCT_LSN */
#if defined(P2_LOG_STRUCT_PREV_LSN)
  x->prev_lsn = log_anchor.prev_lsn;
  assert(lsn_valid(x->prev_lsn));
#endif /* P2_LOG_STRUCT_PREV_LSN */
#if defined(P2_LOG_STRUCT_XACT_ID)
  x->xact_id = P2_get_xact_id();
#endif /* P2_LOG_STRUCT_XACT_ID */
  /* Allow logging of special operations which are executed outside of
     a transaction.  This allows, for example, init_cont() to be
     executed before begin_xact().  */
  /* This if statement is an example of overhead added by generality. */
  /* Q: Is this a good idea?
     A: No, because it adds un-necessary overhead to the common case.
     A: Yes, because it allows more flexibility in user programs.
        And type expressions (log can be used without xact)??? */
  if (P2_get_xact_id() == 0)
    x->xact_prev_lsn = P2_NULL_LSN;
  else
  {
    x->xact_prev_lsn = P2_log_transaction(lsn);
    assert(lsn_valid(x->xact_prev_lsn));
  }

#ifndef NDEBUG
  /* Make sure that the log record passes the validity test. */
  assert(P2_log_struct_valid(x));
#endif /* NDEBUG */

  /* Insert. */
#if defined(P2_TRACE)
  P2_trace("pre-insert: log_curs.obj = %x", log_curs.obj);
  P2_trace("pre-insert: log_anchor.lsn.index = %d", log_anchor.lsn.index);
  P2_trace("pre-insert: log_anchor.lsn.rba = %x", log_anchor.lsn.rba);
#endif /* P2_TRACE */

  insertv(log_curs, *x, total_size, fixed_size);
  if (narg > 0)
  {
    va_list ap;
    int i;
    va_start(ap, narg);
    for (i = 0; i < narg; i++)
    {
      v = va_arg(ap, char *);
      variable_size = va_arg(ap, unsigned);
```

```
      memcpy((char *) log_curs.obj + fixed_size, v, variable_size);
      fixed_size = fixed_size + variable_size;
    }
    va_end(ap);
  }

#if defined(P2_TRACE)
  P2_trace("post-insert: log_curs.obj = %x", log_curs.obj);
  P2_trace("post-insert: log_anchor.lsn.index = %d",
log_anchor.lsn.index);
  P2_trace("post-insert: log_anchor.lsn.rba = %x", log_anchor.lsn.rba);
#endif /* P2_TRACE */

  /* Cursor and log_anchor.lsn should point to the object we just
     inserted. */
  /* Make sure the LSN is correct. */
  assert(log_anchor.lsn.rba == log_curs.obj);
  /* Make sure the non-variable fields match. */
  /* It is necessary to cast offsetof() (an unsigned) to int, because
     if we don't cast it to int, and we get a warning when we compare
     it to total_size (an int) (JAT) */
  assert(memcmp(log_curs.obj, x,
    P2_MIN(total_size, (int) offsetof(P2_LOG_STRUCT, v))) == 0);
#if 0
  /* Make sure the varchar field matches??? */
#endif

  /* Update anchor. */
  log_anchor.prev_lsn = lsn;
  log_anchor.lsn.rba += total_size;
#if defined(P2_TRACE)
  P2_trace("total_size = %x", total_size);
  P2_trace("post-update: log_anchor.lsn.rba = %x", log_anchor.lsn.rba);
#endif /* P2_TRACE */

#if defined(P2_LOG_MANAGER_LOCK)
  /* Release the log lock. */
  /* Unlock the log end. */
  if (P2_unlock_semaphore(&(log_anchor.lock)) != 0)
    P2_runtime_error(
      "P2_log-manager: P2_log_insert:"
      " Could not unlock log manager mutex lock");
#endif /* P2_LOG_MANAGER_LOCK */

  /* Return LSN of record just inserted. Needed by
       paces/P2_xact-manager.p2:P2_xact_manager_checkpoint_schema() */
  return(lsn);
}

/**********************************************************************/
/*                   Print log.                                       */
```

```
/**********************************************************************/

#if defined(P2_PRINT_LOG)

#include <stdio.h> /* printf() */

void
P2_print_log_anchor (void)
{
  printf(“Log anchor:\n”);
  printf(“  lock                    = %d\n”, log_anchor.lock);
  printf(“  index                   = %d\n”, log_anchor.index);
#if defined(P2_LOG_STRUCT_LSN)
  printf(“  lsn                     = %u, %x\n”,
         log_anchor.lsn.index, log_anchor.lsn.rba);
#endif /* P2_LOG_STRUCT_LSN */
#if defined(P2_LOG_STRUCT_PREV_LSN)
  printf(“  prev_lsn                = %u, %x\n”,
         log_anchor.prev_lsn.index, log_anchor.prev_lsn.rba);
#endif /* P2_LOG_STRUCT_PREV_LSN */
  printf(“  xact_manager_anchor_lsn = %u, %x\n”,
         log_anchor.xact_manager_anchor_lsn.index,
         log_anchor.xact_manager_anchor_lsn.rba);
#if defined(P2_LOG_ANCHOR_STRUCT_PERSIST_LSN)
  printf(“  persist_lsn             = %u, %x\n”,
         log_anchor.persist_lsn.index, log_anchor.persist_lsn.rba);
#endif /* P2_LOG_ANCHOR_STRUCT_PERSIST_LSN */
}

/* Might want to add validity checks, such as a computation of the
   logical size of the log record, which we can compare with the
   actual size. */
/* This function has type P2_LOG_STRUCT_FUNCTION. */

int
P2_print_log_struct (P2_LOG_STRUCT *x, unsigned log_struct_number)
{
  printf(“Record %u:\n”, log_struct_number);

  /*** Fields for all operations (including transactions) ***/

#if defined(P2_LOG_STRUCT_LSN)
  printf(“  lsn                     = %u, %x\n”,
             x->lsn.index, x->lsn.rba);
#endif /* P2_LOG_STRUCT_LSN */
#if defined(P2_LOG_STRUCT_PREV_LSN)
  printf(“  prev_lsn                = %u, %x\n”,
             x->prev_lsn.index, x->prev_lsn.rba);
#endif /* P2_LOG_STRUCT_PREV_LSN */
#if defined(P2_LOG_STRUCT_XACT_ID)
  printf(“  xact_id                 = %u\n”, x->xact_id);
```

```
#endif /* P2_LOG_STRUCT_XACT_ID */
  printf("  xact_prev_lsn           = %u, %x\n",
             x->xact_prev_lsn.index, x->xact_prev_lsn.rba);
  printf("  op_code                 = %x ", x->op_code);

