The Design of a CASE Environment Architecture and the Performance Evaluation of Database Designs for Software Documents

by

Luis Miguel

The Design of a CASE Environment Architecture and the Performance Evaluation of Database Designs for Software Documents

by

Luis Miguel

Abstract

CASE environments are the foundation on which software engineering can implement the policies and methodologies needed to efficiently produce the software systems of the future. Our dissertation focuses on CASE data management, in particular how to provide the powerful services demanded by CASE without sacrificing performance, and minimizing the storage space. Our approach has three main steps:

i) Find out what the requirements are.
ii) Design a system architecture with the requirements in mind.
iii) Choose the database designs for the CASE data structures with the requirements in mind.

In the area of requirements, we compile, analyze, and classify the CASE data management requirements. We also compile, analyze, and classify the different options for data manager, and determine which options satisfy which requirements.

In the system architecture area we analyze the main CASE environment architectures, and propose a new architecture that satisfies the data management requirements discussed earlier. We illustrate the architecture with a hypothetical CASE environment, TULUM.

In the area of CASE database designs we identify and classify the important CASE data structure parameters and the important CASE database schema dimensions. Using this framework we design and conduct a series of experiments to measure the space and efficiency of database designs that vary in 3 dimensions: granularity, data representation, and the number of relations.

Our achievements include a comprehensive list and classification of CASE Database requirements, as well as the design of an architecture that satisfies those requirements efficiently. We also provide a data structure analysis framework that allows database designers to make informed database schema choices. The results of our experiments show that data storage and retrieval have three important phases and that each has to be tuned to prevent bottlenecks, that increasing granularity provides substantial space and speed gains which rapidly level off, that data representation has an enormous performance impact, and that we need to minimize the number of relations and the number of tuples to maximize performance.

DEDICATION

To my most loving Maria:
After this, life is a piece of cake.....
To my three boys, Gary (5), Rafael (3), and Gregor (2):
I finally have time to be a real papi to you.....
To my mother Teresa, whose immense love for me only grows:
The best mami I have ever had.....

Acknowledgements

This dissertation would not have been possible without the selfless support of many people. My research advisor, C.V. Rammoorthy, took me in when things were bleak, restored my self-esteem, and allowed me to metamorphose from a student into a researcher. You are a great man, Professor Ram! Arie Segev and Abhiram Ranade took the time to serve on my committee, read my dissertation, and provide valuable comments.

Bienvenido Velez and Larry Rowe provided the initial impetus for this work. Yongdong Wang and I helped each other get through the PhD from early on, reading each others' papers, making sure we were advancing in our research, and finally reading and correcting each others' dissertations. Thanks amigo. Young-Chul Shim read early drafts of this work, and gave suggestions that improved it significantly. Ginger Ogle helped me improve the writing style in the key sections of the dissertation.

I was able to trick the Reentry Program for Women and Minorities, an incredibly successful program that provides new opportunities for talented individuals, into accepting me as a student when all I knew about computers was *that they computed.* Thank you, RP, for the opportunity and all the support. Sheila Humphreys and Eugene Lawler: What can I say about these two incredible human beings? They supported me first through the RP, and then throughout the graduate program. Above all, I thank them both for their trust and friendship.

Finally, and most importantly, I thank my beloved partner, Maria. She has provided emotional support, three wonderful boys, financial backing, and the eternal patience needed to wait for me to become a normal human being again. I think it was five years ago that I first told her *only two more years* when she asked how much longer it would take me to finish. I thank Albert Einstein for giving me the Theory of Relativity to explain to Maria that, after all,

time is relative.......

Table of Contents

List of Figures

List of Tables

List of Abbreviations

Software Engineering	SE
Computer Aided Software Engineering	CAS E
Database Management System	DBMS
Abstract Syntax Tree	AST
Object Oriented	OO
Abstract Data Type	ADT
Programmer's Apprentice	PA
Knowledge Based Software Assistant	KBSA
User Interface Management System	UIMS
Enhanced Workstation Server	EWS
Object Oriented Database Management System	OODBMS
Inter Process Communication	IPC
Attribute per tuple	APT
External Data Representation	XDR
Line per tuple	LPT
Symbol Table entry per tuple	SPT
Symbol Table	SYMTAB
Node per tuple	NPT
Relation per attribute type, attribute per tuple	R-APT
Relation per structure kind, line per tuple	R-LPT
Procedure per tuple	PPT
Module per tuple	MPT
Relation per node type, node per tuple	R-NPT
Frontend	FE
Backend	BE
Binary large object	BLOB
Network File System	NFS
Symbol table entry	STE
Attribute	ATT
Kilo Byte	KByte
Mega Byte	MByte
milliseconds	msecs
Revision Control System	RCS
Software System	SS

CHAPTER 1

INTRODUCTION

1.1. Motivation

As computers have become cheaper and more powerful, they have been used in more domains to develop and operate large systems like air traffic control and telephone switching. During the 1950's and 60's the initial experiences building large software systems showed that the projects were usually late, cost more than originally estimated, were unreliable and difficult to maintain, and performed poorly. With falling hardware costs and increased access to computers, the demand for large software systems kept rising while software development technology could not keep up. This phenomenon became known as the *software crisis*. To tackle the software crisis, the discipline of Software Engineering (SE) was created in the 1970's. The objective of SE is the efficient and cost effective development of high quality software. SE strives to produce *large* software systems that can be easily maintained, have high reliability, and satisfy their requirements.

Starting in the mid 1960's and through the 70's and 80's, important advances in software technology increased programmer productivity significantly (e.g., high-level languages and integrated programming environments). But even as SE has advanced, the software crisis has worsened. The reason is that the improvements in SE, while significant, have not been as significant as the advances in hardware technology. The dramatic increase in the power of computers, their equally dramatic drop in price, and their minitiurization has led to a mushrooming in the demand for software systems with more stringent constraints, and very tight performance and reliability requirements. Therefore, the software crisis is still with us.

To solve the software crisis many difficult SE problems need to be solved, including the development of a practical and realistic software lifecycle model, the use of formal methods throughout the lifecycle, and the development and use of quantitative measures of progress. The help of computers is essential in solving these problems in order to achieve the orders of magnitude improvements needed in software engineering. Computers can take over many mundane and repetitive tasks such as formatting a document in accordance with established rules, or checking a program for static semantic errors. They can speed up other tasks such as the write-compile-test program development paradigm, and they can present information to the software engineer in an easy to understand form such as a program call graph. When we combine the discipline of SE with the power of computers, we produce a hybrid discipline, with the same goals as SE, called Computer Aided Software Engineering (CASE). CASE has gained prominence in recent years, with vigorous research and development efforts in leading universities and private companies. The consensus in the CASE community is that it can best achieve its goals through the design and development of powerful CASE environments.

CASE environments are envisioned as the foundation on which software engineering can implement the policies and methodologies needed to produce the software systems of the future. In order to implement these policies, the CASE environment designer must endow the environment with primitive features that can be used to provide many sophisticated services. For example, a rule system could be included so that we can enforce specific policies, such as requiring that before a new version of a file is checked in, it must undergo a path coverage test.

CASE environments range from independent collections of tools that operate on files, like those described in [1], to very sophisticated, tightly integrated sets of tools that provide explicit support throughout the software lifecycle, as in [2, 3]. CASE environments must store very large amounts of information relating to the software development process for the life of the system. This information is highly structured, interrelated, and heterogeneous and includes requirements, specifications, documentation, system configurations, and program source and object code. CASE data management is a very important issue that affects the level and kinds of services that the environment can provide. Many open issues in this area remain, most notably how to provide the powerful services demanded by CASE without sacrificing performance and minimizing the storage space. This is the focus of our dissertation.

1.2. CASE Data Management

Logically, CASE information is stored in a database and managed by a Database Management System (DBMS). The requirements placed on the DBMS by the CASE tools are many, complex, and stringent (e.g., fine grained access, random querying, data consistency, very fast response times). There are systems that satisfy different subsets of these requirements, but there is no single system that satisfies them all. This is due in part to complexity, but also to the fact that with existing approaches satisfying one requirement often means failing to satisfy another (e.g., in current DBMS's fine granularity designs provide slow access). In order to satisfy the data management requirements, we have determined that three things are essential:

1) We must know what the requirements are.
2) The architecture of the system must be designed with the requirements in mind.
3) The database designs for the CASE data structures must be chosen with the requirements in mind.

If we do not know what the requirements are, it is hard to provide an environment that satisfies them. In particular, by looking at the list of requirements, an environment designer can decide which are worth supporting and how.

The architecture of the CASE environment needs to be carefully designed and implemented so it can effectively support its demanding requirements. For example, the environment must provide efficient access to private data for individual engineers, concurrency control to shared data for a local team, and access to geographically remote data for project managers. That is, the architecture must provide very fast local inter-tool communication, fast local inter-environment communication, and good remote inter-site communication.

In addition, the database designs for the CASE data structures determine what functionality a data structure can support, as well as how much space it consumes and how efficiently it can be accessed. In general, we need to minimize the *impedance mismatch* of the data between the tools and the CASE data manager. For example, an abstract syntax tree (AST) might be represented in the application as a series of records for the nodes with pointers for the links between nodes. It can be represented in the database in ASCII or in binary format, and it can be stored as a node per tuple or as a procedure subtree per tuple.

1.3. Contributions

The main contribution of this dissertation is to provide, in one place, a comprehensive treatment of CASE database management. In doing so, we take advantage of the contributions of other people, complement them with our own, and more importantly, we organize them, and clearly show why and how they are relevant to CASE data management. We identified three areas which are of vital importance, and we addressed each in our work:

CASE Database Requirements: Although lists of CASE database requirements had been compiled before, none was comprehensive, explained why each requirement was relevant, or classified them. Further, there was no attempt at classifying how the different data manager classes satisfy the requirements.

We compiled a comprehensive list of requirements, explained why each was necessary, and classified them according to two different criteria. We then classified the different data managers (existing and proposed) into classes, and rated how each class satisfied each of the requirements.

CASE Environment Architecture: Several architectures exist or have been proposed, but none of them addresses all the requirements that we listed above.

We propose an architecture that addresses all the requirements by explicitly separating three system issues:

1) Physical issues: How processes and programs are physically distributed.

2) Logical issues: How are items represented, and what are the kinds of relationships between them.

3) Computational Paradigm: Specifies the philosophy underlying the implementation of the system, like incrementality and the use of parallelism.

By explicitly addressing the three issues in the design of our architecture, we maintain that our design is superior to the others.

Database Designs for CASE Data: Several authors have stored programs in DBMS's, but none has compared different ways of storing them, or provided criteria to use in deciding how to store a program in a database.

We provide a logical framework that classifies the important in-core characteristics of program data structures. We use this framework to analyze the program data structures that our experiments cover ahead. We also provide a logical framework that classifies the important dimensions of the database designs of a data structure. We use this framework to develop the database schemas that we examine in our experiments. Finally, we design and run experiments that measure the performance and space utilization characteristics of the database designs that we developed above.

1.4. Approach

In this section we explain how we address the requirement, architecture, and database design issues. In the area of requirements, we compile, analyze, and classify the CASE data management requirements. We also compile, analyze, and classify the different options for a data manager, and determine which options satisfy which requirements.

In the system architecture area we analyze the main CASE environment architectures, and propose a new architecture that satisfies the data management requirements discussed earlier. We illustrate the architecture with a hypothetical CASE environment, TULUM.

In the area of CASE data structure database designs we first identify and classify the important CASE data structure parameters, and the important CASE database schema dimensions. We then design and conduct a series of experiments to measure the space and efficiency of database designs for different data structures that vary in 3 dimensions: granularity, data representation, and the number of relations.

We also provide detailed examples of sophisticated CASE services with TULUM, our hypothetical CASE environment: software reuse, version and configuration support, and maintaining consistency between different database representations of the same data. The next section summarizes the main results and conclusions from our research.

1.5. Results And Conclusions

In this section we highlight the major results that we obtained and point out their implications.

CASE Database Requirements: We produce a list of some 30 different CASE database requirements that can be used as a guide for implementors to know what to include in their system, as well as for prospective users who must choose among different systems. We classify the requirements according to whether they are intrinsic to the data model of the DBMS or functional attributes that are implementation dependent. This differentiation is important because the functional attributes can be added to a system at any time, while a change to the other attributes requires a change in the data model with great consequences for existing applications. We also look at the main data manager options, and evaluate how each data manager class satisfies each of the requirements. We find that almost all the classes satisfy most of the data management requirements, about half of them satisfy the object management features, and just a couple have knowledge and process programming support. We conclude that a custom DBMS approach has many drawbacks, and that using either an extended relational or Object Oriented (OO) DBMS is a good idea, since they both satisfy most of the requirements and are currently being enhanced.

CASE Environment Architecture: We designed a CASE environment architecture that provides a sound basis to satisfy the requirements listed earlier. Our design is based on a synergy between the physical organization, the logical organization, and the computational paradigms used in the system. The physical architecture is client-server, with a clear demarcation between the toolkit and the data manager. The toolkit runs on the clients, and the data manager, which has both a client and a server component, transfers much of the CPU and I/O load to the clients, greatly alleviating the server bottlenecks. In addition, the architecture allows for local tool communication via shared memory, and for client to client communication either directly via messages, or through the data manager. The logical architecture is knowledge- and process-centered, giving the environment the power to satisfy all its functionality requirements, while also providing flexibility to the user in choosing a tool-, data-, knowledge-, or process-centered approach (or any combination thereof). The computational paradigm aims to couple the characteristics of the physical and logical architectures to maximize system performance without any loss of functionality. It is based on three principles: lazy reanalysis, incremental computation, and parallelism. Lazy reanalysis prevents the system from computing information that will not be used, incremental computation avoids redundant work by modifying existing structures as opposed to regenerating them from scratch when they change. They both aim to minimize the number of interactions and the amount of information exchanged between the clients and servers. Parallelism plays a role in distributing the work among the clients and servers so that it can be scheduled in parallel.

Data Structure Analysis: To provide a logical framework that classifies the important in-core characteristics of program data structures, we develop five criteria, each having one or more characteristics. The criteria deal address:

1) Internal structure
2) Information content
3) Quantitative characteristics
4) Qualitative characteristics
5) Sharing characteristics

Data Structure Database Designs: We provide a logical framework that classifies the important dimensions of the database design of a data structure. This framework complements and is orthogonal to the in-core data structure analysis framework we developed before. The database design for a given data structure can vary in three important dimensions:

1) Granularity
2) Data Representation
3) Number of Relations

We propose three alternatives to achieve fine granularity functionality with coarse granularity designs: abstract data types, location based, and embedded identifiers. We then classify the data structures that arise in CASE into five classes according to the kinds of data they contain: textual, tabular, graphical, unstructured, and sequential binary.

We then use the data structure analysis framework we developed to analyze the data structures that we use in our experiments, and we develop different database schemas for these structures based on the three database dimensions referred to above:

Database Design Experiments: We design and run experiments that measure the performance and space utilization characteristics of the different database designs we developed. We measure a realistic sample of program files, and we then isolate the effects of data representation, the number of relations, and the granularity. The main results of our experiments are:

- Storing data in the database involves three phases: data extraction, data conversion, and database query. Retrieving data from the database into memory involves three phases: database query, data conversion, and data structure reconstruction. To optimize performance, each phase has to be evaluated separately.
- The medium granularity designs provide performance similar to creating the data structures from scratch (e.g., creating the abstract syntax tree by parsing the source).
- As granularity increases, the main drawback is reduced functionality.

- A small granularity design is very expensive both in time and space, and should only be considered when there is abundant disk space and when the extra DBMS functionality is essential.
- The more limited amount of functionality provided by the larger granularities may be sufficient for many applications.
- The choice of data representations has a small impact on space usage, but, depending on how different the in-core and database representations are, can dominate the operation time. The larger the difference, the more expensive conversion between representations becomes.
- When we deal with in-core data structures, we can trade space for performance (assuming the system has enough memory to accommodate the extra space). But, unfortunately, with the DBMS the added size usually translates into slower performance since the extra size coming from smaller granularity adds to both major DBMS performance determinants: the number of tuples examined plus the number of disk pages processed (An exception is the extra space occupied by indices which in essence obey the in-core space/performance rule).

1.6. Organization

In this section we describe the organization of this dissertation. In Chapter 2 we discuss the most important previous work in the three areas of software databases that our work covers: database designs for programs, CASE database requirements, and CASE environment architectures. For each area we mention what the important work was, provide a brief description of its main features, and highlight what we learned from it. The last section acknowledges the marked influence that the POSTGRES DBMS has had on our work.

In Chapter 3 we examine the requirements of the the CASE DBMS, and we consider the possible choices for a data manager. There are two kinds of requirements: those intrinsic to the data model, and those that can be offered by all models. In addition, we classify the requirements according to whether they satisfy data, object, knowledge, or process management issues. The CASE data manager has the responsibility of meeting these requirements. We investigate the the different data manager classes, define them, and determine how well they each satisfy the requirements. We conclude with a recommendation of which classes satisfy most of the requirements.

In Chapter 4 we cover the main physical and logical CASE environment architectures. We describe the physical and logical architectures as well as the computation paradigm used in TULUM, our proposed new CASE environment. Our objective is to uncover the weaknesses of existing architectures, and to suggest a new system architecture that explicitly addresses those weaknesses.

In Chapter 5 we cover the in-core characteristics and main database design dimensions of CASE data structures. We first examine the data structures while they are in-core. We develop five criteria that are important in a database context: internal structure, information content, quantitative aspects, qualitative aspects, and sharing aspects. Next we focus on the database, and look at the main dimensions of a database design for a data structure: granularity, number of relations, and data representation. We cover the characteristics of software engineering documents, and classify them into five classes according to the kind of data that they hold: textual, tabular, graphical, unstructured, and sequential binary. Finally we use these frameworks to analyze the program source, abstract syntax tree, and symbol table.

In Chapter 6 we report on a suite of experiments that we designed to test the performance of different database designs for major CASE data structure classes in terms of time and space. The program representations that we choose are program source code from the textual class of data structures, an abstract syntax tree from the graphical class of data structures, and a symbol table from the tabular class of data structures. The database designs vary in three dimensions: granularity, number of relations, and data representation. We divide the experiments into two parts. The first part explores the performance and space characteristics of different database designs using program source files from widely used UNIX programs. The objective is to evaluate the designs using realistic data. The second part isolates the effects of the three database dimensions. We close by looking at the performance of a variety of data managers, and by evaluating our results in the light of the different data structure kinds.

In Chapter 7 we show the power of TULUM in providing advanced functionality to CASE environments with three practical example applications. We use several of the advanced features of the data repository, including the ADT capability, the rules system, and database procedures. In addition, we show how we can provide useful services both to fine and coarse granularity designs. The examples are version control, support for software reuse, and support for derived data and multiple views of the same data.