  /*** Operation-specific fields. ***/

  /* Other (e.g., init_rba) operation. */
  if ((x->op_code & P2_LOG_OTHER_OP) != 0)
  {
    P2_OP_ID op_id = (x->op_code & P2_LOG_OP);
    printf("(other operation)\n");
    if (op_id == P2_LOG_INIT_RBA_OP)
      printf("  op_name                 = init_rba\n");
    else if (op_id == P2_LOG_XACT_MANAGER_ANCHOR_OP)
      printf("  op_name                 = xact_manager_anchor\n");
    else
      printf("  op_name                 = (unknown other operation)");
  }
  else
  {
    /* Container operation. */
    if ((x->op_code & P2_LOG_CONT_OP) != 0)
    {
      P2_CONT_LOG_STRUCT *k = (P2_CONT_LOG_STRUCT *) x;
      P2_OP_NAME_VEC *op_name_vec = P2_get_op_name_vec(k->cont_id);
      printf("(container operation)\n");
      if ((*op_name_vec) == NULL)
        printf("  op_name_vec             = NULL\n");
      else
      {
        P2_OP_ID op_id;
        char *op_name;
        op_id = (log_curs.op_code & P2_LOG_OP);
        if (!op_id_valid(op_id))
          P2_runtime_error(
            "P2_log-manager: P2_print_log_struct: container"
            " operation identifier invalid: log corrupted?");
        op_name = (*op_name_vec)[op_id];
        printf("  op_name_vec             = %x\n", (*op_name_vec));
        printf("  op_name                 = %s\n", op_name);
        printf("  cont_id                 = %d\n", k->cont_id);
        printf("  cont_obj_id             = %x\n", k->cont_obj_id);
      }
    }
    /* Schema (aka transaction) operation. */
    else if ((x->op_code & P2_LOG_SCHEMA_OP) != 0)
    {
      /* The id of schema is currently always zero. */
      P2_OP_NAME_VEC *op_name_vec = P2_get_op_name_vec(0);
      printf("(schema operation)\n");
```

```
      if ((*op_name_vec) == NULL)
        printf("  op_name_vec              = NULL\n");
      else
      {
        P2_OP_ID op_id;
        char *op_name;
        op_id = (log_curs.op_code & P2_LOG_OP);
        if (!op_id_valid(op_id))
          P2_runtime_error("P2_log-manager: P2_print_log_struct: schema"
                         " operation identifier invalid: log corrupted?");
        op_name = (*op_name_vec)[op_id];
        printf("  op_name_vec              = %x\n", (*op_name_vec));
        printf("  op_name                  = %s\n", op_name);
      }
    }
    /* Cursor operation. */
    else
    {
      P2_CURS_LOG_STRUCT *c = (P2_CURS_LOG_STRUCT *) x;
      P2_OP_NAME_VEC *op_name_vec = P2_get_op_name_vec(c->curs_id);
      printf("(cursor operation)\n");
      if ((*op_name_vec) == NULL)
        printf("  op_name_vec              = NULL\n");
      else
      {
        P2_OP_ID op_id;
        char *op_name;
        op_id = log_curs.op_code;
        if (!op_id_valid(op_id))
          P2_runtime_error(
            "P2_log-manager: P2_print_log_struct: cursor"
            " operation identifier invalid: log corrupted?");
        op_name = (*op_name_vec)[op_id];
        printf("  op_name_vec              = %x\n", (*op_name_vec));
        if (op_name[0] != '\0')
          printf("  op_name                  = %s\n", op_name);
        else
          printf("  op_name                  = upd %s\n",
                 ((log_curs.op_code & P2_LOG_STR_UPD_OP) != 0)?
                 "str" : "int");
      }
      printf("  curs_id                  = %d\n", c->curs_id);
      printf("  obj_id                   = %x\n", c->obj_id);
    }
  }

  /*** Always return 0. ****/

  return(0);
}
```

```
void
P2_print_log (void)
{
  P2_foreach_log_struct(P2_NULL_LSN,
    (P2_LOG_STRUCT_FUNCTION *) &P2_print_log_struct);
}

#endif /* P2_PRINT_LOG */

/***********************************************************************/
/*                   Foreach log struct.                               */
/***********************************************************************/

/* For each log record starting with the given lsn, call the given
   function.  That is, map the function f over the records of the log
   starting at the given lsn. */

/* This function exists, because this functionality (1) is needed in
   several places, (2) it is sufficiently complex that proceduralizing
   it improves understandability and maintainability, and (3) the
   places where it is used are the uncommon case, so performance is
   not important enough to justify inlining it. (JAT) */

/* This function is analogous to P2_undo_xact() */

int
P2_foreach_log_struct (P2_LSN lsn, P2_LOG_STRUCT_FUNCTION *f)
{
  P2_LSN_INDEX index = lsn.index;
  P2_LOG_STRUCT *x;
  unsigned log_struct_number = 0;

  close_and_open_log_cont(index);
  if (lsn.rba == 0)
  {
    reset_start(log_curs);
  }
  else
  {
    pos(log_curs, lsn.rba);
  }

  while (index <= log_anchor.lsn.index)
  {
    if (end_adv(log_curs))
    {
      index++;
      close_and_open_log_cont(index);
      reset_start(log_curs);
    }
    else
```

```
    {
      /* Return value. */
      int r;
      /* Get the log record. */
      P2_LOG_STRUCT *x = (P2_LOG_STRUCT *) log_curs.obj;
#ifndef NDEBUG
      /* Check to log record for validity. */
      if (!P2_log_struct_valid(x))
        P2_runtime_error("P2_log-manager: P2_foreach_log_struct:"
                         " log struct invalid: log corrupted?");
#endif /* NDEBUG */
      /* Increment the log record number. */
      log_struct_number++;
#if 0
      /* Print log record before we undo it. */
#if defined(P2_PRINT_LOG)
      if (P2_get_xact_status() == P2_XACT_REDO)
      {
        printf("REDO: ");
        P2_print_log_struct(x, log_struct_number);
      }
#endif /* P2_PRINT_LOG */
#endif
      assert(f != NULL);
      /* Call the given function. */
      r = (*f)(x, log_struct_number);
      /* If the return value of f is non-0, return the value f returned. */
      if (r != 0)
        return(r);
      /* Advance to next log record. */
      adv(log_curs);
    }
  }
  /* Return 0. */
  return(0);
}

/*************************************************************************/
/*                  Transaction operations.                              */
/*************************************************************************/

/* Convert LSN to pointer to log record. */

static P2_LOG_STRUCT *
lsn_to_log_struct (P2_LSN lsn)
{
  P2_LOG_STRUCT *x;

  /* Make sure that lsn is valid. */
  assert(lsn_valid(lsn));
  /* Cast LSN as a pointer to P2_LOG_STRUCT. */
```

```
  x = (P2_LOG_STRUCT *) lsn.rba;
#ifndef NDEBUG
  /* Make sure that log record is valid. */
  assert(P2_log_struct_valid(x));
#endif /* NDEBUG */
  /* Return log record. */
  return(x);
}

/* Undo or redo the most recent transaction. */
/* This function is analogous to P2_foreach_log_struct() */
/* It is called by x/log.xp:abort_xact() */

void
P2_undo_xact (P2_LOG_STRUCT_FUNCTION *f)
{
  P2_LSN lsn;
  P2_LOG_STRUCT *x;
  P2_LSN_INDEX index = log_anchor.index;

  assert(log_cont.index == log_anchor.index);
  assert(P2_get_xact_status() == P2_XACT_UNDO
         || P2_get_xact_status() == P2_XACT_REDO);