Chapter 8 restates the major goals of our work, and summarizes its major results and accomplishments in three areas: CASE database requirements, CASE environment architectures, and database designs for programs. We make suggestions for future work, and offer some brief concluding remarks.

CHAPTER 2

PREVIOUS WORK

2.1. Introduction

In this chapter we discuss the most important previous work in the three areas of software databases that our work covers: database designs for programs, CASE database requirements, and CASE environment architectures For each area we mention what the important work was, provide a brief description of its main features, and highlight what in particular we learned from it. The last section acknowledges the large influence that the POSTGRES DBMS has had on our work.

2.2. Software Databases

In this section we cover the software database systems that have influenced our thinking, from the point of view of the kind of CASE information stored, the storage medium, and the database designs used. We have divided previous work in software databases into 4 categories: program information databases, program databases, project information databases, and other systems. We explain each in turn, and within each category we discuss the important systems.

A) Program Information Databases

The databases in this category contain information, derived from the source code, that supports program interrogation facilities. Program interrogation capabilities help the software developer understand the structure of his programs, and the inter-relations between the elements of the program. The consumers of the program information are tools that aid the software engineer in the understanding, debugging, development, and maintenance activities. For example, if we have a relation

proc (caller, callee)

where caller is the procedure that calls the callee procedure, we can find out the names of all procedures that call the procedure Redisplay with the following query:

retrieve (proc.caller) where proc.callee = "Redisplay"

Other examples of program information are the place where a variable is declared, or all the variables of a given type. The two program information systems that influenced us are MasterScope and CIA.

MasterScope is the parent of the program information database systems [4]. It has an English-like query interface, provides incremental updates, and has been incorporated into the Interlisp environment [5]. Within Interlisp, MasterScope is integrated with the editor, debugger, and other tools. For example, the programmer can ask to see all the statements that reference a global variable, and the editor will position the cursor on each statement in turn. Due to its integrated design, Interlisp has a single interface with which the programmer interacts. MasterScope stores program information in files, and implements a subset of relational capabilities and some inferencing based on those files. It pointed the way to the sophisticated services possible when software information is extracted, stored, and used to support the programming process.

CIA [6], the C Information Abstractor, and its ADA derivative, APAS [7], extract information from the source programs and store it in a relational DBMS. CIA concentrates on external, or high level, information between C program entities that include files, macros, variables, types, and functions. CIA ignores local entities because they increase the database size substantially, they affect a small context, and the information can easily and efficiently be generated on the fly if necessary. CIA is part of an active research and development program in CASE at AT&T Bell Laboratories [8]. CIA applications include graphical views of software (e.g., call graphs), subsystem extraction, dead code elimination, software restructuring, and metrics. CIA is significant because it takes an ingenious approach that is simple, yet powerful. By limiting itself to high level entities, it can use a fine granularity database design that can exploit the full query power of the relational database, utilizes little space, and whose performance requirements can be satisfied with current relational technology. In addition, the information subset that it stores provides very powerful services to the software engineers, and requires few maintenance changes during program changes. Nevertheless, with the software documents not residing in the database, CIA provides only a small subset of the functionality needed by CASE environments.

B) Program Databases

Program databases contain the program code itself as well as information that is associated or derived from the source code. The program databases subsume those in the program information category. In addition, the consumers of the data in the program databases are software tools with stringent performance and data modeling requirements, such as compilers, and syntax directed editors. The program representations stored in program databases include symbol tables, source and object code, abstract syntax trees (AST's), and data flow graphs [9]. In Chapter 5 we cover some of these representations in detail. In many cases, the analysis that produced the information stored in a program database is expensive to perform.

We now give an example of the information contained in a program database, and a sample operation. A typical operation of a syntax directed editor is to modify a line of code. Such editors usually operate on an AST rather than the source code because the AST can be quickly analyzed for structural relationships and the presence of lexical and syntactic errors [10, 11]. If the AST is stored in a relation

t_node (id, name, module, parent, operator, attributes)

with the attributes in the relation

attribute (module, name, scope, value, t_node)

we can change the name of the procedure *Draw* to *Repaint* with the query[1]:

```
replace (a.value = "Repaint") from a in attribute, t in t_node
            where a.t_node = t.id and a.value = "Draw"
            and t.operator = proc_name
```

[1] Our examples use the POSTGRES rules system PRS2 [12] syntax, and the POSTGRES query language, POSTQUEL [13].

In addition, several program databases are designed for interactive use during program and tool execution, so that they store real time information. For example, the database may store information like relocatable program addresses and execution breakpoints. and its functionality may be used to perform runtime actions. The following example is taken from OMEGA [14]. To express events that refer to locations in a program we have the relation

callstack (procedure, level)

that contains a tuple for each active procedure at any given time, where *level* is the runtime stack depth. To notify the environment when the procedure "buggy" is called, we can have the following rule:

```
define rule call_buggy
    on append to callstack where callstack.procedure = "buggy"
    then notify_env ("buggy", callstack.level)
```

where notify_env is a procedure that performs the desired notification. The program databases we cover are OMEGA [14], Allegro [15], and a database design by Velez [16].

OMEGA is an ambitious system that stored program information and code in a relational DBMS [14]. OMEGA proposes a fine grained database schema to store the symbol table, program interrogation information, and an attributed AST. OMEGA augments the schema with *views* to provide alternate representations of data and *integrity constraints* to provide interactive semantic checking. It showed how a DBMS system can provide the foundation for all CASE tools. The system suffers from the weaknesses of relational DBMS's: poor data modeling and inadequate performance, although some of its performance problems can be traced to the database design they chose. It does not materialize a very common representation, the source code, and it consists of 58 relations and 15 views. The problems with this approach are several:
i) It computes, at query time, a common representation, the source code, from the AST and symbol table at great cost.
ii) The large number of relations leads to many small queries or few very complex queries to obtain or change desired information. Both approaches are expensive.
iii) The design has a large number of tuples, leading to slower response time and greater space usage.

Linton followed OMEGA with Allegro [15] in which he extends his approach to a distributed, object oriented environment. Allegro goes as far as OMEGA in advocating to use the database during program execution. In the object oriented approach, it encapsulates objects and their operations together, with individual objects responding to queries through a public interface. This approach is elegant, but raises questions of performance for queries spanning large object sets, since we cannot build indices on these operations, and we therefore incur large processing costs at query time.

Velez [16] designed an experiment on a new DB schema to store programs and attempted to measure how this design performs on primitive operations (object fetch, insert, and replace). Velez uses the POSTGRES DBMS [13] to store the source code, the symbol table and an attributed AST in a design that takes advantage of transitive closure queries. As OMEGA, Velez also stores the AST and SYMTAB, and adds the program source. Unfortunately, he was unable to complete the experiment due to performance and reliability problems with POSTGRES at that time. His design improves

on the OMEGA database design in several ways:
i) It stores the program text in the DBMS too, therefore avoiding costly query time computation.
ii) It minimizes the number of relations by using only 4 relations in his schema.

In addition, he proposes to evaluate the performance differences as granularity increases, and points to the importance of multi-append queries. We are deeply indebted to this work, as it provided the starting point for this dissertation.

Finally, Atherton Technology [17] has attempted to save programs in tokenized form. But they abandoned the attempt when it became apparent that this representation used more than twice the space as the source representation, was inconvenient to create and maintain, and could be recreated cheaply on demand.

From all this work, we have learned that space usage is important, and that the cost and feasibility of creating a representation on demand must be taken into account in the decision to store in the DBMS. Program databases require more sophisticated capabilities than the relational systems have. In addition, database performance has been at least two orders of magnitude slower than that required for practical runtime interaction with the execution environment.

C) Project Information Databases

Project information is related to, but not derived from, a program. It includes program specifications, project schedules, software metrics, user documentation, system configurations, and bug reports. For example, if we have a relation

bugs (project, date-reported, date-resolved, title, category, description, solution)

we can ask for all the bugs of category 1 (ie., critical bugs) that are still outstanding in the project *Graphical Interface*:

retrieve (bugs.all) where bugs.project = "Graphical Interface" and
bugs.category = 1 and bugs.solution = NULL

The project information databases also require more sophisticated capabilities than those in relational systems, especially to integrate the database into a sophisticated CASE environment. We cover one project information system, PMDB.

PMDB is a fully specified model of a complete CASE database [18]. Apart from its data modeling requirements, PMDB relies on DBMS extensions like object version support and a rules system. The designers prototyped PMDB using a relational DBMS, and found that the relational system provides unsatisfactory performance, poor data modeling, as well as lacking important features like abstract data types (ADT's), and rule support. It is unclear whether the authors plan on building a PMDB on top of an existing DBMS or implementing a custom DBMS to support their design. What is clear, is that CASE project management requires many features not provided by relational systems.

D) Other Influences

In terms of database designs for programs, we have been influenced by other work that does not fall neatly into the previous three categories.

ODIN [19] is an object store designed to incorporate existing, stand alone tools into integrated environments. It is data centered, and as an integration philosophy uses tools to manage software data repositories to expedite responses to interactive user commands. The repository consists of the objects that are the inputs to, and outputs of, the tools in the environment. ODIN is most effective using large grained tools to manage large grained objects. ODIN can be seen as an interpreter for a process language in which the objects are the operands, and the tools are the operators. From ODIN we learn how important fine grained object management is, and how the main obstacle to its implementation is performance.

Snodgrass and Shannon [20] examine a spectrum of approaches to tool integration. They focus on fine grained data because its performance requirements are the hardest to meet. They assume a central database environment architecture, but advocate a more refined model: a *tool topology* that states explicitly how tools interact and what data is exchanged or shared between them, and in which tools communicate directly via instances of strongly typed data structures. A strong advantage of this approach is that tool interaction need not be through the DBMS, leading to more flexibility and efficiency. The authors then analyze the decisions that have to be made regarding data structures in two contexts: the tool to tool connections, and when they reside in main memory. For each connection between tools, 4 decisions are made about shared data: i) its contents, ii) its representation, iii) consistency with previous representations, and iv) its form. For the form, it gives several choices that include its representation (ASCII external, independent binary, and dependent binary), and the mechanics of sharing (incremental I/O, or shared memory). For each tool 5 decisions are made about its (in-memory) data structures: i) its contents, ii) its representation, iii) its consistency, iv) its physical interface, and v) the binding of code performing data I/O and that manipulates its attributes.

From [20] we learn several things. In the area of architecture, it shows how important it is, both in efficiency and in functionality, to have direct communication mechanisms between tools and between toolkits. More importantly, they provided much of the inspiration for the data structure analysis we perform in Chapter 5 on database representations for data. Finally, they point out how important data representation differences are for both space and time efficiency.

2.3. CASE Database Requirements

In this section we review the work that helped us develop the list of CASE database requirements, the organizational criteria that we use to classify the requirements, and the evaluation criteria by which we judge how well existing systems satisfy the requirements.

There are several on-going CASE research projects that illustrate how important knowledge services are, most notably the Programmer's Apprentice (PA) project [21] and the Knowledge Based Software Assistant (KBSA) project [22]. Those systems support developers of software systems throughout their life-cycles with various functions. For example, the KBSA adopts the transformational approach in which the system, with the help of the developers, generates executable code from the requirement specifications through successive transformations. In the case of the PA, the system assists the developers in making the right decisions. The system detects inconsistencies automatically and suggests alternate ways and solutions so that developers can handle the problem. In addition Stonebraker [12] shows how a simple, but powerful rules system as part of the repository can provide many sophisticated services like data integrity and alerters.

The importance of process centered programming is pointed out by Professor Marvel and Arcadia. Professor Marvel [23] proposes a process model that consists of an extensible collection of rules that specify the conditions that must exist for particular tools to be applied to particular objects. That is, it advocates process programming via rules.

The Arcadia research project [24] is developing a prototype environment architecture to maximize flexibility, extendibility, and integration, and for understanding the tradeoffs between them. To achieve flexibility and extensibility it focuses on the notion of process programming. A process program in a process programming language describes the software processes to develop and maintain programs. Tools are functions or operators, and the operands are the software documents that are the products of tools and users. Flexibility comes from modifications to the process programs. Extensibility comes from writing new programs, and from extending existing ones to support new tools, subprocesses, objects, or types. They claim that a process programming language must be as powerful as a general purpose programming language. From Arcadia we learn about the importance of process centered architectures. In the area of architecture, we were influenced by the use of a user interface management system (UIMS) to provide integration for the users in a manner that is extensible, customizable, uniform, and consistent.

The report of a National Science Foundation workshop [25] that lists the most important items in the DBMS research agenda of the 1990's, convinced us to include support for a multi-level memory hierarchy as an important CASE database requirement.

The inspiration to compile a complete list of CASE database requirements came from attending a workshop on databases for CASE [26]. A significant number of the requirements we cover were contributed by several of the participants to the workshop. In addition, the participants were unanimous in recognizing that, in isolation, all of the requirements that were compiled could be satisfied by one or more existing systems, but that there was no single system that satisfied all or even the most important requirements.

2.4. CASE Environment Architecture

In Chapter 4, which covers CASE environment architecture, we cover in detail the most important previous work in that area. Here we briefly mention the two main systems, *ADMS$\pm$* and Pan. In addition, our discussion on Arcadia and the Snodgrass and Shannon paper mention the architectural influence that they had on our work.

The *ADMS$\pm$* enhanced workstation-server architecture [27] makes a significant advance in client server architectures which allows for dramatically better scaling up performance. It does so by shifting much of the server work to the clients by running a DBMS- on the client which caches and maintains the local database. In addition it minimizes the client-server communication overhead by using incremental computation models as the communication paradigm.

Pan [11] is a language based editing and browsing system that uses a decorated abstract syntax tree as its primary structural representation, and where all language oriented information is derived from the source code. Pan assumes that deriving expensive data is more effective when we can store it in a repository, so that it can be used by multiple tools and incrementally updated. In addition, Pan emphasizes the need to preserve non-recoverable information (like AST annotations during development, which might be equivalent to comments in the source). Pan has a lazy reanalysis policy, since it assumes that users will invoke reanalysis when desired, but it also encourages frequent reanalysis by making it cost-effective. Pan inspired us to include incremental computation methods to BE-FE application interaction, as well as to include the lazy evaluation paradigm as part of TULUM.

2.5. POSTGRES

Last, but not least, we acknowledge the many contributions that we have derived from POSTGRES, both from the many papers that have been written about it [28, 13, 12, 29, 30, 31], and usage of the system. Among the most important lessons we have learned from POSTGRES are how important and powerful a rules system is to support both simple and complex CASE tasks; how fast path access is essential to support performance critical applications; and the advantages of maintaining the relational model of data. In the rest of the dissertation we make specific reference to POSTGRES papers where it is relevant.

CHAPTER 3

CASE DATABASE REQUIREMENTS

3.1. Introduction

In this chapter we look at the CASE database requirements, as well as the possible choices for CASE data management. There are two kinds of requirements: those intrinsic to the data model, and those that can be offered by all models. In addition, we classify the requirements according to whether they satisfy data, object, knowledge, or process management issues. The CASE data manager has the responsibility of meeting these requirements, and we look at the different data manager classes and how well they satisfy the requirements. We conclude with a recommendation of what classes to choose to satisfy most of the requirements. Our objective is to understand what requirements a modern CASE environment places on its data manager, so we can satisfy them when designing the architecture for TULUM. In addition, our evaluation of different data managers allows new environment designers to make an informed choice when choosing the data manager for their system.

Computer Aided Software Engineering (CASE) generates vast amounts of heterogeneous data that includes manuals, program source, test results, and functional specifications [32]. In order to effectively manage this data, a CASE data repository has to fulfill several data management requirements that include large storage capacity, persistence, sharing, and integrity.

To serve as the basis of its data manager, a CASE environment designer can choose an existing database management system (DBMS), or he can "roll his own". Most CASE environments today have opted to roll their own, satisfying a limited subset of the CASE data requirements [19, 6, 7]. And in the cases where the choice has been an existing DBMS [14, 6] different kinds of problems have been identified depending on the specific choice. For example, OMEGA [14] found that a relational DBMS needed strong object identity, while CIA [6] found that it lacked adequate performance. If obstacles like these were overcome and we could use a general purpose DBMS for CASE, the benefits to CASE would include: sharing the same data repository with non-CASE data would allow CASE and non-CASE tools to cooperate or at least exchange data, access to widely used database tools (like report generators), and an unrestricted querying capability.

But for some time now there has been a flurry of research in the DBMS and CASE fields to develop new data models and technology to meet the CASE requirements[2] [33, 26]. Today we have working commercial products [34, 35] and research prototypes [36, 37, 38] that provide a wide choice of data models and technologies. This has broadened the data manager choices available, and we believe it is a good time to evaluate the data manager classes against the requirements to see how well the new systems address the needs of CASE.

Section 3.2 discusses and classifies the CASE database requirements, Section 3.3 discusses and classifies the data manager choices, in Section 3.4 we look at how well each data manager choice satisfies each class of requirement, and we close in Section 3.5 with a summary of our accomplishments in this chapter.

3.1.1. Contributions

Several authors have compiled lists of the database requirements of CASE environments, but their work has fallen short in that the lists have been informal, have not been adequately explained or justified, and they have not been related to specific data managers to see how well they satisfy the requirements.

In this chapter we make an important contribution in compiling requirements from different sources, eliminating those that were repetitive or not relevant, and augmenting them with new requirements that we formulated to support the CASE environments of the 90's and beyond. We also explain what each requirement is, and justify its inclusion in the list. Further we also classify the requirements according to their function, as well as according to whether they are inherent to the data model or implementation dependent. In addition, we classify past, and current data managers into classes. For each class we evaluate how well it fulfills each of the requirements and, based on this evaluation, we make recommendations as to which data manager classes to use in the design of new CASE environments.

3.2. CASE Database Requirements

In this section we list the main database requirements of a CASE environment. Several of these have been listed before [26], but this is the first time that they are explained and classified in a comprehensive fashion. We have divided the requirements into two categories, those specific to the data model, and those that refer to functionality but are data model independent.

Data Model: The data model forms the basis of the data manager. It specifies what the data manager entities are, how they relate to each other, what the operations on them are, and the type system among other things.

- *Extensibility:* It is impossible to include all features in the model that users want. Extensibility allows the data manager to be used in unanticipated ways and in new domains. Several other features provide extensibility in one or more dimensions, e.g. the ADT facility extends the system in the data handling dimension. But the data model needs to be designed with an explicit extensibility strategy in mind, like the five strategies that we identify in the next section.
- *Views:* Through the **view** mechanism, the system isolates the application from low level changes to the schema as well as providing alternate views of the same data. In the database sense, a view can be regarded as a *virtual relation* that does not exist by itself, but rather is defined in terms of other relations [39]. Views may or may not exist in physical memory (different systems may choose to *materialize* a view for

[2] Most of this research is not aimed specifically at CASE applications, but rather to scientific, text processing, and engineering applications in general. But CASE data management requirements have much in common with applications in those fields.

the sake of access time efficiency), and thus provide additional functionality without necessarily wasting space or having data consistency problems.