  /* Undo transaction. */
  lsn = P2_get_max_lsn();
  while (!P2_null_lsn(lsn))
  {
    assert(lsn.rba != 0);
    assert(lsn_valid(lsn));
    assert(lsn.index <= index);

    /* If this log record is not in the log file currently in memory,
       then bring in the (old) log file containing the record. */
    if (lsn.index != index)
    {
      index = lsn.index;
      close_and_open_log_cont(index);
    }

    x = lsn_to_log_struct(lsn);

    assert(f != NULL);
#ifndef NDEBUG
    /* Call the undo or redo function. */
    assert((*f)(x) == 0);
#else
    /* Call the undo or redo function. */
    (*f)(x);
#endif /* NDEBUG */
```

```
#ifndef NDEBUG
    /* assert(x->xact_prev_lsn < lsn) */
    assert(P2_lsncmp(x->xact_prev_lsn, lsn) < 0);
#endif /* NDEBUG */

    lsn = x->xact_prev_lsn;
  }

  /* Close the old log file and bring in the current log file. */
  if (index != log_anchor.index)
  {
    close_and_open_log_cont(log_anchor.index);
  }

#if 0
  P2_init_xact_cb(P2_XACT_NONE);
#endif
}

/**********************************************************************/
/*                 Read and write transaction manager anchor.         */
/**********************************************************************/

/* "Each resource manager can write a checkpoint in the log,
   consisting of an ordinary log record containing the resource
   manager's restart information.  The resource manager then registers
   its anchor LSN and low-water LSN with the transaction manager.  The
   transaction manager, in turn, records this information in the
   transaction manager checkpoint record.  Hence, the system only
   needs to remember the location of the transaction manager's
   checkpoint record.  As shown in Figure 9.6, it is recorded as a
   field of the log manager's anchor record." Gray & Reuter, p. 512 */

/* Read LSN of transaction manager anchor from log anchor.
   Gray and Reuter, call this log_read_anchor(), p. 512 */

P2_LSN
P2_log_read_anchor (void)
{
  assert(lsn_valid(log_anchor.xact_manager_anchor_lsn));
  return(log_anchor.xact_manager_anchor_lsn);
}

/* Save LSN of transaction manager anchor in log anchor.
   Gray & Reuter, call this log_write_anchor(), p. 512 */

void
P2_log_write_anchor (P2_LSN lsn)
{
  assert(lsn_valid(lsn));
  log_anchor.xact_manager_anchor_lsn = lsn;
```

```
}

/**********************************************************************/
/*                  Restart.                                          */
/**********************************************************************/

/* This routine is called from x/log.xp:verbatim_s():P2_warm_restart(). */

/* The warm restart functionality is really part of the transaction
   manager, and thus should be defined in the file
   paces/P2_xact-manager.p2.  Unfortunately, we need to make reference
   to the log record structure, which is defined in the log manager,
   which must be layered on TOP of the transaction manager. (JAT) */

void
P2_log_manager_warm_restart (P2_LOG_STRUCT_FUNCTION *f)
{
  /* P2 warm restart. */
  P2_LSN lsn;
  P2_XACT_MANAGER_ANCHOR_LOG_STRUCT *x;

#ifndef NDEBUG
  P2_runtime_warning(
    "P2_log-manager: P2_log_manager_warm_restart:"
    " restarting from log");
#endif /* NDEBUG */

  /* Set status to REDO, so that operations are not logged. */
  P2_set_xact_status(P2_XACT_REDO);

#if 1
  /* Start from low water mark. */
  /* Get low water mark from the transaction manager anchor. */
  lsn = P2_log_read_anchor();
  /* Check for amnesia. Can't do a warm restart if there is no log file. */
  /* If there is no log file, we know that x will be zero (and not
     some random, un-initialized value), because when
     P2_init_log_manager() open the log_anchor container, it will call
     P2_init_log_anchor() if the container is un-initialized. */
  if (lsn.rba == 0)
    P2_runtime_error(
      "P2_log-manager: P2_log_manager_warm_restart:"
      " no log file found");
  x = (P2_XACT_MANAGER_ANCHOR_LOG_STRUCT *) P2_log_read_lsn(lsn);
  P2_foreach_log_struct(x->low_water, f);
#else
  /* Start from P2_NULL_LSN */
  P2_foreach_log_struct(P2_NULL_LSN, f);
#endif

  if (log_cont.index != log_anchor.index)
```

```
  {
#ifndef NDEBUG
    /* I included this warning only to see if this code is EVER
       executed.  This condition is in no way anomalous or erroneous,
       it is just rare. (JAT) */
    P2_runtime_warning(
      "P2_log-manager: P2_log_manager_warm_restart:"
      " log_cont.index = %d != log_anchor.index = %d",
      log_cont.index, log_anchor.index);
#endif /* NDEBUG */
    close_and_open_log_cont(log_anchor.index);
  }
}

/**********************************************************************/
/*                  Init and delete log manager.                      */
/**********************************************************************/

void
P2_init_log_anchor (void)
{
  /* Amnesia. */

#if defined(P2_LOG_MANAGER_LOCK)
  /* Initialize log lock. */
  if (P2_init_semaphore(&log_anchor.lock, 1) != 0)
    P2_runtime_error(
      "P2_log-manager: P2_init_log_anchor:"
      " Could not initialize log manager mutex lock");
#endif /* P2_LOG_MANAGER_LOCK */

  /* See analogous code in P2_log_insert(). */

  /* Initialize log_anchor.index.
     Must do this before we open log_cont??? */
  log_anchor.index                    = 0;
  log_anchor.lsn                      = P2_NULL_LSN;
  log_anchor.prev_lsn                 = P2_NULL_LSN;
  log_anchor.xact_manager_anchor_lsn = P2_NULL_LSN;
#if defined(P2_LOG_ANCHOR_STRUCT_PERSIST_LSN)
  log_anchor.persist_lsn              = P2_NULL_LSN;
#endif /* P2_LOG_ANCHOR_STRUCT_PERSIST_LSN */
}

void
P2_init_log_manager (void)
{
  /* Make sure SIZEOF_ELEMENT_LOG_ANCHOR_CONT is correct. */
  assert(SIZEOF_ELEMENT_LOG_ANCHOR_CONT
         == sizeof(element<log_anchor_cont>));
```

```
  /* Make sure P2_CURS_LOG_STRUCT is correct. */
  /* That is, make sure that curs_id--the first field of
     P2_CURS_LOG_STRUCT after the dummy field (the beginning of the
     variable part of P2_CURS_LOG_STRUCT)--has the same offset as
     v--the varchar field of the transformed element (the beginning of
     the variable part of the transformed element). */
  assert(offsetof(P2_CURS_LOG_STRUCT, curs_id)
         == offsetof(element<log_cont>, v));