- *Object Encapsulation:* An important feature of object oriented languages, it must include data **and** operations. It also provides the advantages of modularity, where users of an object use its services regardless of its implementation.
- *Explicit Relationships:* When relationships (links) are an explicit part of the data model (like in the Entity-Relationship model [40]), they can be more easily represented, have more power, and can be implemented more efficiently.
- *Rich, Dynamic Type System:* Types that can have different kinds of a finite number of values (called union types), or that can have variable size (eg., arrays and ASCII fields).
- *Comprehensive ADT Facility:* Users should be able to define new types, new operators and indices on the new types, and be allowed to use these new types as members of other new types as well as in other type constructors (e.g., if I define a stack type, we should be able to construct arrays of stacks as well as stacks of arrays).
- *Type Inheritance:* Allows object types or classes to be defined as part of a hierarchy, inheriting the properties of its predecessors (e.g., a student may inherit from person adding new attributes like "major field"), or adding some constraints on these properties (e.g., a "manager" might have a constraint $10{,}000 > \mathit{salary} > 50{,}000$).
- *Database Procedures:* Having procedures as part of the DBMS is a simple but very powerful idea. It not only provides for a more powerful and flexible system, but also allows for better fine tuning. For example, we can couple them with data to encapsulate object semantics, as well as allowing data conversion operations to be performed where it is more appropriate, in the application or in the DBMS.
- *Process Programming:* A process program is one that specifies a series of operations on a set of repository entities. It describes the diverse software processes that the system enforces to develop and maintain the software documents. This support may come in the form of a process programming language [24] or a combination of other features like database procedures and a rules system.
- *Rules System:* The ability for the user to specify a set of **rules** about her data and other objects provides a knowledge management capability. Several important repository capabilities like views, referential integrity [41], and procedures can be provided by a general purpose rules system [12]. Thus a rules system would provide essential services as well as add significant, and needed, new capabilities.
- *Complex Objects:* A complex object is one that has a set of possibly different subentities linked in elaborate ways. They are omnipresent in CASE (e.g., an attributed AST is composed of nodes with links to other nodes as well as attributes). Support must include a mechanism to access and manipulate the subobjects.

Functional Attributes: These are the features that are not inherent in the data model. This means that they could be incorporated into systems using different models, and implemented in different ways.

- *Persistence:* The most basic function of the data manager, which must preserve data, objects, and all other supported entities as well as the structure and restrictions imposed by the system on the entities. Persistence must be orthogonal to all other properties. The main challenge is providing an invisible line between all the storage levels.

- *Concurrency and Distributed Access:* Concurrency allows many users to access the same database, while distribution allows the data to be located close to where it is used.
- *Extensible Access Methods:* Access methods allow us to access data fast. To efficiently support heterogeneous data and storage media (RAM, magnetic disk, digital drives), user defined access methods are needed.
- *Open Programmatic Interface:* The computer science and commercial communities have used, continue to use, and will continue to use a wide variety of programming (e.g., LISP, and C) and application program languages (e.g., Lotus 1-2-3). By making the repository programming language neutral it can be accessed by many more applications.
- *Active Data:* This is the ability for the user to specify that a certain operation be performed when a condition is satisfied. Sometimes called triggers, this ability can be used to notify applications of important events (i.e., incoming report of a *showstopper* bug). Active data can be implemented directly, or on top of a general purpose rule system.
- *General Querying Ability:* This includes both navigational and associative querying. A general query mechanism is essential to allow users to easily access desired data. This is one of the main contributions of current generation relational systems, provided through associative access. Navigational access, which was an integral part of the CODASYL databases, is needed for added flexibility and performance. In such a complex and large system as a CASE environment with different needs for different users, unexpected queries combining all kinds of data are inevitable and important.
- *Version and Configuration Support:* CASE documents are developed in stages over long periods of time. Each stage accomplishes a high level objective through a series of small changes to the document. In order to document and preserve the document at different stages, we create and name new versions that can be retrieved and queried in the future. A configuration is a complex entity, composed of specific versions of different objects. Since versions and configurations are essential aspects of the software lifecycle, direct support is needed.
- *Access Control:* In large, complex projects responsibility for different documents is distributed among many people. In order to exercise their responsibility over a document, the responsible party must be able to selectively grant different kinds of access to the objects that are part of the document. In addition, it allows people and teams to store private data in the system.
- *Bulk Data:* CASE applications require moving large amounts of data efficiently both to and from the repository. This can only be done by directly supporting bulk data operations, like the ability to read many tuples simultaneously in internal DB format. The input and output sources must include files as well as processes.
- *Transactions:* We require both short and long transaction support. We can currently support short transactions very effectively, and short transactions work very well for transaction processing systems. To support short transactions, systems lock every data item touched during the transaction until the transaction commits, at which time all locks are released. This model does not work well for CASE, where a user might want to check out a code module for days at a time, work on it, and then check it in. It is unreasonable to deny access to this module to all other users while it is checked out. Several alternatives are suggested by Kaiser [26].

- *Crash and Failure Recovery:* This protects sensitive data from loss and inconsistencies in the face of system problems. For example, if the system crashes in the middle of a transaction, at system initialization time there will be a recovery process that restores the database to the state before the transaction started.
- *Undo Capability:* This allows users to roll back a series of valid operations to a previous, valid database state. Essential for a small number of the most recently issued operations, since it allows for quick recovery and stimulates people to try new things.
- *Large and Small Objects:* The operations, access methods, storage organization, and processing overhead of large objects is very different from that of small ones. Explicit support for both granularities is required since CASE data can be both fine grained (e.g., document authors) and coarse grained (e.g., pictures in a user manual).
- *Meta-Schema Support:* The system must allow for easy access to the definitions and specifications of the data, procedures, rules, views and those of all other entities and constraints. This allows for efficient system evolution and makes it much more extensible, since definitions can be updated, added, or deleted efficiently.
- *Hierarchies:* Graphs are so pervasive in CASE that efficient manipulation and representation is very important. A rich type system and explicit links will allow for efficient representation, while navigational access support and pointer *swizzling* provide for efficient manipulation.
- *Multi-Level Storage:* The volume of data that DBMS's manage will continue to grow at a rapid rate, making it necessary to extend the data storage hierarchy beyond main memory and magnetic disks [42]. Not only does a Software System consist of many different documents, but each document will have many different versions, as well as be represented in several ways in the database. With large scale SSs in the millions of lines of code, developing a single SS generates massive amounts of data over its lifecycle.

In addition, the data model must be simple and powerful. Simplicity makes it easy to understand and implement, while power allows it to be widely applicable. The model should provide simple, efficient primitives on which more complex operations can be based. Finally, all these features need to be delivered with good performance, otherwise they will not be used.

We can see that the requirements are numerous and non-trivial. In some cases they can be satisfied by one or other features, for example process programming can be supported with a combination of DB procedures and a rules system, and the rules system can provide the view mechanism. The requirements can also be divided into categories according to what type of capability they address: data, object, process, or knowledge management.

Data management support refers to the traditional DBMS services like concurrency and crash recovery. Object management services support complex data needs, like abstract data types (ADT) and inter-object references. Knowledge support is essential to provide intelligent assistance to the software engineers. Finally, process programming is used to describe the software processes of the lifecycle so that a specific methodology can be followed[3].

As an example let's look at the lifecycle of a commercial DBMS. The system is large enough that it is developed by several teams of engineers. Data management features like concurrency support are essential to allow multiple developers to work on the same module. To store a complex data structure like the data flow graph of a program module, we need object management features like inter-object references. Knowledge management features like a rule system are necessary to specify and enforce constraints like *All code checked in has to undergo a path coverage test.* Finally, we need a process programming language to specify and enforce that the requirement specification phase is followed by the design phase in the development lifecycle. We now examine each category in more detail.

Data Management: Features found in the current generation of relational systems, and expected of a commercial strength DBMS. The data management features are:

Persistence.	Short Transactions.
Programmatic Interface.	Failure Recovery.
Views.	Undo Capability.
Associative Queries.	Concurrency.
Access Control.	Distributed Access.
Bulk Data.	Meta Schema Manipulation.

Object Management: Features geared to support the more complex data requirements of non-traditional DBMS applications like CASE and CAD. They are not specific to objects.

Extensible Data Model.	Navigational Queries.
Object & Operation Encapsulation.	Version & Configuration Support.
Explicit Relationships.	Long Transactions.
Types: Dynamic, ADTs, Inheritance.	Objects: Large, and Complex.
Database Procedures.	Multi Level Storage.
Extensible Access Methods.	Hierarchies.

Knowledge Management and Process Programming: Knowledge management includes features geared to the representation and management of knowledge activities, like rules systems. Process management features are those that support the expression and execution of different actions during the software lifecycle. The knowledge management and process programming features include:

Rules System.
Active Data.
Process Programming.

In the next section we discuss the different approaches to satisfying the CASE DBMS requirements we just covered.

[3]Process programming capabilities can be provided by a rules system, part of knowledge support. But this is also true of other requirements, e.g. a rules system can also provide database views [12]. We consider process programming important enough to list as a different requirement, regardless of how it is implemented.

3.3. Data Managers

In this section we look at what the different options for CASE data manager are. Our list is by no means exhaustive, but we cover what we consider the most important:
1) Build a custom DBMS from scratch.
2) Build a custom DBMS from a generator.
3) Use a persistent object store.
4) Use a relational DBMS.
5) Use a relational object shell.
6) Use an object oriented DBMS.
7) Use an extended relational DBMS.

Other possibilities include the Entity-Relationship DBMSs [40], DBMSs for CAD [43], the older network or CODASYL DBMSs [44] that were largely replaced by the relational systems, and the rule based DBMSs like LOGRES [45] and LDL [46] derived from knowledge representation languages. We now explain each category in turn.

1) *Build a custom DBMS from scratch:* A custom written DBMS like Cadre [3] and the Software BackPlane [2] seems at first glance to be what is needed, since it is unlikely that a pre-existing system will satisfy all the requirements of CASE, and one can design and develop a custom system that does what is needed. As a matter of fact, many CASE environments use custom DBMSs which satisfy a **subset** of the required features. They range from systems that just add a layer between the OS and the CASE tools [19], to those that provide a data model and more advanced capabilities [3]. Nevertheless, the subset usually includes: support for complex objects, *limited* querying capability, concurrency support, and persistent storage. The thing to note is the features left out: powerful access methods, *general* querying capability, support for multiple views, active data, and reliable storage. These are at the core of the services provided by general purpose DBMS's, and they are left out because they constitute the bulk of the code and complexity of these systems. In other words, they take many man years to build, refine, and maintain by highly skilled people [47]. Other disadvantages of the custom DBMS route are that an environment based on one cannot share data or tools with other CASE environments or with non CASE applications, and that applications built on one tend to be non-portable.

2) *Build a custom DBMS from a generator:* The systems like Exodus [48] and Genesis [49] allow the designer to generate a full function DBMS customized to her specifications. Customization can be done by allowing users to substitute different modules, or by providing a customized description of a specific module to a program generator. For example, Genesis allows the buffer and recovery management modules to be substituted to implement different strategies, while the EXODUS optimizer generator takes a description of a customized optimizer and generates it as part of the custom DBMS. This seems like a very attractive option, but it has several problems. It requires that substantial parts of the DBMS be coded by the application developer. For example, to implement a customized DBMS with EXODUS we use the database toolkit and the E persistent programming language to supply a data model, query language and parser, catalogs, access methods, data storage structures, and query optimization and query execution strategies [50]. In addition, this approach may make it difficult to share and integrate data from other application domains.

3) *Use a persistent object store:* Systems like Kala [51] and ObServer [52] provide persistent objects and object references and a few other sophisticated features. For example, Kala provides a low level, untyped storage manager, with the primitives to implement specific transaction, access control, and version and configuration models. But they lack many other sophisticated features like rules, procedures, query language, etc.

4) *Use a relational DBMS:* Systems like Ingres [53] and Oracle [54] have many desired features, including those that are usually left out by the custom written ones: concurrency, crash recovery, associative queries, etc. In addition, the relational model is simple and is so widely used that many applications have been built for it. Nevertheless, the relational DBMS's have several problems that have precluded their use as CASE data managers: inadequate performance, little support for hierarchies, a non extendible data model, and lack of triggers. These problems force the CASE data [14] to be mapped in an elaborate and unnatural way to the DBMS. In addition, relational systems provide slow performance for complex CASE operations. For example, Omega [14] reports that, with a specially tuned DBMS, it took 7 seconds elapsed time to fetch a 5 line procedure from the DBMS.

5) *Use a relational object shell:* Systems like the Ingres Object Management Extension [55] and Penguin [56] provide object management on top of a relational DBMS. This approach adds substantial new functionality, especially dealing with object support, but also inherits several of the relational systems weaknesses, like poor performance.

6) *Use an Object Oriented (OO) DBMS:* Systems like Iris [38] and Gemstone [57] have been designed with the needs of complex data in mind. The features that any respectable OODBMS must have include "is-a" hierarchies (inheritance), object identity, encapsulation, and tight integration of data model with a programming language [58, 59]. Many OODB designers start from an OO programming language and extend it by adding traditional DBMS features like object persistence, and query capabilities. This approach leads to architectural decisions (e.g., including pointer *swizzling*) and data model features (like tight coupling to a specific OO programming language like C++) that differentiate them from the extended relational systems. They provide great support for the navigational style of the CODASYL DBMSs, in which the user needs to know and specify the path to all data items. The new OO databases like O2 [60] include features that some of the first and more specialized systems omit, like associative queries, and good support for traditional business data applications.

7) *Use an Extended Relational DBMS:* Systems like POSTGRES [13] and Starburst [61] are safely anchored in the relational model, with extensions to remedy its known weaknesses in object and knowledge management, like object encapsulation, rules system, and performance tuning mechanisms like fast path. In contrast to the OODBMSs, they start from a DBMS point of view and extend it with sophisticated object, and knowledge management features. For example, POSTGRES [13] combines the advantages of the Relational model with extensions to support new applications: active data (triggers), extensibility throughout the system (data model, access methods, etc), a procedure data type, mechanisms to enhance performance (fast path and precomputation by background asynchronous deamons), historical data, support for versions, and others. Other advantages of the extended relational systems include the existing

variety of existing tools and applications, the coexistence with business data in the same repository allowing "hybrid" applications and queries, and an attractive migration path to the myriad existing applications for relational systems.

Extensibility And Customization Strategies: There are several strategies that these systems use to provide customization and extensibility. We have identified five:

1) Provide low level primitives for a specific feature, giving users a mechanism, and allow them to implement their own policies. This strategy is appropriate for areas that have no widely accepted policy, like Kala [51] does for transaction support. This approach can have important disadvantages: i) More work on the part of the system designers (both in design and development) ii) The policy implemented, if non-standard, may lead to incompatibilities with other systems, even those implemented with the same primitives. On the other hand its virtue is most evident in features for which there is no clear consensus as to what policy is best, like transaction support and configuration management.
2) Use a module generator approach. The module generator is given a well defined description of the subsystem, and it produces the code that implements the desired behavior. An example of the generator approach in a different domain is the Yacc [62] parser generator, and in the DBMS area we have the EXODUS [48] optimizer generator to generate the optimization code. This approach works very well in narrow domains with well understood algorithms like parsers. Its disadvantages include the use of a possibly different language in the generator and the rest of the system, as well the difficulty in incorporating special purpose actions in the module generated [50].
3) Provide an interface and mechanism to allow applications to integrate their own policies. For example, to define a new access method in POSTGRES [13], the system designer writes and provides the key functions to the system. which takes the necessary steps to incorporate them into the optimizer. That is, POSTGRES identifies the features that differentiate one access method from another, and provides an interface so users can easily provide their own access method by specifying the items that make it unique.
4) Design and implement the system in a modular fashion, with well defined interfaces between modules. Users can substitute one module for another. This is the approach used in GENESIS [49] to customize buffer management. That is, GENESIS provides a variety of buffer management strategies, each encapsulated in a module. At system installation time we choose which strategy to include. This approach is nice in that it allows us to choose between several standard strategies. On the other hand it does not provide for user defined strategies.
5) Provide a flexible, open interface so that the system designer can use it to incorporate her own code. For example, ObServer [52] provides persistence for objects and object references, with an interface so the system designer can implement her own query language. The disadvantage is that the user has to provide the module even when she wants a standard implementation. This approach combined with a system supplied choice of standard modules is a really attractive combination.

3.4. Data Manager Classification

In this section we classify the data manager classes discussed above according to how well they meet the CASE repository requirements. It is hard to pinpoint which class of managers satisfy the functional requirements, since many different implementations exist for each class, and each implementation may be different. We therefore use the criterion that a class of managers satisfies a requirement if we know of at least 1 implementation that does.

In Tables 3.1, 3.2, and 3.3 we list the data manager options as the columns, and different features as the rows. Table 3.1 lists the data management features, Table 3.2 lists the object management features, and Table 3.3 lists the knowledge and process management features. The entries have a value of **NO** if the class of servers do not support the specific feature, of **LIM** if they provide limited support for that feature, of **PRIM** if the class supplies primitives that allow for user implementation of a specific policy, and of **FULL** if the feature is fully supported.

Features	Custom	Custom Generator	Object Store	Relational	Relational Object Shell	Object Oriented	Extended Relational
Persistence	FULL	FULL	FULL	FULL	FULL	FULL	FULL
Distributed Access	FULL	FULL	FULL	FULL	FULL	FULL	FULL
Bulk Data	FULL	FULL	PRIM	FULL	FULL	FULL	FULL
Short Transactions	LIM	FULL	PRIM	FULL	FULL	FULL	FULL
Access Control	FULL	FULL	PRIM	FULL	FULL	FULL	FULL
Programmatic Interface	LIM	PRIM	PRIM	FULL	FULL	FULL	FULL
Views	NO	PRIM	NO	FULL	FULL	FULL	FULL
Associative Query	NO	PRIM	NO	FULL	FULL	FULL	FULL
Undo Capability	FULL	PRIM	PRIM	FULL	FULL	FULL	FULL
Concurrency Support	FULL	FULL	FULL	FULL	FULL	FULL	FULL
Crash Recovery	LIM	PRIM	NO	FULL	FULL	FULL	FULL
Meta Schema	LIM	PRIM	NO	FULL	FULL	FULL	FULL

Table 3.1: Data management features of different models.