  /* Make sure P2_LOG_STRUCT_DUMMY_SIZE is correct. */
  /* That is, make sure that P2_LOG_STRUCT_SIZE--the size of the dummy
     field--accurately represents the size of the fields added by
     xform--the difference between the size of the transformed element
     (element <log_cont>) and the original element (P2_LOG_STRUCT) */
  assert(P2_LOG_STRUCT_DUMMY_SIZE
         == (sizeof(element <log_cont>) - sizeof(P2_LOG_STRUCT)));

  /* Make sure P2_SIZEOF_ELEMENT_LOG_CONT is correct. */
  assert(P2_SIZEOF_ELEMENT_LOG_CONT == sizeof(element<log_cont>));

#if defined(LOG_ANCHOR_CONTAINER_STRUCTURE)
  open_cont(log_anchor);
#else
  P2_LOG_ANCHOR a;
  open_cont(log_anchor_cont);
  init_curs(log_anchor);
  /* If we are not using a container structure, then might be better
     (stylistically) to initialize the struct a before inserting it,
     but it isn't necessary, and it would require different
     initialization code when we are using a container structure. */
  insert(log_anchor, a);
#endif /* LOG_ANCHOR_CONTAINER_STRUCTURE */

#if 0
  /* BUG!!! Do NOT want to always initialize the log anchor. */
  /* Want to initiailize it only if it is un-initialized. */
  /* When do we initiate recovery??? */
  P2_init_log_anchor();
#else
  /* Open new log file. */
  open_log_cont(log_anchor.index);
  /* If log_anchor.lsn is uninitialized, initialize it.
     We assume it is uninitialized if its rba is zero. */
  /* A more exact test for log_anchor.lsn uninitialized would be
     P2_lsncmp(log_anchor.lsn, P2_NULL_LSN), but this test would be
     more expensive. (JAT) */
  /* See analogous code in
     P2_log-manager.p2:P2_log_manager_warm_restart() */
#ifndef NDEBUG
  /* Assert that the two tests are equivalent. */
  assert((log_anchor.lsn.rba == 0)
```

```
         == (P2_lsncmp(log_anchor.lsn, P2_NULL_LSN) == 0));
#endif /* NDEBUG */
  if (log_anchor.lsn.rba == 0)
  {
    /* Amnesia!!! */
#ifndef NDEBUG
    P2_runtime_warning(
      "P2_log-manager: P2_init_log_anchor:"
      " Amnesia: no log file found");
#endif /* NDEBUG */
    /* Index should be 0 */
    assert(log_anchor.index == 0);
    log_anchor.lsn.index = log_anchor.index;
    /* Relative byte address is address of 1st log record. */
    log_anchor.lsn.rba = init_rba();
  }
#endif

#if 0
  /* Initialize transaction control block. */
  P2_init_xact_cb(P2_XACT_NONE);
#endif
}

void
P2_delete_log_manager (void)
{
#if 0
  /* Delete xact and lock managers. */
  P2_delete_xact_manager();
#endif

#if 0
  /* Never delete the log lock--it will be stored persistently in the
     log anchor. */
#if defined(P2_LOG_MANAGER_LOCK)
  /* Delete log lock. */
  if (P2_delete_semaphore(&(log_anchor.lock)) != 0)
    P2_runtime_error(
      "P2_log-manager: P2_delete_log_manager:"
      " Could not delete log manager mutex lock");
#endif /* P2_LOG_MANAGER_LOCK */
#endif

#if 1
  /* Synchronize log file. */
  P2_log_manager_sync_log();
#endif
  /* Close log file. */
  close_cont(log_cont);
```