Features	Custom	Custom Generator	Object Store	Relational	Relational Object Shell	Object Oriented	Extended Relational
Extensible	LIM	PRIM	PRIM	NO	LIM	LIM	FULL
Object Encapsulation	FULL	FULL	FULL	NO	FULL	FULL	FULL
Explicit Relationships	LIM	FULL	PRIM	NO	LIM	FULL	LIM
Dynamic Types	LIM	PRIM	NO	NO	FULL	FULL	FULL
ADT	FULL	PRIM	NO	NO	FULL	FULL	FULL
Inheritance	FULL	PRIM	NO	NO	FULL	FULL	FULL
DB Procedures	FULL	LIM	NO	NO	FULL	FULL	FULL
Complex Objects	FULL	PRIM	FULL	NO	FULL	FULL	FULL
Navigational Queries	FULL	PRIM	NO	NO	FULL	FULL	FULL
Version Control	FULL	PRIM	PRIM	NO	NO	FULL	FULL
Long Transactions	FULL	NO	PRIM	NO	NO	FULL	NO
Large Objects	FULL	PRIM	FULL	NO	FULL	FULL	FULL
Multi Level Storage	NO	NO	LIM	NO	NO	LIM	FULL

Table 3.2: Object management features of different models.

Looking at the three Tables, we can spot several trends. Almost all the classes satisfy the data management requirements, except for the Object Store class which provides a small subset of primitives, and the Custom class which omits difficult to implement features like associative queries.

The object management features are satisfied very well by the extended relational and object oriented classes. The relational object shell covers most of them, while the relational class covers none. The custom generator class provides indirect or limited support to most of the object management requirements, and the custom class provides full or limited support of most of them. The Object Store class provides indirect support for about half of these features.

The relational, relational object shell, and Object Store classes provide no process programming and knowledge management support. The custom, custom generator, and object oriented classes provide some support for these features, but it is only the extended relational class that fully supports them.

From this classification, we conclude that there are two data manager classes that satisfy most of the requirements and require the least amount of work to use: the object oriented and extended relational DBMS classes. They either already have the features, or they are being currently added to research or commercial systems. The main impediment that we see to their success is how well they perform in production environments. But, we believe that the performance problem can be addressed through careful system architecture choices and judicious database designs. We address these two issues in later chapters.

3.5. Summary

In this chapter we have listed, explained, and classified the CASE database requirements. We also looked at the main data manager classes, and classified them according to how well they satisfy each of the requirements. The list of requirements can be used to guide system implementors in choosing which to include in their system, as well as to which will be inherent in the data model they choose. It also points out how large and complex a CASE data manager needs to be. The identification of the major data manager classes, as well as their classification against the CASE requirements can be used as a guide in choosing the data manager for a new CASE environment. The major conclusion is that the object oriented and extended relational DBMS's currently satisfy well most of the CASE database requirements, and that they are evolving in the right direction so that they will provide all the power needed by CASE

Features	Custom	Custom Generator	Object Store	Relational	Relational Object Shell	Object Oriented	Extended Relational
Active Data	FULL	LIM	NO	NO	NO	LIM	FULL
Rules System	NO	PRIM	NO	NO	NO	NO	FULL
Process Prog.	FULL	PRIM	NO	NO	NO	LIM	PRIM

Table 3.3: Knowledge and process management features of different models.

environments in the next few years. Our work in this area has been very useful in the design of TULUM, which we cover in the next chapter.

CHAPTER 4

SYSTEM ARCHITECTURE

4.1. Introduction

In this chapter we look at different CASE environment architectures, both physical and logical, and then go on to discuss the physical architecture, the logical architecture, and the computation paradigm for TULUM, our proposed new CASE environment. Our objective is to evaluate the different existing systems, their contributions and shortcomings. to make sure that TULUM incorporates those features that we like, and that TULUM addresses their shortcomings.

We take the view that a CASE environment is a collection of tools that take certain inputs and produce outputs. Both inputs and outputs are data. The outputs may be the (possibly) modified inputs, or new data items. The tools may also delete some of the inputs, as well as produce side-effects. In addition to tools and data, there is a knowledge component, whether it is embedded in the tools, the data, provided by the users of the system, or a combination of forms. Finally, software is produced and managed by a series of active processes, like having software satisfy a code coverage test before it can be released. We therefore have that a CASE environment contains tools, data, knowledge, and processes. This view is general, and leaves many issues unresolved, such as whether tools are integrated with each other, or if the environment supports and enforces processes directly. These issues are specified by the architecture of the system. Three aspects are of major importance in designing a CASE environment: the physical architecture, the logical architecture, and the computation paradigm of the system.

The physical architecture specifies the hardware components of the system, how they are connected, how the different software components are distributed in the hardware, and how they communicate with each other. The logical architecture of the system specifies how its elements are represented and what the relationships between them are. The computational paradigm specifies the philosophy underlying the implementation of the system, like the use of parallelism and incremental algorithms.

4.1.1. Contributions

Several existing and proposed architectures are examined, and their major weaknesses are pointed out. But we also make note of their good features. A major weakness of these systems is that they were not designed to fulfill a comprehensive list of requirements like the one we developed in Chapter 3.

We design a new CASE environment architecture, named TULUM, that explicitly addresses the weaknesses that we uncovered in previous architectures, as well as fulfilling the requirements that we listed in Chapter 3. An innovative feature of our design is the explicit attention that we devoted to the physical, and logical aspects, as well as to the computational paradigms used in the environment. By making these three aspects explicit and making sure that each complements the other two, we produce a more balanced architecture.

4.2. Logical Architectures

The logical architecture of the system specifies how its elements are represented and what are the relationships between them. More concretely, what are its elements, how are they viewed and how do they interact? For example, do tools interact with each other and, if so, is it via an intermediary or directly via messages. It also specifies how the users interact with the system. For example, in order to customize the system do we modify a set of rules and data schemas, or a process program. There are several logical ways of structuring the architecture of a CASE environment [19, 63, 64, 24]: i) tool centered, ii) data centered, iii) knowledge centered, and iv) process centered. We cover each in turn.

1) Tool Centered

In the tool centered architecture each tool has its (possibly) distinct data representation and makes arbitrary assumptions about how data is represented and stored. In other words, a tool centered approach is no more than a collection of independent tools used during the program lifecycle. It is typically file based, knowledge is scattered among the tools, and each tool uses its own data format and management. All lifecycle process information is supplied by the users, be it in the way of conventions, in the way that specific tools are used to accomplish a goal, or with the aid of interface programs and scripts to enforce and support those conventions. Due to its inherent structure, an independent collection of tools, this architecture lacks and cannot provide basic functionality like concurrency or transaction support. Figure 4.1 illustrates this architecture, in which tools (represented by ovals) operate on files (represented by squares). Most software development work is done in this kind of environment. For example, *emacs* is used to edit[65], *RCS* to support version control [66], and *make* to compile and link programs [67].

This approach is simple to conceptualize and implement, and is the one used in most current program development environments. In addition it extends simply by adding more tools to the toolkit, or by replacing existing tools with new ones. But its disadvantages include:

i) Common services for data objects are provided independently by each tool that needs them. For example, if an editor and a debugger want to provide multi-user access to a file, they each have to implement it. A compiler and a debugger might both have their own parser to convert the source to a form they can easily manipulate.

ii) There are differences in the semantic content of similar objects created by different tools, For example, a symbol table that is created by a compiler may not contain cross reference information, while one created by a debugger would.

iii) The tools work from the lowest common definition of data objects. That is, since tools are written and work independently from each other, they assume a commonly available, primitive, representation of the objects that they use as input. In the case of programs, this definition is usually the source code.

iv) There is processing redundancy between tools leading to inefficiencies. For example, the compiler produces a symbol table, while the debugger might also produce a symbol table.

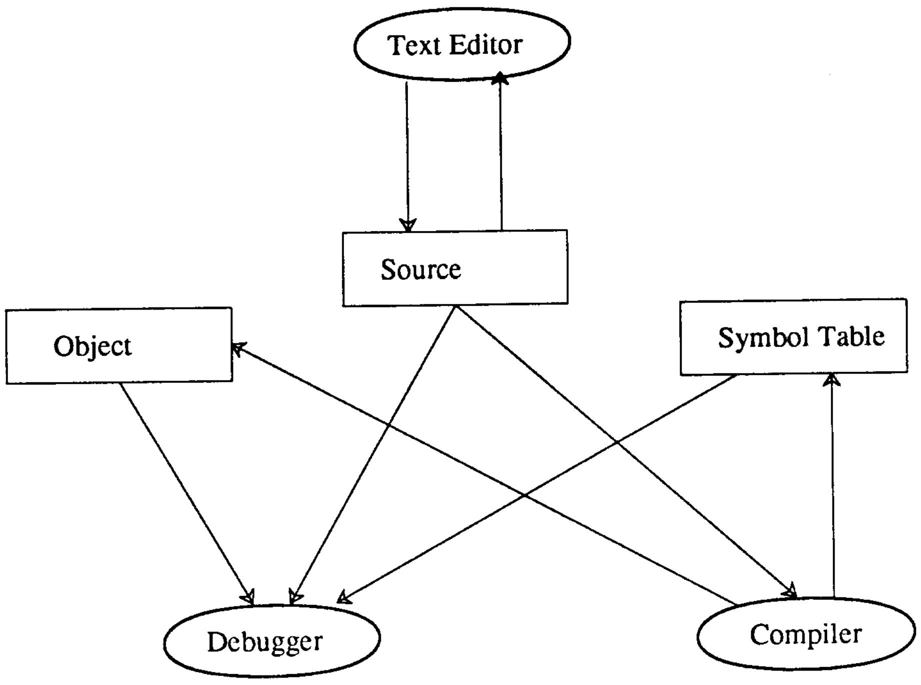


Figure 4.1: Tool centered architecture.

2) Data Centered

The data centered approach [19, 63] defines a common data format as the interface between tools. Tools need to abide by the common format in order to be able to interact with each other. As in the tool centered approach, there is no specific environment support for knowledge or process programming. Conceptually, it has at its core a common repository that stores, manages, and provides sophisticated services to the data produced and manipulated by the different tools, as depicted in Figure 4.2. A relational database like Ingres [68] or Oracle [54] can provide the repository services. Integration is achieved by defining a common interface used by all the tools. The data manager becomes the medium of communication and coordination between tools, making it the logical and physical foundation on which the environment rests. In this architecture a CASE environment is simply a device to use tools to manipulate the common data repository. The advantages of the data centered approach include:

i) Common services provided by a single source,

ii) The repository can maintain the core semantic content of objects, and provide different views to different tools,

iii) Improvements to the repository bring benefits to all tools,

iv) Tools can draw on many views of the data, on data about all facets of the project, and even on other data maintained by the same repository,

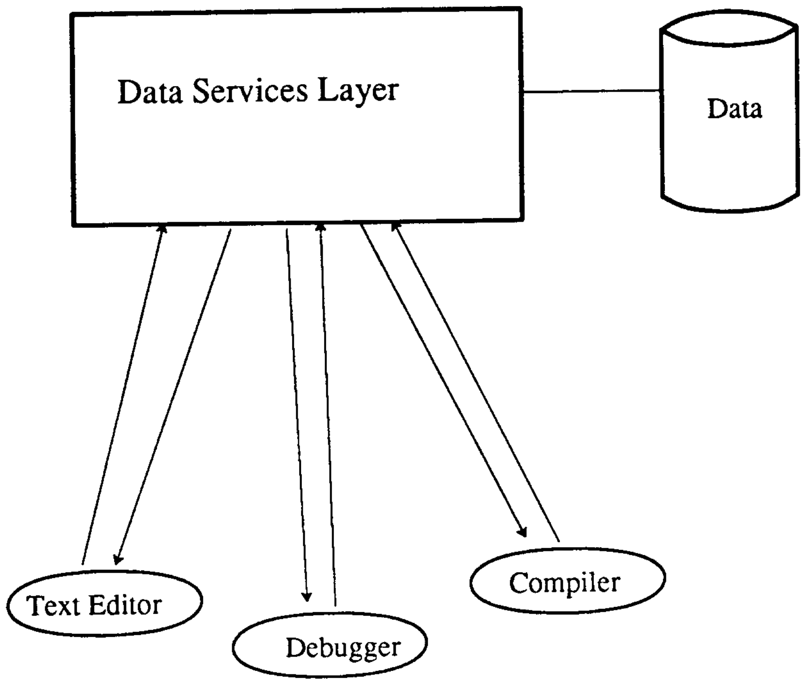


Figure 4.2: Data centered architecture.

v) The repository provides the *glue* that unifies the tools into a cohesive whole,

vi) The main memory / secondary storage boundary is efficiently managed by the repository, therefore all queries performed use all the data in the system (compare this with the architecture in which the system resides and works on objects in main memory only), and

vii) Tools are simpler because they do not have to implement the storage abstraction, and more powerful because they can be more easily integrated.

Its main disadvantages are that it does not explicitly acknowledge the importance of knowledge support and process programming. This leads to a major weakness when incorporating more advanced features into the environment, for example specifying a set of formal methods to be used during system maintenance. If knowledge and process programming support are not *first class citizens* in the data model, it is likely that their functions will be relegated to secondary status too.

3) Knowledge Centered

In the knowledge centered architecture, knowledge and data services are provided by the same, logically centralized, layer, with software engineering knowledge becoming the main focus. See [69] for a classification of the different kinds of SE knowledge. Conceptually, the repository becomes active with inferencing capabilities that can be provided by a rule system as that in [12]. The knowledge centered architecture is illustrated in Figure 4.3, which is identical to Figure 4.2 except that it bundles knowledge and data services in the same layer. A next generation DBMS like POSTGRES [13] can provide the required services. The advantages of incorporating the data and knowledge managers in a conceptually centralized repository include:

i) The rules system (part of the knowledge component) can be used to provide many services, like maintaining multiple data views synchronized.

ii) The *memory* of the knowledge system includes all operations on the data.

iii) There is no *impedance mismatch* between the rules system and the data manager. That is, data does not have to be copied and converted between the knowledge and data managers.

Although knowledge support could be used to provide process programming capabilities, the main disadvantage of this architecture is that it fails to explicitly account for the importance of the processes involved in the software lifecycle. That is, by not making processes *first class* objects in the data model, their handling is not accorded the necessary importance and is done in an ad-hoc manner.

4) Process Centered

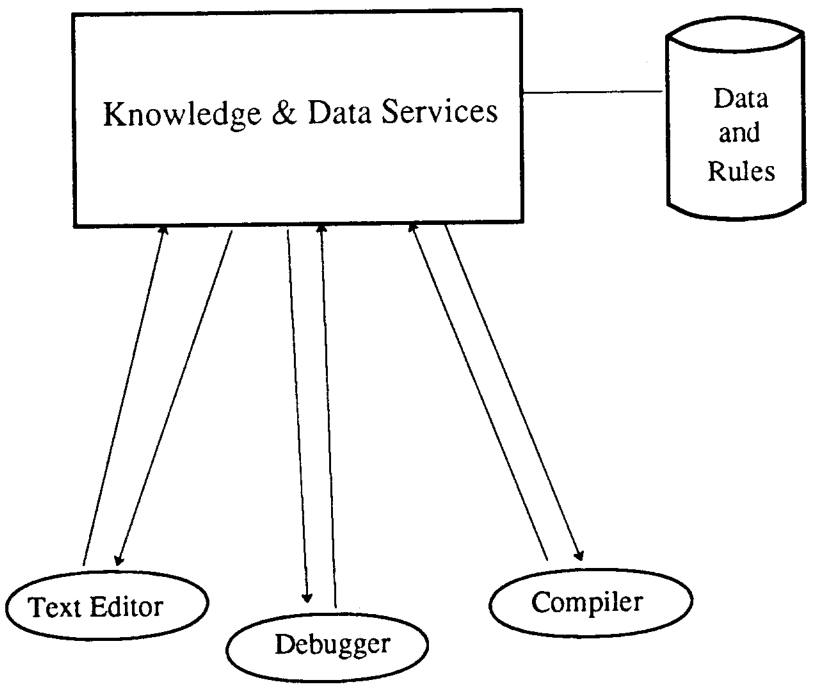


Figure 4.3: Knowledge centered architecture.

In a process centered approach, the software lifecycle processes become the focus of the environment. It is similar to a knowledge centered approach, in that the data manager also provides data services, but it also explicitly makes the process the unifying theme behind the environment design activities. Process programs describe the software processes to develop and maintain software systems, the tools are viewed as functions or operators, and the software documents are the operands and products of the tools and users. The main shortcoming of the process centered approach is the lack of explicit knowledge support.

The Arcadia project [24] is developing a process centered environment architecture to maximize flexibility, extendibility, and integration with a process programming language as powerful as a general purpose one. Flexibility comes from modifications to the process programs, while extensibility comes from writing new programs, and extending existing ones to support new tools, subprocesses, objects, or types.

There are several ways of supporting process centered environments, including a procedural process language as in Arcadia, declaratively with a rules system like the one in Postgres, or with a declarative language as in Meld [70]. It is not clear which approach is "best".

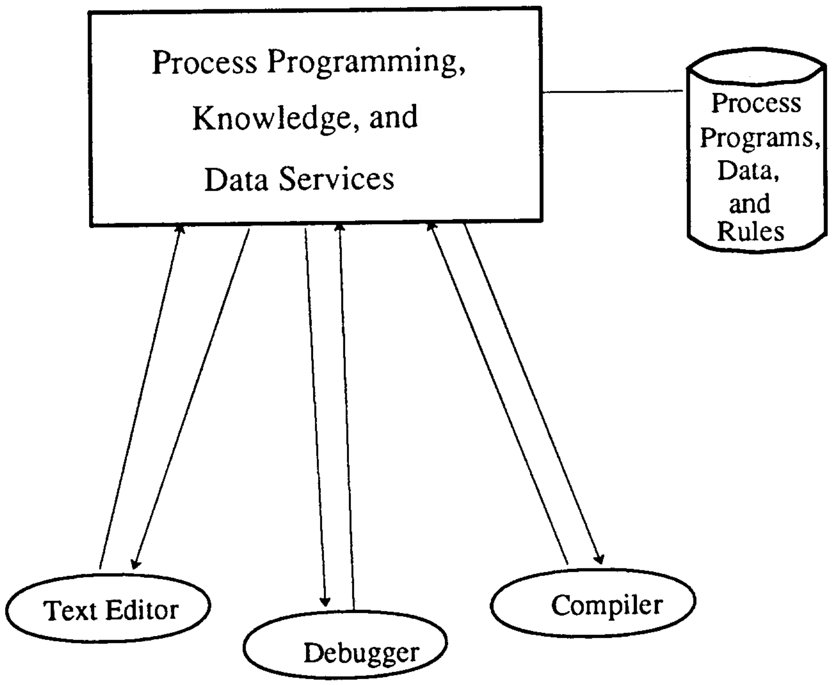

Process Programming, Knowledge, and Data Services
Process Programs, Data, and Rules
Text Editor
Debugger
Compiler

Figure 4.4: Process centered architecture.