```
  /* Close log anchor. */
  close_cont(log_anchor_cont);
}
```

Bibliography

[Ara91] Guillermo Arango and Ruben Prieto-Diaz. Part 1: Introduction and Overview Domain Analysis Concepts and Research Directions. In [Pri91].

[Bal85] Robert Balzer. A 15 Year Perspective on Automatic Programming. In *IEEE Transactions on Software Engineering*, November 1985, pages 1257-1268.

[Bat88a] D. S. Batory. Concepts for a Database System Compiler. In *Proceedings of the Seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Austin, Texas, March 21-23 1988, ACM Press, pages 184-192. Also in [Pri91], pages 250-257.

[Bat88b] D. S. Batory, J. R. Barnett, J. F. Garza, K. P. Smith, K. Tsukuda, B. C. Twichell, and T. E. Wise. GENESIS: An extensible database management system. In *IEEE Transactions on Software Engineering*, November 1988. Also in [Zdo90].

[Bat92a] Don Batory and Sean O'Malley. The Design and Implementation of Hierarchical Software Systems With Reusable Components. In *ACM Transactions on Software Engineering and Methodology*, 1(4):355-398, October 1992.

[Bat92b] Don Batory, Vivek Singhal, and Jeff Thomas. Database Challenge: Single Schema Database Management Systems. Technical Report 92-47, Department of Computer Sciences, University of Texas at Austin, February 1992.

[Bat93] Don Batory, Vivek Singhal, Marty Sirkin, and Jeff Thomas. Scalable Software Libraries. In *ACM SIGSOFT*, December 1993.

[Bat94a] Don Batory. *The LEAPS Algorithms*. Technical Report 94-28, Department of Computer Sciences, University of Texas at Austin, 1994.

[Bat94b] Don Batory, Jeff Thomas, and Marty Sirkin. Reengineering a Complex Application Using a Scalable Data Structure Compiler. In *ACM SIGSOFT*, December 1994.

[Bat94c] Don Batory, Vivek Singhal, Jeff Thomas, Sankar Dasari, Bart Geraci, and Marty Sirkin. The GenVoca Model of Software-System Generators. In *IEEE Software*, September 1994.

[Bat95] Don Batory, Lou Coglianese, Mark Goodwin, and Steve Shafer. Creating Reference Architectures: An Example from Avionics, In [Sam95], pages 27-37.

[Bat97a] Don Batory and Bart J. Geraci. Composition Validation and Subjectivity in GenVoca Generators. In *IEEE Transactions on Software Engineering (Special Issue on Software Reuse)*, February 1997, pages 67-82.

[Bat97b] Don Batory and Jeff Thomas. P2: A Lightweight DBMS Generator. In *Journal of Intelligent Information Systems*, Kluwer Academic Publishers, September/October 1997, pages 107-123.

[Bat98a] Don Batory, Bernie Lofaso, and Yannis Smaragdakis, JTS: Tools for Implementing Domain-Specific Languages. In *Proceedings of the Fifth International Conference on Software Reuse*, Victoria, Canada, June 1998.

[Bat98b] Don Batory, Gang Chen, Eric Robertson, and Tao Wang, Web-Advertised Generators and Design Wizards. In *Proceedings of the Fifth International Conference on Software Reuse*, Victoria, Canada, June 1998.

[Bax92] Ira D. Baxter. Design Maintenance Systems. In *Communications of the ACM*, April 1992, pages 73-89.

[Bax94] Ira D. Baxter. Design (Not Code!) Maintenance. In *Proceedings of VIII Brazilian Symposium on Software Engineering (SBES)*, Curitiba, Brazil, October 25-28, 1994.

[Bax96] Ira D. Baxter. *Tutorial on Transformation Systems*, Presented at the Fourth International Converence on Software Reuse, Orlando Florida, April 23, 1996.

[Bax97a] Ira D. Baxter and Christopher W. Pidgeon. Software Change Through Design Maintenance. In *Proceedings of International Conference on Software Maintenance '97*, Bari, Italy, September 28-October 2 1997. Also in http://www.semdesigns.com/

[Bax97b] Ira D. Baxter and Michael Mehlich. Reverse Engineering is Reverse Forward Engineering. In *Proceedings of Fourth Working Conference on Reverse Engineering*, Amsterdam, The Netherlands, October 6-8 1997. Also in http://www.semdesigns.com/

[Big87] Ted Biggerstaff and Charles Richter. Reusability Framework, Assessment, and Directions. In *IEEE Software*, July 1987, pages 41-49. Also in Will Tracz, editor, *Software Reuse: Emerging Technology*, IEEE Computer Society Press, 1988, pages 3-11. Also in [Big89], pages 1-17.

[Big89] Ted J. Biggerstaff and Alan J. Perlis, editors. *Software Reusability, Volume 1: Concepts and Models*, ACM Press, 1989.

[Big92] Ted J. Biggerstaff. An Assessment and Analysis of Software Reuse. In *Advances in Computers*, Volume 34, Academic Press, 1992.

[Big94] Ted Biggerstaff. The Library Scaling Problem and the Limits of Concrete Component Reuse. In [Fra94].

[Big97] Ted Biggerstaff. A 15 Year Perspective of Reuse and Generation. In *Proceedings of the Knowledge-Based Software Engineering Conference*, Syracuse, New York, September 1996.

[Bir73] Graham Birstwistle, Ole-Johan Dahl, Bjorn Myrhaug, Kristen Nygaard. *Simula Begin*. Auerbach, Philadelphia PA, 1973.

[Bod94] Francois Bodin, Peter Beckman, Dennis Gannon, Jacob Gotwals, Srinivas Narayana, Suresh Srinivas, and Beata Winnicka. Sage++: An Object-Oriented Toolkit and Class Library for Building Fortran and C++ Restructuring Tools. In *OON-SKI '94: Proceeedings of the Second Annual Object-Oriented Numerics Conference*, Sunriver, Oregon, April 1994.

[Boo87] Grady Booch. Software Components with Ada, Benjamin/Cummings, Menlo Park, California, 1987.

[Boo98] Grady Booch, Jim Rumbaugh, and Ivar Jacobson. *Unified Modeling Language User Guide*, Addison-Wesley, Reading, Massachusetts, 1998.

[Boy89] James M. Boyle. Abstract Programming and Program Transformation. In [Big89], pages 361-413.

[Bra91] David A. Brant, Timothy Grose, Bernie Lafaso, and Daniel P. Miranker. Effects of Database Size on Rule System Performance: Five Case Studies. In *Proceedings of the Seventh International Conference of Very Large Data Bases*, Barcelona, September 1991, pages 287-296.

[Bra93a] David Brant and Daniel P. Miranker. Index Support for Rule Activation. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, Washington, DC, May 1993.

[Bra93b] David Brant. *Inferencing on Large Data Sets*, Ph.D. dissertation, Department of Computer Sciences, The University of Texas at Austin, 1993.

[Bro86] Frederick P. Brooks, Jr. No Silver Bullet: Essence and Accidents of Software Engineering. In H. J. Kugler, editor, *Information Processing '86*, Elsevier Science Publishers B. V. (North-Holland), 1986. Also in *IEEE Computer*, April 1987, pages 10-19. Also in [Bro95a].

[Bro95a] Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition*, Addison-Wesley, Reading, Massachusetts, 1995, pages 179-203.

[Bro94] James C. Browne, Allen Emerson, Mohamed G. Gouda, Daniel P. Miranker, Aloysius Mok, Lance Obermeyer, Furman Haddix, Rwo-hsi Wang, and Sarah Chodrow. Modularity and Rule Based Programming. In *International Journal on Artificial Intelligence Tools*, Volume 4, Numbers 1 and 2, June 1995, World Scientific.

[Bro95b] James C. Browne, Syed I. Hyder, Jack Dongarra, Keith Moore, Peter Newton, Visual Programming and Debugging for Parallel Computing, In *IEEE Parallel and Distributed Technology: Systems and Applications*, Volume 3, Number 1, Spring 1995, pages 75-83.

[But97] David R. Butenhof. *Programming with POSIX Threads*, Addison-Wesley, 1997.

[Car90] M. J. Carey, D. J. DeWitt, G. Graefe, D. M. Haight, J.E. Richardson, D. T. Schuh, E. J. Shekita, and S. L. Vandenberg. The EXODUS Extensible DBMS Project: An Overview. In [Zdo90].

[Cha96] David Chappell. *Understanding ActiveX and OLE*, Microsoft Press, Redmond, Washington, 1996.

[Che83] Thomas E. Cheatham, Jr. Reusability Through Program Transformations. In *Workshop on Reusability in Programming*, Newport, Rhode Island, September 1993. Also in *IEEE Transactions on Software Engineering*, September 1984, pages 589-595. Also in [Big89], pages 321-335.

[Cla86] Billy Claybrook, Dick Huber, Gary Fernandez, Harriet Goodpaster, John Marien, Steve Sherry, and Craig Smelser. *Design & Implementation of a Lock Manager*, Wang Institute of Graduate Studies, Tyngsboro, Massachusetts, 1986.

[Cle88] J. C. Cleaveland. Building application generators. In *IEEE Software*, July 1988, pages 25-33. Also in [Pri91], pages 241-249.

[Cod95] Code Farms Incorporated. *C++ Data Object Library–Data structures combined with automatic persistence*, Code Farms Incorporated, 7214 Jock Trail, Richmond, Ontario, K0A 2Z0, Canada, 1995.

[Coh93] Donald Cohen and Neil Campbell. Automating Relational Operations on Data Structures. In *IEEE Software*, May 1993, pages 53-60.

[Coo88] Thomas Cooper and Nancy Wogrin. *Rule-based Programming with OPS5*, Morgan Kaufmann, San Mateo, California, 1988.

[Cor95] J.R. Cordy, I.H. Carmichael and R. Halliday. T*he TXL Programming Language - Version 8*, Software Technology Laboratory, Dept. of Computing and Information Science, Queen's University, Kingston, Canada K7L 3N6, and Legasys Corporation, Suite 530, 366 King St. E., Kingston, Canada K7K 6Y3. Also ftp://ftp.qucis.queensu.ca/pub/txl/8.0/TextDocs/TXL_Lang_v8.txt.

[Dav89] Jack W. Davidson and David B. Whalley. Quick Compilers Using Peephole Optimisation. In *Software - Practice and Experience*, Volume 19, Number 1, January 1989, pages 79-97.

[DeR76] F. DeRemer and H. J. Kron, Programming-in-the-large versus programming-in-the-small. In *IEEE Transactions on Software Engineering*, June 1976, pages 80-86.

[Edi98] Edison Design Group. *C++ Front End: Internal Documentation*, Edison Design Group, Upper Montclair, New Jersey, 1998.

[Ell90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*, Addison-Wesley, Reading, Massachusetts, 1990.

[Exo94] *EXODUS Database Toolkit*. ftp://ftp.cs.wisc.edu/exodus/, Computer Sciences Department, University of Wisconsin Madison, 1994.

[Fea87] Martin S. Feather. A Survey and Classification of some Program Transformation Approaches and Techniques. In L. G. L. T. Meertens, editor, *Program Specification and Transformation: Proceedings of the IFIP TC2/WG 2.1 Working Conference on Program Specification and Transformation*, Bad Tolz, Germany, April 1987, pages 165-195.

[Fra94] William B. Frakes, editor, *Proceedings of the Third International Conference on Software Reuse: Advances in Software Reusability*, Rio de Janeiro, Brazil, November 1994.

[For82] Charles Forgy. *OPS5 User's Manual*. Technical Report, CMU-CS-81-135, School of Computer Science, Carnegie Mellon University, 1981.

[For94] Ira R. Forman, Scott Danforth, and Hari Madduri. Composition of Before/ After Metaclasses in SOM. In *OOPSLA '94 Conference Proceedings: Object-Oriented Programming Systems, Languages and Applications*, Portland, Oregon, October 1994.

[Gam95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*, Addison-Wesley, Reading, Massachusetts, 1995.

[GNU98] *Using and Porting GNU CC*, Version 2.8.1, 1998.

[Gol89] Adele Goldberg and David Robson. *Smalltalk 80: The Language*, Addison-Wesley, Reading, Massachusetts, 1989.

[Gog86] Joseph Goguen. Reusing and interconnecting software components. In *IEEE Computer*, February 1986, pages 16-28. Also in [Pri91], pages 125-137.

[Gos96] James Gosling, Bill Joy, Guy L. Steele. *The Java Language Specification*, Addison-Wesley, Reading, Massachusetts, 1996.

[Gra93a] Jim Gray, editor, *The Benchmark Handbook For Database and Transaction Processing Systems, Second Edition*, Morgan Kaufmann, San Francisco, California, 1993.

[Gra93b] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, San Mateo, California, 1993.

[Gri94] Martin L. Griss and Kevin D. Wentzel. Hybrid Domain-Specific Kits for a Flexible Software Factory. In *Proceedings of the 1994 ACM Symposium on Applied Computing (ACM SAC '94), Reuse and Reengineering Track*, Phoenix, Arizona, March 1994, pages 47-52.

[Haa90] Laura M. Haas, Walter Chang, Guy M. Lohman, John McPherson, Paul F. Wilms, George Lapis, Bruce G. Lindsay, Hamid Pirahesh, Michael J. Carey, Eugene J. Shekita. Starburst Mid-Flight: As the Dust Clears, In *IEEE Transactions on Knowledge and Data Engineering*, pages 143-160.

[Hal96] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion and M. S. Lam. Maximizing Multiprocessor Performance with the SUIF Compiler. In *IEEE Computer*, December 1996.

[Har92] William Harrison, Mansour Kavianpour, and Harold Ossher. Integrating Coarse-grained and Fine-Grained Tool Integration. In *Proceedings of the Fifth Workshop on Computer-Aided Software Engineering*, July 1992.

[Har93] William Harrison and Harold Ossher. Subject-Oriented Programming (A Critique of Pure Objects. In *OOPSLA '93 Conference Proceedings: Object-Oriented Programming Systems, Languages and Applications*, Washington, DC, September 26 - October 1, 1993, pages 411-428.

[Har94] William Harrison, Harold Ossher, Randall B. Smith, and David Ungar. Subjectivity in Object-Oriented Systems: Workshop Summary. In *Addendum to OOPSLA '94 Conference Proceedings: Object-Oriented Programming Systems, Languages and Applications*, Portland, Oregon, October 23-27 1994, pages 131-136.

[Har95] Philip A. Hardin. *A Runtime Comparison of STL and P2*. Unpublished. December 11, 1995.

[Hei94] John S. Heidemann and Gerald J. Popek. File system development with stackable layers. *ACM Transactions on Computer Systems*, February 1994, pages 58-89.

[Hew96] *Smallbase API Reference Manual (Smallbase 4.5)*, Database Technology Department, Hewlett-Packard Laboratories, Palo Alto, California, March 14, 1996.

[Hey95] Michael Heytens, Sheralyn Listgarten, Marie-Anne Neimat, and Kevin Wilkinson. *Smallbase: A Main-Memory DBMS for High-Performance Applications (Release 4.2).* Database Technology Department, Hewlett-Packard Laboratories, Palo Alto, California, September 7, 1995.

[Jac97] Ivar Jacobson, Martin Griss, and Patrik Jonsson. *Software Reuse: Architecture Process and Organization for Business Success*, Addison-Wesley/ACM Press, 1997.

[Jim98] Guillermo Jimenez. Private Communication. 1998.

[Joh79] S. C. Johnson. Yacc: Yet Another Compiler-Compiler. In *UNIX Programmer's Manual–Supplementary Documents, Seventh Edition*, AT&T Bell Laboratories, Indianapolis, Indiana, 1979.

[Joh88] Ralph E. Johnson and Brian Foote. Designing Reusable Classes. In *Journal of Object-Oriented Programming*, June/July 1988, Volume 1, Number 2, pages 22-35.

[Kan93] Elaine Kant. Synthesis of Mathematical-Modeling Software. In *IEEE Software*, May 1993, pages 30-41.