4.3. Physical Architectures

The physical architecture of a system includes a hardware and a software component. The hardware aspect is how many computers are involved, how they are connected, how many IO ports there are, etc. The software aspect includes how many processes there are, how they are distributed in the hardware configuration, and where data is stored and how. We cover the most important physical architectures here: i) independent tools with files, ii) single user, integrated environments, iii) multiple user, single system, and iv) client-server.

1) Independent Tools with Files.

All files and tools reside on the same system, or tools use remote file access via the network. This is the physical equivalent of the tool centered architecture, and meets very few of the CASE requirements. Tools run independent of each other in the same or different processor with access to the same file system. This is how most software development takes place today in systems like UNIX[4] [1].

Two systems, Field and Odin, add a layer to the independent tools that provides an integration framework, both in the context of a workstation network environment. The approach used in Field [71] uses selective broadcasting in which tools communicate via a central message server. Each tool registers with the message server the kinds of messages it is interested in getting, and the tools communicate by sending messages to the server, who in turn forwards them to the interested tools. This kind of simple, loose integration is attractive since it simply and cheaply provides extra power to existing tools. On the other hand, it fails to provide many of the sophisticated services that CASE requires like transactions and abstract data types.

The approach followed by Odin [19] is more elaborate and provides more power. Odin is an environment integration system that uses a centralized store of persistent software objects to integrate existing tools, adding many of the advantages of the data centered approach. It provides for the typing of software objects, tool composition out of modular software fragments, automatic rederivation of software objects, and isolation of tool connectivity information in a single object for easy maintenance. Odin is a very interesting system and satisfies many of the object and knowledge requirements of CASE environments, like active data and large object support, but also fails to provide other important requirements like transactions and crash recovery.

2) Single User, Integrated Environments

This architecture is illustrated by systems like Interlisp [5] and Smalltalk [72], that provide a powerful, single user, program development environment in which the user has a single interface and the tools communicate and share data effectively, hiding the file system interface from the user. Tools and data are designed to work together,

For example, within Interlisp, MasterScope (a program information system) is integrated with the editor, debugger, and other tools. With it, the programmer can ask to see all the statements that reference a global variable, and the editor will position the cursor on each statement in turn. Due to its integrated design, Interlisp is the only

[4]Unix is a trademark of Bell Laboratories.

interface with which an Interlisp programmer interacts when using its various tools. Although this architecture is a significant advance over the independent tool approach, it is inadequate for large scale SE due to a lack of important features like multi-user support, crash recovery, and version support. In addition, these environments are not easily extensible and do not interface well with existing tools.

3) Multiple User, Single System.

This architecture is similar to a multi processing operating system, in which a single program runs on one CPU and handles multiple users through multi processing techniques. The DBMS and the applications run on the same computer. Omega [14] used this architecture with Ingres providing the CASE database services.

This architecture fails to capitalize on the trend toward distributed computation via connected workstations. In addition, it severely stresses the system resources since the CPU, the memory, and the IO bandwidth are shared by both the DBMS and the applications.

4) Client-Server

In the mid-80s a new trend in system architectures began, in which a powerful computer with efficient IO channels and ample memory (the server) manages disks and other peripherals, and is connected via a network to other computers (the clients) that access the resources it manages through their network connection. This trend, fueled by the emergence of cheap, fast workstations with large amounts of memory gave rise to the client-server architecture. In the traditional client-server setup we have 1 or more servers communicating over a LAN with N clients. The tools run on the clients and send all repository requests to the server over the network. All caching on the clients is done by the individual tools.

This architecture has several resource bottlenecks that limit its throughput as the number of clients increases including data contention, IO, and server CPU. In demanding applications, efficient DB access in multiuser environments requires that each user have a dedicated DBMS server with direct access to the DB information [73]. In addition as databases grow it becomes impractical, in terms of time and space, to load the entire database to virtual memory, and therefore the databases need to be "segmented" according to users' needs. so that only the relevant segments are cached by users. Two variants of the traditional client-server architecture, RAD-Unify and *ADMS*$\pm$, have recently been introduced, in order to address these concerns by shifting more work to the clients.

The RAD-Unify architecture [74] moves query optimization and execution to the clients. The server only performs low level operations like locking and DB IO. During query processing, all IO requests are forwarded to the server who performs the required IO operations and forwards the desired data to the client who then uses its virtual memory to continue its query processing. Under a few scenarios, namely small to medium size databases with a single writer, its performance is significantly better than the traditional architecture. But, since it does not affect the biggest client-server bottleneck, the server I/O bandwidth, its improvement is marginal in more realistic scenarios [75].

ADMS$\pm$ [27] introduces the Enhanced Workstation Server (EWS) architecture, which distributes most of the query processing **and** the IO load to the servers by incorporating a full function DBMS on the clients which cooperates with the server DBMS, as well as by downloading and caching results in the clients. In addition, it uses incremental computation, backlogs, and differential files to minimize the network traffic and load on the servers. In simulation experiments done in [75] the EWS architecture performs much better than the traditional and the RAD-Unify architectures (in some cases more than an order of magnitude) and also scales up much better. EWS has two advantages over the traditional client server and the RAD-Unify architectures:

i) The ability to perform concurrent and parallel access of cached data from multiple client disks which reduces server I/O load and blocking, increasing transaction throughput.

ii) An increase in client autonomy, with lazy or periodic update propagation allowing clients to continue working over short server disconnections.

Unaddressed Issues: Several issues are not addressed by the above architectures. These include multi-level storage of data, and DBMS-toolkit communication.

DBMS-Toolkit Communication: The DBMS and the client processes run in separate address spaces, and therefore any communication and transfer of data between them implies some form of inter-process communication plus at least one copying of data from the DBMS cache to the application address space. This is done for reasons of data integrity and toolkit autonomy. With shared memory, the application can corrupt or alter the database without control from the DBMS. In addition, if the application accessed the data in the DBMS address space, it would need to know the low level DBMS data format, and thus would make the toolkit and application code dependent on low level, internal DBMS formats that could change. For these reasons, the shared memory approach is not acceptable and a different solution is needed.

Multi-Level Data Management: In addition, traditional DBMSs are optimized to handle data on secondary storage, so that both the way the relations and the access methods are organized is very efficient when the data is on disk, but not as efficient when the database is in the cache. For example, as indicated in [42] a B+ tree may be optimal for database access when almost all the DB is on disk, while an AVL-tree or a T-tree may be more efficient when almost all the DB is in memory [76,77]. That is, the optimal access method for the same database is different when the database moves along the storage hierarchy. Further, the optimal representation for a given database may also be different at different levels of the memory hierarchy. For example, a large tree node database may be in compressed form while on tertiary storage, and in uncompressed binary format in the DBMS buffer pool. In addition, the size of the unit of manipulation also differs at different storage hierarchy levels. Finally, as data is transferred from the DB to the application, it might need to be modified. For example, the pointers used on disk (object id's) can be "swizzled" to a main memory address when the database is brought into memory, as several object oriented DBMSs currently do [26]. The performance results that we show on Table 6.2 show that the only system that caches objects in main memory format (the OODB) outperforms all others in *warm* (all data cached) operations by over a 2 to 1 margin. With the architecture of TULUM we directly address these issues.

4.4. TULUM System Architecture

In this section we present the physical architecture and logical work organization of **TULUM**, our proposed CASE environment. The objectives that we address with this architecture are:

- Explicit mechanisms for the heterogeneous communication needs.
- Efficient cooperative, multi-user support on shared data.
- Efficient single user support on private and shared data.
- Object, data, knowledge, and process programming services.
- Graceful scaling up.
- Extensibility.
- Performance.

With this architecture we aim to support the different kinds of CASE user needs, as well as to avoid bottlenecks in the flow of information to and from the database when the number of users accessing the database increases or the database grows. The physical architecture is an enhancement of the EWS model, the logical architecture is process and knowledge centered, and the computational paradigm uses incremental computation methods, lazy evaluation, and parallelism. We cover each in turn.

4.4.1. Physical Architecture

Given the rapid and accelerating trend toward networks of powerful workstations connected via high speed links, with access to common resources (disks, printers, etc) managed by one or more servers, we believe that a client-server architecture offers the best fit between the CASE environment requirements and the physical computing environment. But, the classic client-server architecture falls short on two requirements: scalability and speed. The speed limitation comes from the fact that all repository requests (read or write) are processed over the network, while scalability is limited by resource contention on the network and server, especially the IO bandwidth. We therefore use many of the enhancements from the *ADMS*$\pm$ EWS architecture [27] [5]

We address both the speed and scalability requirements by having a private, full function DBMS, DBMS-, reside on the clients, with a coordinating DBMS, DBMS+, on the server. The client, DBMS-, caches data used by the application, and uses it to do as much local query processing as possible. In addition, incremental computation methods and differential files are used to minimize the amount of data transmitted between clients and servers. Data caching, local query processing, and incremental computation are geared to increase repository response times. We use the object and data manager to provide lower level integration among tools and processes, and a user interface management system (UIMS) to provide integration for the users so that user interaction is uniform, consistent, and appropriate. *DBMS*$\pm$ manages the components of the products produced, and the tools and information structure that constitute the environment itself. It provides the underlying mechanism for object, knowledge, process, and data management. This architecture scales up well because local client caches allow for parallel

There are many issues that the *ADMS*$\pm$ client server approach raises, including consistency between client and server caches. These are outside our scope, and are covered in [27].

data access without IO contention, and incremental computation and differential files lower server IO contention. Figure 4.5 zooms in on a single server and client, showing how the different subsystems are distributed, as well as the different places where data is stored (server disk, server memory, etc). We now cover the contents and services of the clients and servers.

Server: Full Knowledge and Database Management System.
Optimized to support many clients.
Optimized for I/O intensive operations

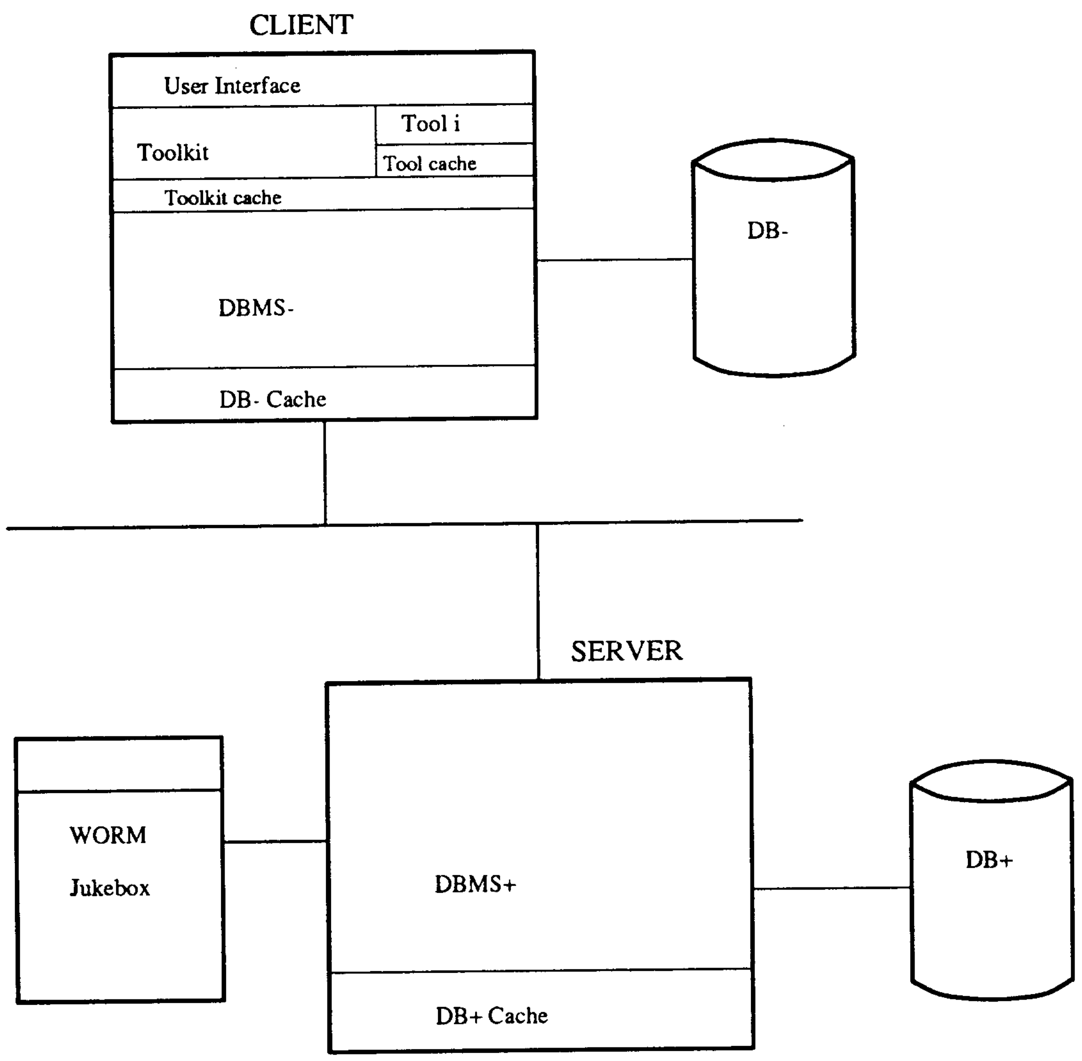


Figure 4.5: Architecture of TULUM.

Multi-Level data support.

The server manages the shared knowledge and database that consists of schema definitions (access methods, views, rules, relations, procedures), relations (classes), procedures, and rules. The services it provides include concurrency control, queries on non-cached data, crash recovery, transaction support, and security, In addition, it keeps track of data cached by specific clients, and provides support to the clients that cache this data as well as others that request access to this data. Finally, it also provides explicit multi-level store [42,25] support, which we illustrate in Figure 4.5 by showing the WORM jukebox at the bottom of the storage hierarchy.

Client: Single user K/DBMS without concurrency control or security.
Caches a relevant subset of the knowledge and data base, and maintains a private database.
Optimized for interactive response from local disk and memory.

The client manages a cached knowledge and database subset, together with the local private database. It runs the user interface code, as well as the environment tools required by the task at hand. The services provided include tool and application computation, queries to the cached and private database, execution of the user interface manager, and interaction of the environment with external processes. Each tool maintains a private object cache with objects that are temporary or will be saved to the DB later on.

4.4.2. Logical Architecture

We use a knowledge and process centered organization. This organization has enough power to fulfill many of the CASE environment requirements, combining data, object, process, and knowledge services in the repository. We think that a hybrid approach is best because both data, knowledge, and process are essential parts of the environment and they all should be given explicit attention when designing, extending, or customizing the environment. A general purpose next generation DBMS [78,79,58] serves as the knowledge, process, object, and data server. It provides support for customization, uniform policies, and extensibility. For example, the rule system and database procedures can be used to customize the repository for CASE; database procedures and fast-path access can create a uniform tool interface; and the system can be extended by defining new rules, procedures, and modifying the meta-schema. This architecture is extensible, since it allows the integration of new objects, tools, and processes. Integration is provided through the *meta-schema*, the rule base, and the procedure base. In addition, the next generation storage manager can easily support a tool, data, knowledge, or a process centered environment. The tool centered organization can be provided by storing whole files in the repository, and adding a layer between the tools and the operating system that routes the appropriate file system calls to the repository. The data and knowledge centered organizations map directly onto the repository. Further, the process centered organization can be supported by using the rules system in combination with database procedures and fast path access to implement or simulate a process programming language. Finally, this architecture can also implement a "hybrid" organization, for example one in which both process and data are central.

In this system model, the users interact with documents through the user interface, the tools interact with documents through system services (including those provided by the repository), and the tools communicate with each other directly or through the active data repository.

4.4.3. Computation Paradigm

The computational philosophy or paradigm that is used within the CASE environment is very important, and must be carefully chosen together with the physical and logical architectures to satisfy as many of the CASE environment requirements as possible. That is, the computational paradigm has to be chosen to complement the architecture. Our approach is based on three principles: lazy reanalysis, incremental computation methods, and the exploitation of parallelism.

Our system uses a **lazy** reanalysis and update policy, assuming that users will invoke reanalysis when desired, or when one of a series of events occurs. This policy is important for several reasons. First it allows the user or system to determine what and when to save data to the repository, it provides for high performance needs in exchange for having larger amounts of work lost in case of failure, minimizes the amount of transient data that is unnecessarily stored, and allows for only the final of a series of updates to a given item to be saved.

In addition, the system minimizes the cost of updating and reanalizing through the use of incremental algorithms throughout the CASE system. For example, as we mentioned above, the DBMS- to DBMS+ module communication is done via incremental algorithms and differential files. We extend this approach to the software document manipulation operations, for example by using the incremental parsing and tree-building algorithms in [80] to keep the source synchronized with the AST, and provide real time syntactic feedback to the users.

Further, we exploit parallelism in various places. For example, in getting the large amounts of data from the server to the clients that result from query processing, the server overlaps query processing with data transmission so that the data arrives at its destination almost at the same time as the query finishes. In the same vein, the clients can start processing the data received from the server, even while still receiving more of it.

4.4.4. Discussion

In this section we discuss two issues that sets our architecture apart by addressing them explicitly: different kinds of communication within the environment, and multiple levels of data caching.

Communication: We address three different kinds of communication: i) application to local DBMS-, ii) local tool to tool, and iii) client to client. We cover each in turn.

As we mentioned above, there are high performance applications that demand top speed for tool to DBMS- communication. Since issues of modularity and data integrity preclude a shared memory approach, we need a very fast, local inter process communication (IPC) mechanism and an efficient memory to memory block copy command. This support is provided by the operating system. In Chapter 5 we cover other techniques that can speed up tool to DBMS communication that include multiple append queries, and minimizing the impedance mismatch between the DBMS and the application.

A central repository architecture is elegant and conceptually simple, but we believe that it is a mistake to restrict tools to communicate solely through the central repository as in the *ADMS±* architecture. This is because using the repository as intermediary imposes a large performance penalty that is unacceptable in many cases, and also because much of the communication is of transient data that does not require dbms storage. Therefore, providing alternate, high performance and low overhead tool to tool and client to client communication mechanisms is important. For applications that require sharing data efficiently and with no DBMS functionality, we provide a shared memory area for the local toolkit. For example, a compiler can place a program at a certain location, and pass a pointer to that location to the profiler tool. In addition, we provide a comparable facility for *inter-user* communication of data that does not need DBMS services, and requires top performance (eg. members of a team monitoring, in their own workstation, their module during a test run of the software system). If, as in this example, most of this data is transient there is no ned to save it to the DBMS. Whichever of it needs to be preserved, can be done so by any of the tools involved. We propose a client to client (1 to n) message passing system to provide this high performance, low cost data sharing.

Levels of Caching: Our architecture provides several levels of data caching to improve performance: level 0 of the caching hierarchy is the DB+ on the server disk, it caches data from the WORM jukebox; level 1 is the buffer pool of the DBMS+ on the server, and it caches data from the level 0 disk; level 2 is the cached DB- on the client disk; level 3 is the buffer pool of the DBMS- on the client; and finally at level 4 is the cache maintained by the application and the toolkit cache.