[Kat87] Shmuel Katz, Charles Richter, and Khe-Sing The. PARIS: A System for Reusing Partially Interpreted Schemas. In *Proceedings of the 9th International Conference of Software Engineering*, Monterey, California, Januray 1987, IEEE Computer Society Press, pages 377-385. Also in Will Tracz, editor, *Tutorial: Software Reuse: Emerging Technology*, IEEE

Computer Society Press, Los Alamitos, California, 1988. Also In [Big87], pages 257-273.

[Kel82] Arthur Keller. Updates to Relational Database Through Views Involving Joins. In Peter Scheuermann, editor, *Improving Database Usability and Responsiveness*, Academic Press, 1982.

[Ker88] Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language, Second Edition*, Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[Kic91] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*, MIT Press, Cambridge, Massachusetts, 1991.

[Kru84] Philippe Kruchten, Ed Schonberg, and Jacob Schwartz. *Software Prototyping using the SETL Programming Language*. In IEEE Software, October 1984, pages 66-75.

[Kru92] Charles W. Krueger. Software Reuse. In *ACM Computing Surveys*, Volume 24, Number 2, June 1992.

[Lei94] Julio Casar Sampaio do Prado Leite, Marcelo Sant'Anna, Felipe Gouveia de Freitas, Draco-PUC: a Technology Assembly for Domain Oriented Software Development. In [Fra94].

[Les79] M. E. Lesk and E. Schmidt. Lex: A Lexical Analyzer Generator. In *UNIX Programmer's Manual–Supplementary Documents, Seventh Edition*, AT&T Bell Laboratories, Indianapolis, Indiana, 1979.

[Lew91] John A. Lewis, Sallie M. Henry, Dennis G. Kafura, and Robert S. Schulman. An Empirical Study of the Object-Oriented Paradigm and Software Reuse. In Andreas Paepcke, editor, In *OOPSLA '91 Conference Proceedings: Object-Oriented Programming Systems, Languages and Applications*, Phoenix, Arizona, November 1991.

[Mad93] Ole Lehrmann Madsen, Birger Moller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the Beta Programming Language*, Addison-Wesley, Reading, Massachusetts, 1993.

[McI68] M. D. McIlroy. *Mass produced software components*. In [Nau68].

[Met96] Michael Metcalf and John Reid. *Fortran 90/95 Explained*, Oxford University Press, May, 1996.

[Mey91] Bertrand Meyer. *Eiffel: The Language*, Prentice Hall, New York, 1991.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction*, Prentice Hall, New York, 1997.

[Mic98] http://www.microsoft.com/

[Mir90] Daniel P. Miranker, David Brant, Bernie J. Lofaso, and David Gadbois. On the performance of lazy matching in production systems. In *Proceedings of the 8th National Conference on Artificial Intelligence*. Boston, Massachusetts, July, 1990, pages 685-692.

[Mir91] Daniel P. Miranker and Bernie J. Lofaso. The Organization and Performance of a TREAT Based Production System Compiler. In *IEEE Transactions on Knowledge and Data Engineering*, March 1991.

[Mus96] David R. Musser and Atul Saini. *STL Tutorial & Reference Guide: C++ Programming With the Standard Template Library*, Addison-Wesley, Reading, Massachusetts, 1996.

[Nau68] P. Naur and B. Randall, editors, *Software Engineering: A Report on a Conference Sponsored by the NATO Science Committee*, NATO Scientific Affairs Division, Brussels, Belgium, 1968.

[Nei81] James M. Neighbors. *Software Construction Using Components*, Ph.D. dissertation, Department of Information and Computer Science, The University of California, Irvine, 1981.

[Nei84] James M. Neighbors. The Draco Approach to Constructiong Software from Reusable Components. In *IEEE Transactions on Software Engineering*, September 1984, pages 564-573.

[Nei89] James M. Neighbors. Draco: A Method for Engineering Reusable Software Systems. In Ted J. Biggerstaff and Alan Perlis, editors, *Software Reusability*, Addison-Wesley/ACM Press, 1989. Also in [Pri91].

[Nei94] James M. Neighbors. An Assesment of Reuse Technology after Ten Years. In [Fra94].

[Nov83] Gordon S. Novak. GLISP: A Lisp-based Language with Data Abstraction. In *AI Magazine*, Volume 4, Number 3, Fall 1983, pages 37-47. Also in Robert Engelmore, editor, *Readings from AI Magazine*, AAAI Press, 1988, pages 545-555.

[Nov91] Gordon S. Novak and William C. Bulko. Understanding Natural Language with Diagrams. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, AAAI Press/MIT Press, July 29-August 3, 1990, pages 465-470.

[Nov92] Gordon S. Novak. Software Reuse through View Type Clusters. In *Proceeding of the Seventh Knowledge-Based Software Engineering Conference (KBSE-92)*, McLean, Virginia, September 1992, pages 70-79.

[Nov93] Gordon S. Novak Jr. and William C. Bulko. Diagrams and Text as Computer Input. In *Journal of Visual Languages and Computing*, Volume 4, Number 2, 1993, pages 161-175.

[Nov97] Gordon S. Novak. Software Reuse by Specialization of Generic Procedures through Views. In *IEEE Transactions on Software Engineering*, July 1997, pages 401-417.

[Nor96] Mike Norman and Robin Bloor. To Universally Serve. In *Database Programming & Design*, Miller Freeman, July 1996, pages 26-35.

[OMa92] Sean W. O'Malley and Larry L. Peterson. A Dynamic Network Architecture. In *ACM Transactions on Computer Systems*, May 1992, pages 110-143.

[OMG98] http://www.omg.org/

[Oss92] Harold Ossher and William Harrison. Combination of Inheritance Hierarchies. In *OOPSLA '92 Conference Proceedings: Object-Oriented Programming Systems, Languages and Applications*, Vancouver, British Columbia, October 18-22 1992, pages 25-40.

[Oss95] Harold Ossher, Matthew Kaplan, William Harrison, Alexander Katz, and Vincent Kruskal. Subject-Oriented Composition Rules. In *OOPSLA '95 Conference Proceedings: Object-Oriented Programming Systems, Languages and Applications*, Austin, Texas, October 1995, pages 235-250.

[Pau96] Lawrence C. Paulson. *ML for the Working Programmer, Second Edition*, Cambridge University Press, 1996.

[Par79] David L. Parnas. Designing Software for Ease of Extension and Contraction. In *IEEE Transactions on Software Engineering*, March 1979, pages 128-138.

[Par90] Helmet A. Partsch. *Specification and Transformation of Programs: A Formal Approach to Software Development*, Springer-Verlag, 1990.

[Pri86] Ruben Prieto-Diaz and James M. Neighbors. Module Interconnection Languages. In *Journal of Systems and Software*, Volume 6, Number 4, November 1986, pages 307-334. Also in Peter Freeman, editor, *Tutorial: Software Reusability*, IEEE Computer Society Press, 1987.

[Pri91] Ruben Prieto-Diaz and Guillermo Arango. *Domain Analysis and Software Systems Modeling*, IEEE Computer Society Press, 1991.

[Pri93] Ruben Prieto-Diaz. Status Report: Software Reusability. In *IEEE Software*, May 1993, pages 61-66.

[Pri95] Ruben Prieto-Diaz. Systematic Reuse: A Scientific or an Engineering Method? In [Sam95], pages 9-10.

[Rea92] Reasoning Systems, Palo Alto, California. *REFINE User's Guide*, 1992.

[Ric88] Charles Rich and Richard C. Waters, *Automatic Programming: Myths and Prospects*. In IEEE Computer, August 1988, pages 40-51.