The caching levels have different characteristics in terms of operations, capacity, and physical organization. For example, an AST stored in a node per tuple schema may use unique object id's as node links as part of the DBMS, but use memory addresses when cached by the application. Similarly, if we store the picture of the programmers working on a certain module, we may have it in compressed form on the disk, but uncompressed in the application cache. We need a mechanism as that proposed in [42], that explicitly supports a multi-level store by taking into account factors like i) different representations for the same object at different levels of the storage hierarchy, ii) different access methods for different levels, and iii) different transfer sizes between levels.

4.5. Summary

In this chapter we discussed past and current CASE environment architectures, highlighting their weak areas. We then propose a new architecture that borrows ideas from those discussed before, but also adds new features that address their weak areas. In addition, in designing the architecture of TULUM we use a comprehensive approach that combines physical design, logical design, and the computational paradigm used.

CHAPTER 5

DATABASE DESIGNS FOR SOFTWARE DOCUMENTS

5.1. Introduction

In order to produce the best database design for a given data structure and under a specific environment, we need to understand several factors: the relevant data structure characteristics, the important database design dimensions, and the major features of the storage and execution environment. In this chapter we look at each of these, which provide us with a framework to help us make good choices when designing the database schema for a CASE document.

We first look at the data structures while they are in-core. We develop five criteria that are important in a database context: internal structure, information content, quantitative aspects, qualitative aspects, and sharing aspects. Each of these criteria consists of one or more characteristics. The result is a framework that allows us to identify and understand the important attributes of a data structure in a database context.

We then move our focus to the database, and look at the main dimensions of a database design for a data structure: granularity, number of relations, and data representation. We explain what each is, why it is important, and how it affects the functionality, space utilization, and performance characteristics of the database design. We then examine how each of these dimensions is affected by the important environment variables in TULUM.

We move on to cover the characteristics of software engineering documents. With the aid of an object oriented model we classify CASE data structures into five classes according to the kind of data that they hold: textual, tabular, graphical, unstructured, and sequential binary.

In the last part of the chapter we use the in-core and database analysis frameworks that we developed to analyze and develop database designs for the CASE data structures that we use in our experiments: software source code, abstract syntax tree, and symbol table. We close by looking at the kinds of CASE operations on the selected data structures, and their performance requirements.

5.1.1. Contributions

Other researchers have analyzed in-core data structures from different points of view like productivity metrics, or complexity metrics. Others have analyzed them in regards to their sharing characteristics, like its representation. But nobody has proposed criteria that extracts the important characteristics of the data structure in regards to its storage and handling. Further, nobody has coupled the in-core analysis of the data structure with an examination of how this analysis influences the database schema design decisions.

We pull ideas from many different contexts, and develop five criteria that are of importance in an in-core data structure: internal structure, information content, quantitative aspects, qualitative aspects, and sharing aspects. We move on to distill the three main dimensions of a database design for a program data structure: granularity, number of relations, and data representation. We then use the in-core criteria to analyze three important data structure classes, and the database criteria to develop different database designs for each of the data structure classes. We provide tables that show what the advantages and disadvantages of each different design are, so that they can be used as a guide by CASE environment users.

5.2. Data Structures

In this section we look at different kinds of data structures, and explore their important characteristics. As examples we use important CASE data structures that include text documents like program source, tabular structures like symbol tables, graph structures like an AST, picture data structures like pictures of the software engineers, and sound data structures like a recording of the requirement specifications.

We are interested in five aspects of data structures: 1) whether the data structure has any internal structure that may be independently manipulated like an array entry, 2) the information content of the data structure (e.g., syntactic program structure in an AST), 3) quantitative parameters like its size and cost, 4) qualitative information like what are the *natural* aggregation and manipulation units like statement subtrees in the AST, and 5) the data sharing characteristics like how heterogeneous are the processes that may access this data. We cover each in turn.

1) Internal Structure: The first dimension that we look at is whether the data structure is simple or complex (ie, made up of sub-objects). A simple data structure, like a bitmap, does not have any internal substructures that are independently manipulated, but a complex data structure like an AST is made up of nodes that correspond to tokens[6] from the source code. For both kinds of structures it is important to know the cost to produce them, as well as whether they can be regenerated on demand if they are not saved. For complex data structures, we have to look at several parameters including the number of elements, the element size, is it homogeneous or heterogeneous (more than 1 kind of simple subobject), and how much actual data and how much overhead (e.g., reverse pointers) is in the structure.

2) Information Content: The kind and amount of information in a data structure determines how useful it is. There are two main ways of representing that information, explicitly or implicitly. We need to know what kind of information the data structure has, as well as which is explicit and implicit. In general, for a given kind and amount of information, the more explicit, or **deeper**, it is, the more expensive and complicated to produce the data structure. On the other hand, the deeper the representation the easier and cheaper it is for tools to extract information from it. That is, the choice of

[6] A token in a programming language is a group of contiguous characters in the source code that, according to the lexical rules of the language, belong together. Examples include keywords like *dotimes*, identifiers like *counter*, and operators like >=.

representation involves a tradeoff between its cost and its power. Because of the cost and complexity of producing deep representations, it makes sense to make explicit only as much information as the system needs.

For example, parse trees are relatively shallow representations that make the syntactic program structure explicit but omit the data and control-flow information. They are relatively easy and cheap to produce through scanning and parsing, and allow for good support of syntactic changes, but provide no support for algorithmic change, or semantic information.

On the other hand, to support more complex software tasks, the software representation must explicitly contain all the information required. For example, the Programmers Apprentice (PA) [21] uses the *plan calculus* to represent programs and their designs. The plan calculus makes data flow, control flow, and design information explicit, and thus allows the PA to directly manipulate these features. It can be used for very accurate cross-referencing (e.g.. when searching for where a variable is set, we can ignore those instances that are not in the relevant data path). If we used shallow representations, we would need a parse tree, a symbol table, and a data flow graph to perform the same operation.

In the DBMS context, the breadth of topics that the DBMS can support is limited by the amount of explicit information stored in its database and the power of the operations available to manipulate the data. For example, with an AST we explicitly store the syntactic structure of the program, which the program source contains in implicit form. But, if we provide parsing capabilities to the DBMS, we can use those capabilities on the program source to extract and answer queries about the syntactic structure.

3) Quantitative Characteristics: The parameters we can measure are important because they determine, among other things, how the data structure can be organized, its space requirements, and how much it costs to produce. Among the important parameters are:

- *Cost to Produce:* This parameter is important because it can be used to compare against the cost to store and fetch from the repository, like we do in our experiments.
- *Kinds of Atomic Elements:* The kinds of elements tell us if the data structure is homogeneous or heterogeneous, and how they are represented. In particular, we want to know how many different types of atomic elements there are, and what these types are.
- *Number of Elements:* Combined with the element size, the number of elements tells us how large the data structure is. The important parameter is whether the number of elements is related to another parameter (e.g., the number of nodes in an AST is a function of the number of lines in the source code), and what is the asymptotic behavior of that relationship (e.g., linear or quadratic).
- *Atomic Element Size:* Size has three components, net information content, data structure overhead, and information overhead (overhead refers to space devoted to purposes other than holding the net information). For example, if an AST node is a record that has parent and children pointer fields, the record instance header is data structure overhead, the parent pointer is information overhead, and the children pointers are net information.

4) Qualitative Characteristics: These are the data structure aspects that we cannot measure, but that are nevertheless important in determining how to store the data structure. Among the important parameters are:

• *Links or pointers:* Data structure links need to be preserved when they are stored in the database. All kinds of links are important, including those between elements, as well as those between different data structures. The links can be explicit (e.g., pointer fields), or implicit (e.g., attributes in a hash table that whose key is the symbol table entry with which they are associated). We might produce a "link forest", i.e. a graph that shows which element kinds are linked to which other element or data structures.
• *Natural Aggregation Units:* The units of aggregation reflect how the data structure is organized and how it is manipulated, since the operations on the data structure are directed to them. For example, the AST is made up of nodes at the atomic level, but has several natural units of aggregation like expression and statement subtrees. In many occasions, these units form an aggregation hierarchy like program, procedure, statement, expression, and node in the AST.
• *Natural Data Representation:* We are interested in how the data structure is represented when in memory. For example, source code is represented as a hash table of variable length strings by some editors, and our compiler frontend represents the AST as a series of records linked by pointers.
• *Persistence*: To evaluate if a data structure or an element is worth storing, we need to consider several factors: whether it is transient (temporary) or not, whether it is dependent on program state, the number of tools or applications that need it, its usefulness, its cost in space, its cost to store and fetch, the cost to regenerate when needed, and the cost to maintain it.

5) Sharing Characteristics: If we decide that a data structure will be shared between processes, we need to consider the medium we will use to share it, the way that the applications will interact with the data, and the differences in the sharing environments.

• *Sharing Medium:* There are several media that allow data to be shared: central repository, shared memory, or direct data copying. The factors that determine which medium to use include its availability (e.g., if shared memory is not available it is not an option), the performance requirements (e.g., if we require millisecond response times shared memory is very attractive), the physical placement of the applications (e.g., if the applications can be located in separate machines we cannot use shared memory), and the functionality requirements of the data (e.g., if concurrency control is needed data copying will not do).
- Shared Memory: Provides the highest level of performance since it involves no data copying or any other I/O, and the tools simply pass a pointer to the relevant data. The data is modified in place, so changes are immediately visible. The applications need to provide some form of locking if multi user access is desired. But this approach is limited to applications that reside in the same computer, with an operating system that supports shared memory (not standard), and when no DBMS functionality is required. In addition, there is no data security in the shared memory segment since it can be corrupted by any of the applications that access it.
- Direct Data Copying (messages): The processes can be in arbitrary locations. It involves copying and transmitting the data. Applications need to maintain the data synchronized. Performance is limited by the number of data copy operations and by the latency and throughput of the communication medium. Perfect for client to client communication of data that does not require DBMS services.
- Central Repository: This involves the use of an intermediary process, the DBMS. This

is slow, but provides very sophisticated functionality. The DBMS provides synchronization and other services like data conversion if needed.

• *Application Interaction Mode:* The way that applications interact with the data may affect its storage design. There are two main ways, get all in memory at once, or use incremental I/O. In addition, it is important if the interaction is update or read-only.
- Update or Read-Only: In read-only mode no synchronization is needed. On the other hand, if applications update the data we need a synchronization mechanism to prevent any lost updates or data corruption.
- Incremental I/O: Involves only bringing in data that is needed at any one time. Optimizes memory use and minimizes the number of I/O's. On the other hand, cache residency checks required on every access, and if all data is eventually needed it uses too many disk seeks since data is not read sequentially.
- All In Memory: Send and copy all the data at one time. Very simple, data is transmitted at a fast rate (minimizes total number of seeks). Once data is received, fastest access to individual data items. But, may be wasteful of memory, can produce thrashing if data too large for memory, and involves very long startup times.

• *Environment Heterogeneity:* The execution environment of different applications may vary in many respects, in some cases requiring that data be converted when moving between environments. Factors that are important include the machine architectures and programming languages. Both of these factors may require low level conversion of the data.

In the next section we look at the data structure dimensions that are important when the data is stored in a DBMS.

5.3. Database Design Dimensions

For a given data structure, we have identified three important dimensions in which a database schema can vary: the number of relations (e.g., store the SYMTAB in one relation as a symbol entry and its attributes in one tuple, or in two relations with the symbol table entry in one relations and its attributes in another one), the granularity of the unit of storage (e.g., store the source code as a procedure per tuple[7], or as a text line per tuple), and the data representation (e.g., store the AST nodes in an ASCII or a binary format). These dimensions have effects on space utilization, the kinds of operations supported, and the speed of the operations. We explain each one in turn.

5.3.1. Number of Relations

The number of relations has an impact on space utilization and operation performance. The impact on space starts with the entries that each relation has in the system catalogs. In addition, depending on how the relations are organized, the number and size of the tuples in the relations can vary. The number of tuples has two kinds of overhead: the per tuple book-keeping overhead, and the access method tuple overhead. For example, in our experimental implementation, the SYMTAB consists of two distinct

The relational model of data is based on relations or tables. A table consists of rows and columns. We use the relational terminology that calls a column a *field*, and a row a *tuple* entities, a symbol table entry and its associated attributes. Among the different SYMTAB schemas that we look at are SYMTAB entry per tuple (SPT) in which we have one relation with a single entry for each SYMTAB entry and its associated attributes, and attribute per tuple (APT) in which we have 2 relations, one for the SYMTAB entries and the other with an entry for each attribute. For a given SYMTAB, the SPT design has an advantage since it has fewer tuples than the APT design (e.g., an entry with 4 attributes has 1 tuple in SPT but 5 tuples in APT), but APT has an advantage in that it has no empty fields (e.g., an entry with 1 attribute leaves many fields empty in the SPT design but none in APT). Another consideration is that the attribute relation in APT has an extra field that points to its corresponding SYMTAB entry. Figure 5.1 shows the SYMTAB entry and some attributes for the statement

int i;

The two relation schema, APT, has more tuples and fields, leading to worse space utilization. In this example we have a 4:1 tuple ratio, and (counting all fields as 4 bytes, but no tuple headers) a 3:1 space ratio.

In addition, the number of relations per schema has an impact on performance, since the queries to perform a given operation are different. For example, to obtain a set of SYMTAB entries and its attributes is a single relation query with qualification for

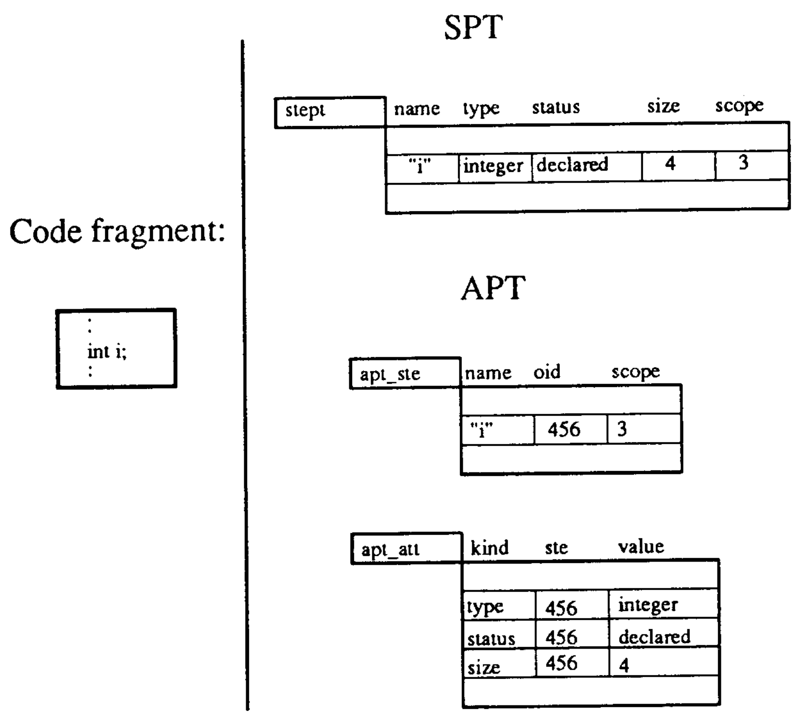


Figure 5.1: Entry for the same statement with SPT and APT designs.

SPT, but a join or a union query for APT. The impact on performance depends on how the queries are formulated, what access methods are available, and the specific optimizer used. As an example, if we want to know the status of variable *i* when we encounter the statement *i = 4 * y;*, with the SPT schema the query[8] is:

```
retrieve (s.status) from s in stept
where s.scope = 3 and s.name = "i"
```

while with the APT schema it is:

```
retrieve (a.value) from a in attpt_atts, s in attpt_symtab
where a.ste = s.oid and s.name = "i" and s.scope = 3 and a.kind = "status"
```

The SPT query is single relation, with two constant clauses. The APT query is a two way join with three constant clauses, more complex and usually more expensive.

5.3.2. Granularity

Granularity, or the size of the unit of storage, has a pronounced impact on the space and time efficiency as well as on the kind of support from the DBMS. A CASE environment needs to manipulate a range of object granularities, from coarse objects like modules and projects to fine grained objects like SYMTAB entries and AST nodes. The most obvious impact on space is that the number of tuples diminishes as granularity increases, leading to lower bookkeeping and access method overhead. For example, in one of our AST experiments, the node per tuple (NPT) representation has 105,845 tuples, whereas the procedure per tuple (PPT) representation has 1923 tuples. The 55 to 1 tuple ratio is the main contributor to a 5 to 1 space utilization ratio. In addition, the change of granularity leads to further space savings from the elimination of redundant data. That is, we need to include some extra fields with every tuple that allow us to group the tuple with others like it, like the *file_name* field to identify nodes and procedures. These extra fields accrue on a per tuple basis.

The effect of granularity on performance goes hand in hand with that on space, since it comes from the number of tuples and the database size. As granularity increases query time decreases since both the database size and the number of tuples decrease. On the other hand, the FE time overhead might increase since the data has to be "packed" together and might also include some type conversion. In the face of parallelism, the performance advantages of coarse granularity designs diminish, both with I/O and processing parallelism. Processing parallelism comes from multi-processor systems, while I/O parallelism can come from Redundant Arrays of Inexpensive Disks (RAIDs), where the data is distributed in many small disks that can be accessed in parallel [82].

Fine granularity objects are numerous and highly interconnected, and are manipulated by tools that require very high performance. They must be moved between the tools and the DBMS with little overhead. This requires tight binding between the database data representation and the tool's programming language, since loose bindings require too much data conversion overhead and can be an order of magnitude too slow.

[8]We use the POSTQUEL [81] query language to express our queries throughout this thesis.

In addition, fine granularity objects maximize DBMS functionality.

Large granularity designs allow for higher performance by grouping many small objects together. Both the space and time overhead per object is lowered by this grouping. The greater space and time efficiency of larger granularities can allow us to save more information than with smaller granularities (e.g., save the parent and children with the tree nodes). This would cost time during storage, but would save time when we fetch the data from the DB. On the other hand, a large granularity design will make the *per record* operation cost higher. This is a disadvantage in environments where many fine grained, low locality updates are common. Finally, larger granularities diminish the functionality that the DBMS can support, since the DBMS does not know about their internal structure. For example, with a PPT design the DBMS will not support queries on the number of *if* statements in a procedure, whereas it will with NPT. This problem is so important, that we cover it in detail in the next section.