[Ric95] Jeffrey Richter. *Advanced Windows: the developers guide to the Win32 API for Windows NT and Windows 95*, Microsoft Press, Redmond, Washington, 1995.

[Rum98] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual*, Addison-Wesley, Reading, Massachusetts, 1998.

[Sam95] Mansur Samadzadeh and Mansour Zand, editors, *Proceeedings of the ACM SIGSOFT Symposium on Software Reusability*, Seattle, Washington, April 28-30 1995.

[SEI90] Software Engineering Institute, *Proceedings of the Workshop on Domain-Specific Software Architectures*, Technical Report CMU/SEI-88-TR-30, Hidden-Valley, Pennsylvania, July 9-12 1990.

[Sel88] Richard W. Selby. Empirically Analyzing Software Reuse in a Production Environment. In Will Tracz, editor, *Software Reuse: Emerging Technology*, IEEE Computer Society Press, 1988, pages 176-189.

[Sim95] Charles Simonyi. The Death of Computer Languages, the Birth of Intentional Programming. In *Proceedings of the NATO Science Committee Conference*, 1995. Also Technical Report MSR-TR-95-52, Microsoft Research, http://research.microsoft.com/

[Sin92] Vivek Singhal, Sheetal Kakkad, and Paul R. Wilson. Texas: An Efficient, Portable Persistent Store. In *Persistent Object Systems: Proceedings of the*

Fifth International Workshop on Persistent Object Systems, San Miniato, Italy, September 1992, pages 11-33. Springer-Verlag Workshops in Computing.

[Sin93a] Vivek Singhal and Don Batory. *P++: A Language for Software System Generators*. Technical Report 93-16, Department of Computer Sciences, University of Texas at Austin, November 1993.

[Sin93b] Vivek Singhal and Don Batory. P++: A Language for Large-Scale Reusable Software Components. In *Proceedings of the Sixth Annual Workshop on Software Reuse*, Owego, New York, November 1993.

[Sir94] Martin J. Sirkin. *A Software System Generator For Data Structures*. PhD dissertation, University of Washington, 1994.

[Sit94] Murali Sitaraman and Bruce W. Weide, editors, Special Feature: Component-Based Software Using RESOLVE. *ACM Software Engineering Notes*, October 1994, pages 21-67.

[Sma96] Yannis Smaragdakis and Don Batory. *Scoping Constructs for Program Generators*. Technical Report 96-37, Department of Computer Sciences, University of Texas at Austin, December 1996.

[Sma97] Yannis Smaragdakis and Don Batory. DiSTiL: a Transformation Library for Data Structures. In [Use97].

[Smi85] Douglas R. Smith, Gordon Kotik, and Stephen J. Westfold. KIDS: Research on Knowledge-Based Software Environments at Kestrel Institute. In *IEEE Transactions on Software Engineering*, November 1985, pages 1278-1295.

[Smi91] Douglas R. Smith. KIDS–A Knowledge-Based Software Development System. In Michael R. Lowry and Robert D. McCartney, editors, *Automating Software Design*, AAAI Press/The MIT Press, Menlo Park, California, 1991, pages 483-514.

[Sou92] Jiri Soukup. Memory-Resident Databases. In The C++ Report, February 1992, pages 11-15.

[Ste90] W. Richard Stevens. *UNIX Network Programming*, Prentice Hall, Englewood Cliffs, New Jersey, 1990.

[Sto91] Michael Stonebraker, Greg Kemnitz. The Postgres Next Generation Database Management System. In *Communications of the ACM*, October 1991, pages 78-92.

[Sto93] Michael Stonebraker. The Miro DBMS. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, Washington, DC, May 1993, page 439.

[SUI98] http://suif.stanford.edu/

[Sun98] http://java.sun.com/beans/

[Tho93] Jeff Thomas, Don Batory, Vivek Singhal, and Marty Sirkin. A Scalable Approach to Software Libraries. In *Proceedings of the Sixth Annual Workshop on Software Reuse*, Owego, New York, November 1993.

[Tho95] Jeff Thomas and Don Batory. *P2: An extensible Lightweight DBMS.* Technical Report 95-04, Department of Computer Sciences, University of Texas at Austin, February 1995.

[Tra93] Will Tracz. LILEANNA: A Parameterized Programming Language. In Ruben Prieto-Diaz and William B. Frakes, editors, *Advances in Software Reuse: Selected Papers from the Secomd International Workshop on Software Reusability*, Lucca, Italy, March 24-26 1993, IEEE Computer Society Press, pages 66-78.

[Tra94] Will Tracz, editor. Domain-Specific Software Architecture (DSSA) Frequently Asked Questions (FAQ). In *ACM Software Engineering Notes*, April 1994, pages 52-56.

[Tra95] Will Tracz. Confessions of a Used-Program Salesman: Lessons Learned. In [Sam95], pages 11-13.

[Ube94] Michael Ubell. The Montage Extensible DataBlade Achitecture. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, Minneapolis, Minnesota, May 24-27 1994, page 482.

[Ung87] David Ungar and Randall B. Smith. Self: The Power of Simplicity. In Leigh Power and Zvi Weiss, editors, In *OOPSLA '87 Conference Proceedings: Object-Oriented Programming Systems, Languages and Applications*, Orlando, Florida, October 1987.

[USE97] USENIX Association. *Proceedings of the Conference on Domain-Specific Languages*, Santa Barbara, California, October 15-17, 1997.

[Van95] Michael VanHilst and David Notkin. *Using C++ Templates to Implement Role-Based Designs*. In Kokichi Futatsugi and Satoshi Matsuoka, editors, *Object-technologies for advanced software: Second JSSST International Symposium, ISOTAS '96: Proceedings*, Kanazawa, Japan, Springer-Verlag, March 1996.

[Vil94] Emilia E. Villarreal. *Automated Compiler Generation for Extensible Data Languages*, Ph.D. Thesis. Department of Computer Sciences, University of Texas at Austin, 1994.

[Vil97] Emila E. Villarreal. Rosetta: A Generator of Data Language Compilers. In *Proceeedings of the ACM SIGSOFT Symposium on Software Reusability*, Boston, Massachusetts, May 17-19 1997, pages 146-156.

[Wat88] Richard C. Waters. Program Translation via Abstraction and Reimplementation. In *IEEE Transactions on Software Engineering,* August 1988.

[Wei90] David M. Weiss. *Synthesis Operational Scenarios*. Technical Report 90038-N, Version 1.00.01, Software Productivity Consortium, Herndon, Virginia, August 1990.

[Wel92] David L. Wells, Jose A. Blakeley, Craig W. Thompson: Architecture of an Open Object-Oriented Database Management System. In *IEEE Computer*, October 1992, pages 74-82.

[Wil93] D. Wile. *POPART: Producers of Parsers and Related Tools, Reference Manual*. Technical Report, USC/Information Sciences Institute, 4676 Admiralty Way, Marina del Rey, CA 90292, 1993.

[Zdo90] Stanley B. Zdonik, David Maier, and Stanley Zdonik, editors, Readings in Object-Oriented Database Systems, Morgan-Kaufmann, 1990.

Vita

Jeffrey Alan Thomas was born in Philadelphia, Pennsylvania on July 12, 1967, the son of Dr. Donald Harvey Thomas and Ann Rosenberg. After graduating from Haverford Township High School, Havertown, Pennsylvania in June 1985, he entered Cornell University, Ithaca, New York. There he received the degrees of Bachelor of Science with Distinction in Computer Science in June 1989, and Master of Engineering in Computer Science in January 1992. In June 1992, he entered the Graduate School of the University of Texas at Austin. There he received the degree of Master of Science in Computer Sciences in December 1995.

Permanent Address: 7037 Elementary Road
Coopersburg, PA 18036-3644

This dissertation was typed by the author.