5.3.2.1. Intra-Object Access

Given the high space and time cost of fine granularity designs, large granularity designs look very attractive. The main disadvantage of increasing granularity is that the DBMS can no longer directly manipulate small program units. DBMS's do not directly manipulate the internal structure of objects for reasons of efficiency and complexity. The efficiency consideration has to do with the fact that operations on the internal structure of objects are expensive and proportional to object size or structure. Therefore,

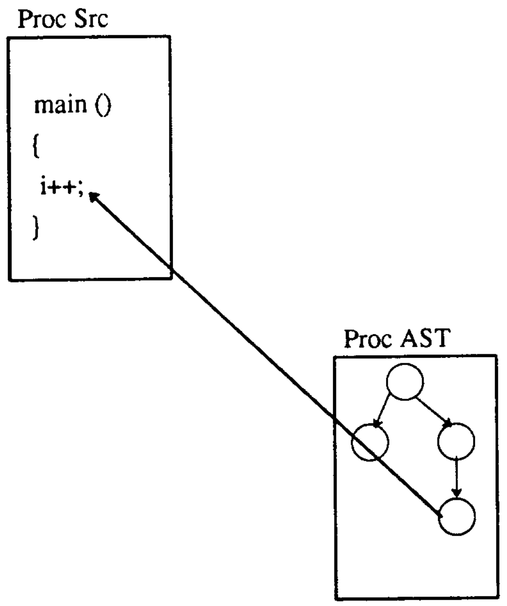


Figure 5.2: Intra-object reference between AST and source at procedure granularity.

queries that use them are expensive and it is difficult to estimate their cost when optimizing queries. The complexity consideration comes from the fact that there are so many different object types, each with a set of operations, that it is impossible to support them all.

We call this loss of DBMS functionality the *intra-object access* problem, since it involves access to the internal structure of objects. Not only would we like to be able to query the small objects, but also to be able to refer to them from other relations and objects. For example, with a procedure in a single tuple the DBMS cannot directly reference or query its variables or parameters. Figure 5.2 shows the issue with PPT granularity. Notice how a node inside ProcAST has a reference to the identifier "i" in the ProcSrc tuple corresponding to the source code of the AST tuple. We propose three ways to provide sub-object access: location based, with embedded-oid's, and via the ADT mechanism. To illustrate the three different approaches we use an example in which we have source code stored in the *proc* relation in PPT granularity, and where we ask a query about variables used in a procedure. We also show, in each case, any auxiliary relations needed to support the query.

Location Based: Location based reference provides efficient access, with the ability to provide indices. For example, in [16], a tree node uses line# and column# to refer to an object inside the source code procedure. In Figure 5.3 we show the *proc*, *stms*, and *var_used* relations needed to support queries on variables used inside the procedures. As can be seen in the figure, we use the procedure name to specify the procedure object,

proc	source	line	name

	main () {
	i++;
	}
		1	main

stmt	proc	line	col	kind
	main	2	1	arithmetic

var_used	name	line	col	proc
	i	2	1	main

Figure 5.3: Location based solution to intra-object reference problem.

and line# and column# to index inside it. The DBMS directly supports indices on proc.name, and the *var_used* and *stmt* relations support locating the internal variables. To ask about variables used in the procedure main, we formulate the query

```
retrieve v.name from v in var_used, s in stmt
where s.proc = main and v.line = s.line and v.col = s.col
```

This design is not very general (e.g., it would not apply for text lines referring to a node), and requires potentially many updates after text formatting operations (e.g., if we add a new line to a procedure, on the average we need to adjust the line number of half the entries in the *stmt* and *var_used* relations). In addition, it does not provide strong sub-object identity (Identity is that property of an object that distinguishes it from all others. Strong identity identifies an object regardless of its value or its location [83]).

Embedded OID: A more general design assigns a private object id to each sub-object, and embeds this object id with the sub-object when its containing object is saved to the DB. For example, in Figure 5.4 the statement "i++;" has been assigned the embedded id 088, which is automatically included in the source code in a non-destructive fashion. This approach also allows queries to be performed on the embedded objects by maintaining auxiliary relations. For example, we can use the relations in Figure 5.4 to ask the same query (all variables used in procedure *main*)

proc	source	line	name

	main () {
	i++; /* oid(088) */
	}
		1	main

var-use	v-id	stmt-id
	890	088

var	name	id
	i	890

stmt	proc	stmt-id
	main	088

Figure 5.4: Embedded Id solution to intra-object reference problem.

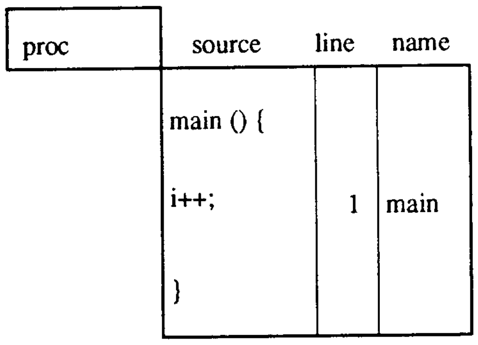
retrieve (v.name) from u in var_use, s in stmt, v in var
where s.proc = "main" and s.stmt_id = u.stmt_id and u.v_id = v.id

The embedded id solution provides strong sub-object identity, and also allows for building indices, providing for high performance access to the sub-objects. The disadvantages of this approach include the added space of the extra relations (e.g., var-use), indices if we create them, and in the data itself; and the performance penalty and complexity of having to pre-process the data both when storing it (to create the embedded id's, insert them in the data, and insert the entries in the extra relations and indices) and retrieving it (to return it with no modifications to the application).

ADT Support: A third way to support intra-object access is by using the abstract data type mechanism by defining a new type with new operators to perform the access that we want. This is analogous to implementing DB procedures to extract implicit information from a data structure, as well as the object oriented principle of encapsulating operations with objects, as done in Allegro [15]. For example, in Figure 5.5 we define the ADT proc.source with the new operators *params, nth_stmt, and vars*, and can ask about all variables used in the procedure main with the query

retrieve (vars ("proc", "main"))

The ADT approach is attractive in that it does not require any modification to the source and requires no auxiliary relations, therefore it wastes no space. It is the most general and powerful solution. On the other hand, it does not provide sub-object



proc.source is an ADT with operators:

params
nth-stmt
vars
....

Figure 5.5: ADT solution to intra-object reference problem.

identity or the ability to build indices, sub-object access is slow since it involves expensive examination of all objects, and it requires the ADT designer to provide the code for every new operator. As shown in equation 5.1, the cost of processing a single ADT clause query would be

$$f(n)\times C_i(S)+w\times f(p) \tag{5.1}$$

Where:

n is the number of objects considered,
C_i is the CPU cost function for ADT operation i,
w is the I/O to CPU weight factor,
p is the number of disk pages processed.

We like the ADT solution because of its power and generality. The main problem is with its performance, since a clause with an ADT operation cannot be used to restrict the number of tuples examined. For applications that require top performance we think that the embedded-oid solution is superior to the location based approach because of its generality, since it provides strong sub-object identity, and does not require maintenance updates after formatting operations.

5.3.3. Data Representation

Data representation is the way that data is physically stored. For example, in the FE the AST has a binary representation, but when we store the AST in the BE it can be in an ASCII or independent binary format. The data representation choice hinges on various factors: speed and space constraints, extent of DBMS and application support needed, degree of language and machine independence desired, how we will divide the conversion work between the BE and FE, possible loss of accuracy in the conversions, the frequency of different kinds of operations on the data structure, and the evolutionary stage of the CASE environment. We cover each of these factors in turn.

- *Space:* The space utilization of different representations can vary significantly. For example, ASCII uses a byte per printing character so a 3 digit integer uses 3 bytes while a 10 digit integer uses 10 bytes. In contrast they would both use 4 bytes in a 32 bit binary representation.
- *Speed:* Data representation may affect speed during data transmission and in converting it from one format to another. Data transmission speed is proportional to the number of bytes, therefore a larger representation increases its cost. And the conversion overhead, also proportional to the data size, can be very substantial when the two representations are very different, depending on the specific algorithm used.
- *Support Required:* In choosing a representation, it is important to consider how much support for it exists in the DBMS. The easiest case is when the DBMS already supports a representation, otherwise we need to extend the DBMS with user supplied routines to store and manipulate the chosen representation. For example, if we are dealing with the AST at the PPT granularity, there might already be database procedures that can access the ASCII representation, and we would need to write new ones to access the procedures in binary representation.
- *Division of Labor:* We can have conversions done either in the backend or in the application. If the conversion is necessary for query processing, it has to be performed at the backend. On the other hand, if we store data in a compressed format and are fetching the data, it is best to compress and decompress[9] at the FE to unload the server

and minimize the amount of data transmitted.
• *Language Independence:* If the applications that will manipulate the data are written in different programming languages, our representation has to be language independent.
• *Architecture Independence:* If the applications that will manipulate the data will run on a heterogeneous computing environment, the representation must be architecture independent.
• *Loss of Precision:* When data is converted between different formats, we might have a loss of precision with conversion back and forth. The data representation and the conversion algorithms must be chosen to minimize the space and time cost with acceptable levels of data precision.
• *Operations On Data:* We must take into account the operations that will manipulate the data, which are more frequent, which are more critical, and which have more stringent requirements. For example, whereas ASCII might be best for ad-hoc queries, dependent binary might be best for performance intensive operations. Factors like which operations are more frequent or more critical need to be considered.
• *Application Evolution:* As pointed out in [20], during the development phase of a system, the internal data structures and data manipulation routines undergo constant change. while performance is not as critical. Therefore, a representation that is easy to implement and manipulate is more appropriate at the early stages of SDE development, while a representation that has higher execution time efficiency may be better when the system is released.

In [20], the authors classify the data representation choices as ASCII, dependent binary, and independent binary. We add a fourth choice, compressed, and we now cover each in turn.

- **ASCII:** The ASCII representation is both machine and tool language independent. Because it is easy to implement and almost universally supported, it provides the greatest functionality in the DBMS for medium and large granularities. In fine granularity databases an ASCII representation may be inappropriate since we store primitive types (e.g., integers and floats) whose ASCII representation buys nothing and costs plenty. Its disadvantages are increased time and space costs. It costs time since an ASCII representation usually requires extensive BE/FE type conversions, since most FE daa is not ASCII. It costs space because it requires 1 byte per printing character, which means that the size of an ASCII representation is completely dependent on things like the magnitude of an integer, the way we print a floating point number, etc. Depending on the content of the data, an ASCII representation can be smaller (e.g., a list of digits would use 1 byte per digit in ASCII, 2 bytes per digit using *shorts* in C), equal to, or larger than its binary equivalent. However, it is almost always larger.
- **Dependent Binary:** The dependent binary representation is neither machine nor tool language independent, since it is a core dump of the internal FE data representation with pointers virtualized and some omitted attributes. This representation is impedance matched with the FE, and therefore has little conversion overhead. In

[9] From now on, when we mean *compress and decompress* we use the term [de]compress for brevity.

addition, it is generally more space efficient than ASCII. Its main disadvantages are its strong coupling to the tool language and the machine architecture.

- **Independent Binary:** The independent binary representation is both machine and language independent, but more strongly coupled with internal FE representations than ASCII. It encodes names in 1 byte, integers in 4 bytes, and booleans in 1 bit. It uses slightly more conversion time and size than dependent binary, but less than ASCII. An example independent binary representation is the XDR (eXternal Data Representation) standard [84] that is a method of describing data in a language and architecture independent form. For example, there is a standard way of encoding integers, 32 and 64 bit floating point numbers, as well as fixed and variable length arrays.
- **Compressed:** Each of the data representations may be compressed. The compressed format is special since it can be applied to any of the other representations. Data compression has two major advantages: it saves space, and it also saves time during I/O operations since more compact data structures require fewer disk seeks and network messages. The disadvantages include the extra complexity from implementing the compression algorithms, the CPU cost of [de]compression, and the potential loss of functionality from the loss of internal structure in the compressed data. [De]compression can be applied to single objects, or sets of objects. When there are many small objects, [de]compression is most effective when applied to large sets of objects. In an environment of large objects [de]compression might be effective on single objects or small sets.

  In a client server environment, there are two places where [de]compression can take place: i) At the client, in the FE module that sends and receives data from the BE, and ii) At the server, in the DBMS module that fetches and stores data from the database disk. Applying the [de]compression algorithms at the client has the dual advantage of offloading the server, and also of minimizing the amount of data transmitted between the FE and the BE. The major reason to have [de]compression done at the server is to have DBMS functionality on the compressed data, for example to allow querying of small grained objects or to apply ADT operators to large grained objects.

The final concept that we discuss in this section is that of *impedance match* of the data or *data coupling* The impedance between data representations refers to how close the data representations in the FE and BE are. When the representations are far apart, we say there is an impedance mismatch between them, or that they have weak data coupling. When the representations are very similar, we say that they are impedance matched or that they have strong data coupling. In [20] the authors compare how data coupling parameters affect ease of environment evolution[10] and operation performance. They note that the stronger the coupling between the data and the application, the better the performance and also the slower and more complex the evolution, and the weaker the data coupling the slower the performance and easier the evolution. The performance effect is due to the conversions that the data must undergo when moving between environments. The weaker the coupling, the more extensive the conversions. And, as our experiments show in later chapters, data conversion can be **very** expensive, in some cases over 50% of the total data fetch or storage cost.

5.3.4. The Effect of Environment Variables

In this section we examine how changing each one of the database representation dimensions we study: number of relations, data representation, and granularity, is affected by some of the important characteristics of our proposed CASE environment. To make our explanations clear, in the following discussion we assume that the application is manipulating a single representation of a single data structure. We explore several areas: bulk updates and lazy evaluation, having many versions of documents, the use of incremental algorithms, the use of access methods, intra-objec references, multiple representations of data, and multi-level storage. We cover each in turn.

• **Bulk updates and lazy evaluation**: Key to our computational paradigm is the notion of keeping local updates confined to the local cache until one of the following happens:

i) users explicitly propagate them, or
ii) a certain number of updates is reached or,
iii) a certain amount of data is affected, or
iv) a certain amount of time has elapsed, or
v) some of the modified data is requested by another application.

This policy has several advantages that include minimizing the number of messages between clients and servers, minimizing the system overhead per item updated, providing fast interactive response, and giving the users control of when to propagate. In addition, if a data item is repeatedly updated between propagations, lazy evaluation only propagates the final value, minimizing the number of updates sent to the server. The disadvantages include the possibility of lost updates if either the client or the server goes down, as well as extra time when another client requires using updated but not yet propagated data.

Varying the number of relations does not affect this policy, The effect of different data representations will be mostly on the overhead of converting the FE data structure into its BE representation and viceversa. If the conversion operation is expensive, an application requesting use of this data may have to wait a long time for the bulk conversions to take place (since two redundant conversions are required). Granularity can have significant effects on this policy, depending on how the data is updated and what the propagation parameters are. For example, if we have large granularity and the *amount of data* propagation parameter is set low, we could have propagation after every update, leading to substantial inefficiencies. Under a *number of updates* propagation policy we would propagate much less data simultaneously at fine granularities, more at coarse granularities.

In addition, the lazy evaluation and bulk update policy applies in two different places: at the application to DBMS- interface, and at the DBMS- to DBMS+ interface. This allows for finer control of the system, for example by allowing an application to request frequent updates without having to pay the price of updating the server every time.

---

[10] The term evolution is used to denote the changes that a CASE environment undergoes while it is being developed.

• **Many versions**: Throughout the life of a software system a given document is modified on many separate occasions, first during development, and later on during testing and maintenance. After a series of changes that logically accomplish one or more high level actions, a new version of the document is created. Since on most occasions the new version is not very different from the previous version, systems that support versions store 1 full copy of the file (either the first or the last version) and a *differential file*, ie. a file that contains the differences between the stored document and its different versions. This is a classical space versus time tradeoff, in which we trade space by storing the differences between versions for performance when requesting a random version. For example, as part of our experimental environment, we have a LISP file called prog-to-ast.cl that contains high level code to run the program representation experiments, record their results, and generate reports on them. By version 1.9, and using RCS for version support [66] the differential file has size of 117 Kbytes, while version 1.9 has size of 40 Kbytes. If each version was about 30 Kbytes, the differential file saves over 50% of the space.

The effect of data representation in an environment with many versions is primarily one of space. A representation that is more compact will use less space in its differential file which in turn might lead to faster recreation of a given version (since less space translates into fewer I/Os). The effect of number of relations is mainly in efficiency, since recreating a version with fewer relations would probably be cheaper than recreating one with more relations both in the number of I/O's and the CPU time. The effect of granularity depends on how the repository supports versions. Assuming that versions are stored in a differential fashion, the important factor is whether the differential granule is at the relation, tuple, tuple field level, or byte level[11] The larger the differential granularity, the more expensive the space cost of the larger granularities.

• **Incremental algorithms**: In the context of a data repository, an incremental algorithm applies to how data structures are updated in response to changes in the environment. For example, when a statement in a procedure is changed to eliminate a bug, an incremental algorithm on the abstract syntax tree would localize the AST subtrees affected, and only changes those as opposed to discarding the existing tree and producing a new one as in PAN [11]. In an ideal situation, incremental algorithms will require less computation to update a structure, and will also require less data to be transmitted and handled. As detailed in [80, 27] the use of incremental algorithms is crucial to achieve the performance levels required by modern CASE environments.

Neither the number of relations nor the data representation have an impact on the use of incremental algorithms. Granularity impacts the performance advantage of incremental algorithms, mainly through the amount of data transmitted and handled. For example, assume there is a 3 node addition in an AST statement. With node per tuple granularity, we only send 3 nodes to the repository, but with procedure per tuple granularity we

[11] To illustrate the difference, suppose we create a new version of a relation in which only 1 byte of 1 tuple is different from the previous version. At relation granularity, the differential stored is the whole relation, at the tuple granularity, the differential is the complete tuple, at the field level the differential is the field whose byte is different, and at the byte level the differential is the byte that has changed.

need to update the whole procedure.

• **Access methods**; Access methods are the data structures and algorithms that allow for fast retrieval of data. In essence an access method is an ordered relation computed by applying a function on a set of values. The values are fields in relation instances.

Data representation impacts indexing only in that the ordering function has to be modified to manipulate whichever data representation is chosen. The number of relations impacts the number of indices, essentially multiplying them by the number of relations. The actual performance and space impact of different numbers of relations comes from the total number of tuples in each alternate design. In general, the more tuples there are the greater the space usage and the slower the access via indices. Also, we know of no system that supports multi-relation indices.

Granularity affects indexing in several ways. First, as granularity increases we get fewer tuples, indexing provides less of a speedup over sequential searches, and the index consumes less space. Second, since indexing only provides an ordering at the tuple level, as granularity increases there is no direct repository support for accessing, and ordering, the subobjects. As far as subobject access is concerned, we shift the computation burden toward runtime. Third, whether subobject access is done at the repository via database procedures, or by the tools in the FE, we increase the complexity of either the BE, the FE, or both.

• **Intra-object references**: In a modern CASE environment, objects within a software document have links to other objects in the same document and to objects in other documents. The references or relations between objects can be of several types [85] but in all cases require two things: some kind of object identity, and some way to access an object given its identifier.

Data representation and number of relations do not affect intra-object references. But, granularity has an important impact on them. With fine granularity, the repository supports links directly. But with coarse granularity representations, the repository no longer supports links to subobjects. See Section 5.3.1 for our proposal for support of sub-object references.

• **Multiple representations**: In order to provide sophisticated services, a CASE repository must explicitly provide a wide array of information about a given software document (eg., the AST provides syntactic and the SYMTAB semantic information about the source code document). Some researchers advocate deep representations that make explicit all the information of interest that they contain, like the lambda calculus [21]. But in most cases, we derive different representations to make explicit different kinds of information (eg., from the source code we derive a data flow graph and a control flow graph for different purposes). When we have multiple representations of the same document, we need to have a mechanism to ensure that they are all up to date when accessed.

The number of relations has a slight impact in the fact that to maintain the multiple representations synchronized we need to manipulate more relations, which might be more inefficient. The data representation impacts the routines that have to maintain the representations synchronized in that they need to understand the different representations. Granularity impacts multiple representations in the issue of intra-object references as mentioned above. Both granularity and data representation have an impact on space, and this becomes more significant when we have many representations of a single document.

• **Multi-level storage**: Current generation data managers deal with two storage levels, magnetic disk and main memory buffer pool. In order to accommodate the increasing number of applications with mammoth data processing requirements, next generation DBMSs are being extended to explicitly account for and manage data at multiple storage levels. An important aspect of multi-level storage is that data is optimally managed in different ways at each level, and an optimal format in one level is not necessarily optimal at other levels. Therefore data representation is an integral part of storage at multiple levels. In our discussion assume that we use 12" WORM technology for tertiary storage, in the case of moving data between the disk based database and the WORM platter, and that the data conversion between these levels is compression and decompression.

The number of relations do not have an impact on multi-level storage. Data representation has a strong impact as we mentioned above. The application choice of data representation in the database impacts the space utilization, the amount of processing to transform the data when moving through the storage hierarchy, and the *division of labor* of the transformation process. For example, if the application stores pictures in compressed format at the disk level, [de]compression is performed at the application-DBMS interface, whereas if it is stored uncompressed then [de]compression is done at the WORM-disk or WORM-memory interfaces.

Granularity impacts multi-level storage in that the objects at different granularities require different conversions when moving between levels. For example, if we look at two ways of storing the AST, at the NPT and MPT granularities. It makes no sense to handle nodes individually in this setting, since the processing and I/O overhead would be too large, therefore the [de]compress algorithms for nodes would work at the relation level. On the other hand, an AST for a module would be so large, that it can efficiently be processed individually.

5.4. Software Engineering Documents

As we mentioned before, the end products of all software lifecycle phases are different documents. The documents range from requirement specifications, through program source, to the users guide. A CASE environment is geared to efficiently create, store, and manipulate these documents. In this section we first introduce a methodology to analyze the requirements of the documents, and then we focus on the software code document.

We have developed an object oriented (OO) methodology for the systematic study of software engineering documents. The methodology is based on two observations: i) the end products of the software lifecycle are all documents, and ii) documents have many characteristics in common. Example documents are the source code, requirement specifications, and user documentation. A common characteristic is that all documents are written in a language (e.g., C for source code and Spanish for the user documentation).

The methodology consists of creating a hierarchy of different kinds of abstractions (e.g., aggregation, generalization [86]) that are common to software systems, rooted in the software system (SS) class at the top level, and concentrating on those aspects of the system that we want to analyze. We now show the model we developed to analyze the database schemas of SS documents.

The model we use to analyze the requirements and organize the database schemas (operations and data manager requirements) for software documents has five layers: *software system, document, notation, representation, and schema.* The hierarchy reflects the fact that a SS is composed of different documents, a given document can be written in many different notations (e.g., a program can be written in C or Pascal), a given notation can have several representations (e.g., a C program has source code and a symbol table), and that each representation can have many different database schemas (e.g., source code can be stored as unstructured text or as comments + data structures + procedures). The model is illustrated in Figure 5.6, where we show a subset of the class hierarchy. The names of the leaf classes reflect the granularity of the units stored in the instances of the class (i.e., LPT means line per tuple, SPT means symbol table entry per tuple). The characteristics of the model include:

- A natural, easy to understand description of a software system.

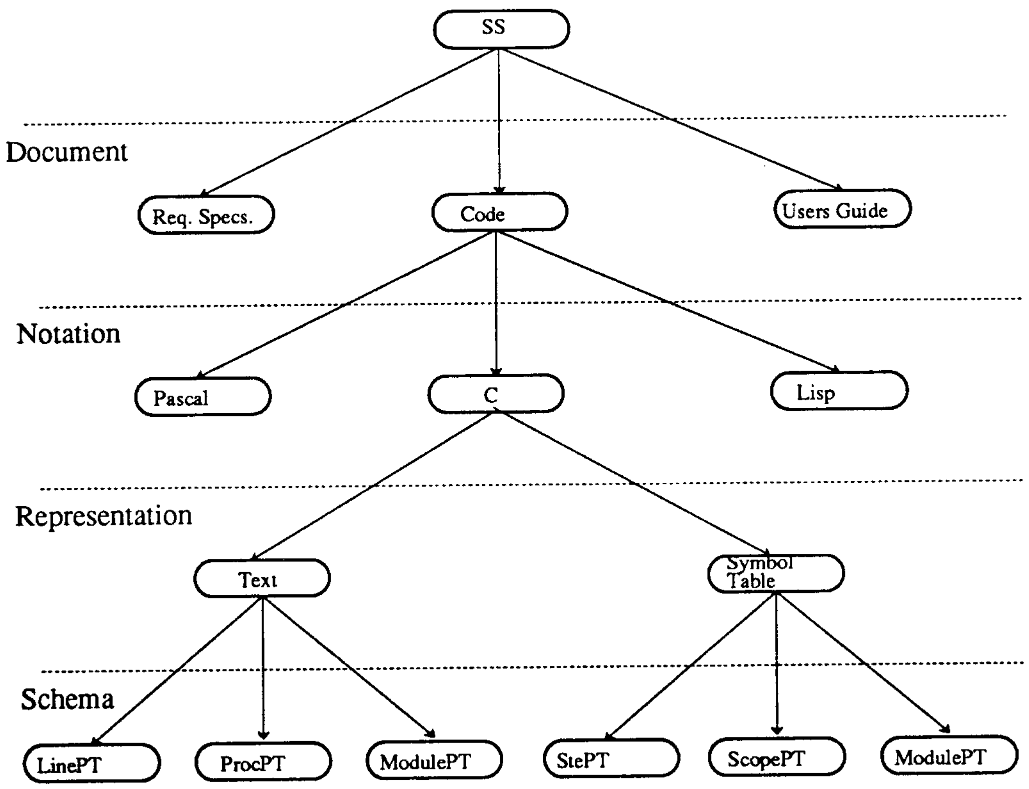


Figure 5.6: Software System model for database schemas.

- It is process independent: It does not change irrespective of how the documents are produced.
- Adaptable to any lifecycle model: Simply adjust the documents to reflect those of your favorite lifecycle model.
- Extensible: Can add, subtract, or modify subtrees as well as levels of abstraction.

If we look at the software engineering process as a transformation of documents from one form to another (e.g., requirements specifications to functional specifications to architectural specifications), we have made two interesting observations with the model. First, the transformations wanted are those between siblings at different levels of the hierarchy and in all subtrees. Second, the higher the abstraction level, the more difficult the transformation. For example, a desired transformation at the document level would be from the requirement specification document to the program code document. At the representation level, a desired transformation is from program text to symbol table, which can be easily accomplished by a compiler frontend. On the other hand the transformation from requirement specifications to program code is the as yet unattained dream of automatic programming systems [69].

By including all the different forms of source code in the model, we have used it to classify the different representations of software according to the structure of the data that they contain:

i) Textual data like program source.
ii) Tabular data like a symbol table.
iii) Graphical data like abstract syntax trees.
iv) Unstructured data like pictures.
v) Binary sequential data like object code.

Unstructured data is, more than a form of software code, a type of data structure that software code manipulates. The binary sequential data ia an ordered sequence of fine grained objects, 32 bit instructions in the case of object code for the SPARC architecture, whose internal structure and information content is not statically available.

5.5. System Code

Both in the examples we have been using, as well as on the experiments we run in later chapters, we focus on the *program code* Software System document. The FE takes as input C source code files, and produces a parse tree and a symbol table for each file. Figure 5.7 shows a simplified view of the different phases of a typical compiler, and the program code representations that each takes as input and produces as output. The source program is typically written in a high level language, like C, and the target program is often object code. The reasons for focusing on the code document include that we can use different program code representations to illustrate the database designs for the different data kinds: the program source as textual data, the symbol table as tabular data, the abstract syntax tree as graphical data, and any of them as large unstructured data. In addition, there have been several previous attempts at program code storage in DBMS's [14, 15, 16], and they have failed to satisfy many of the CASE requirements like performance and data modeling. Further, the algorithms to transform code between its different representations are fast and well understood. The different code representations have obvious and important uses. Since software is plentiful in the EECS department, we have access to, and use, realistic, large scale SS code like that of BSD Unix and POSTGRES.

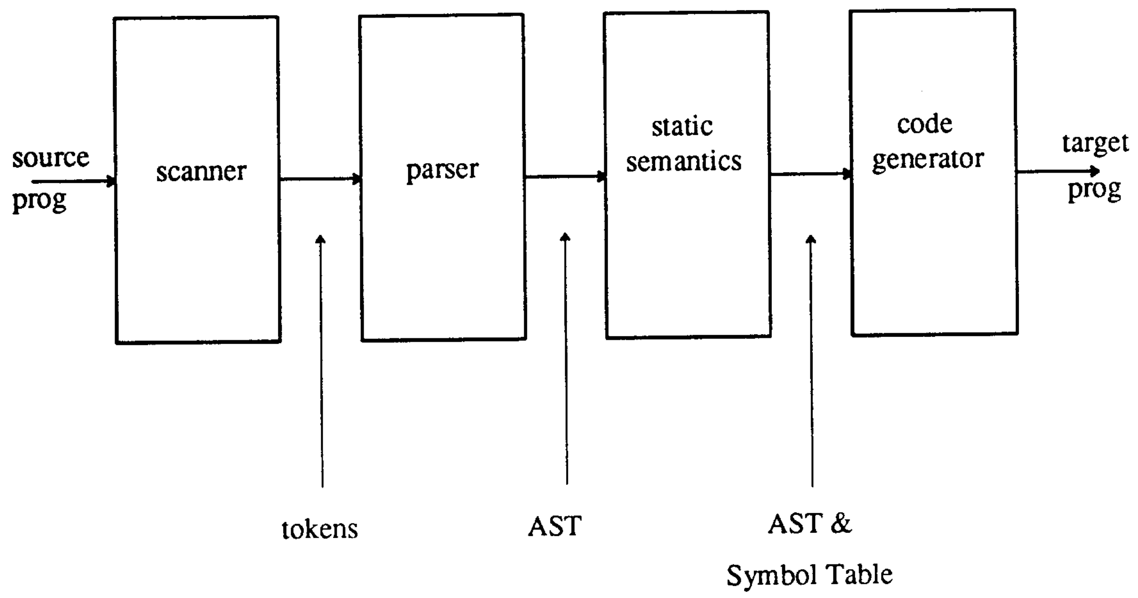


Figure 5.7: Program code representations produced by a typical compiler.

We chose the C programming language to illustrate our ideas, as well as the language in our sample implementation. The choice was dictated by the abundance of projects that use it here at UC Berkeley, and because it is simple and small. The specific language does not impact our algorithms or implementation significantly. In this section we analyze the representations using the data structure parameters that we presented and developed in Section 5.2. In making the analysis, we use the quantitative figures that we derived in our experimental environment, especially the equations in Section 6.5 on data structure generation, and we assume a heterogeneous hardware environment and a homogeneous language environment.

5.5.1. Text

Program source code, which we use as representative of the textual data structures, is usually written in a high level programming language like Pascal or C. It is perhaps the most manipulated form of the SS, and all the other code representations are derived from it. Most of the program source code is manually written by software engineers, with some syntactic and static semantic support from the text editors. Some of it may be generated from fourth generation language compilers like Yacc or Ingres Query By Forms. The tools that operate on text include text and syntax directed editors, formatters, and debuggers. Figure 5.8 shows the analysis of the software source code document using the criteria that we developed in Section 5.2.

(1) Internal Structure: Complex.

(2) Information Content:
- Syntactic Structure (implicit)
- Static Semantics (implicit)
- Algorithmic (implicit)
- Control Flow (implicit)
- Data Flow (implicit)
- Other kinds, in comments (implicit)

(3) Quantitative Information:
- Cost to Produce: Organization and project dependent. Measured as lines of code per programmer per day [87].
  Observed rates range from 20 to 1500 lines of code per person-month of effort.
- Kinds of Atomic Elements; Tokens and lines.
- Number of Elements: 1195.5 lines per file
  Projects from 50 to 10,000 Kilo lines of code.
- Atomic Element Size: 25.1 bytes per line

(4) Qualitative Characteristics:
- Natural Aggregation Units: statements, procedures.
- Natural Data Representation: ASCII, array of variable length strings.
- Links: None internal; none external.
- Persistence: Yes, all of it.

(5) Sharing Characteristics:
- Sharing Medium: Central repository.
- Application Interaction Mode: All in memory, update.
- Environment Heterogeneity: Different architectures.

Figure 5.8: Analysis of the source code data structure.

5.5.2. Abstract Syntax Tree

The abstract syntax tree (AST) is a representation produced after scanning and parsing the source text. An AST incorporates syntax analysis (i.e., we only get an AST if the source code is syntactically correct) and some static semantics analysis (i.e., parsers usually detect instances of variables not declared). The AST is the representation of choice for tools like code optimizers, generators, and syntax directed editors because it is easily manipulated, traversed, and analyzed. We use the AST as representative of the graph class of data structures. It is from 5 to 10 times larger than the program source, with a 1 to 1 correspondance between the AST nodes and the source tokens. Figure 5.9 shows the analysis of the software abstract syntax tree document using the criteria that we developed in Section 5.2.

(1) Internal Structure: Complex.

(2) Information Content: Syntactic Structure (explicit)

(3) Quantitative Information:
- Cost to Produce: 4.1 millisecs per node (Equation 6.3)
- Number of Elements: N=5.0L+247 (Equation 6.1)
- Kinds of Atomic Elements; Doubly linked nodes.
- Atomic Element Size: 104 bytes per node.

(4) Qualitative Characteristics:
- Natural Aggregation Units: statements, procedures, file subtrees.
- Natural Data Representation: Dependent binary, records with pointers.
- Links: None internal; node to attributes and node to source code (external).
- Persistence: Need to evaluate taking into account: space cost, cost to produce, cost to fetch, how many tools use it, and how much do we use static semantic information in the DBMS.

(5) Sharing Characteristics:
- Sharing Medium: Shared memory.
- Application Interaction Mode: Read Only, incremental I/O.
- Environment Heterogeneity: Heterogeneous architecture.

Figure 5.9: Analysis of the abstract syntax tree data structure.

5.5.3. Symbol Table

The symbol table (SYMTAB) is a tabular representation that incorporates much semantic information about a program. It is the result of both the parsing and static semantic analysis phases of compilation. Conceptually, a SYMTAB is a table that has an entry for every identifier (symbol) in the program. The entries contain a list of attributes of each identifier, like its type, if it has been initialized, and if it is used. It is our choice to represent the class of tabular data structures. The SYMTAB is used by tools that require static semantic information about the program like code generators and interactive debuggers. Figure 5.10 shows the analysis of the software abstract syntax tree document using the criteria that we developed in Section 5.2.

5.6. Database Designs

In this section we examine the different data base designs for the three data structures that we analyzed above. We first look at the possibilities for each database design dimension, granularity, number of relations, and data representation.

Granularity: We look at three levels of granularity: fine, medium, and coarse. At the fine granularity level we consider the smallest natural units of aggregation: line for source code, node for AST, and symtab entries and attributes for the SYMTAB. At medium granularity we consider the procedure unit of aggregation for the three data structures, and at coarse granularity we consider the module, which corresponds to a physical file, for the three data structures. There are other units of aggregation that we do not consider: for text we could look at the *token* for fine granularity, but we chose to go up to the line granularity because most tools operate on lines and because extracting tokens from the source is cheap and easy. In addition, between the fine granularity and the procedure unit of aggregation that we chose for medium granularity we have the expression and the statement units of aggregation, but we skip them because the problems they pose are the same as with procedure granularity but they save less space and time.

Number of Relations: There are three different cases where the number of relations vary:
- *1 Relation:* For all medium and large granularities. For fine granularity when: i) all fine grained elements are of the same kind, like text lines (LPT); ii) all fine grained elements are of similar kind, like tree nodes (NPT); iii) we group together all associated elements, like symbol table entries and their attributes (SPT).
- *2 Relations:* For fine granularity, when we have two major kinds of elements like symbol table elements and attributes or program code and comments.
- *3 or More Relations:* For fine granularity, when we have fine grained objects of

---

(1) Internal Structure: Complex.

(2) Information Content: Static semantics (explicit)

(3) Quantitative Information:
- Cost to Produce: 0.8 millisecs per entity (attributes and Symtab entries)
- Number of Elements: $A=6.7L+251$ attributes (Equation 6.4)
  $S=0.7L-38$ Symtab entries (Equation 6.5)
- Atomic Element Size: 94 bytes per symtab entry; 22 bytes per attribute.
- Kinds of Atomic Elements; Symtab entries, and attributes.

(4) Qualitative Characteristics:
- Natural Aggregation Units: Symtab entry plus attributes, scopes, procedures.
- Natural Data Representation: Dependent binary, hash table of symtab entry records linked to attribute records.
- Links: Symbol table entries to attributes (internal); none external.
- Persistence: Yes.

(5) Sharing Characteristics:
- Sharing Medium: Central repository, and shared memory.
- Application Interaction Mode: Read only, incremental I/O.
- Environment Heterogeneity: Multiple architectures, same language.

Figure 5.10: Analysis of the symbol table data structure.

---

different kinds like comments, data definitions, and macro definitions in source code, or different kinds of attributes for the symbol table (R-APT).

Data Representation: We explain what representations make sense for each different schema.
- *ASCII:* The ASCII representation makes sense for all medium and large granularities when the objects have internal structure, like the AST at procedure per tuple. In addition, it is also appropriate for the small granularity whose FE representation is also ASCII, like program text lines.
- *Independent Binary:* This representation comes for free with non-ASCII fine granularity designs like node per tuple for the AST. This is because the DBMS has the same fine granularity types as the FE (e.g., floats and integers). It also makes sense for medium and large granularities in a heterogeneous architecture and/or language environment, or if they require BE manipulation.
- *Dependent Binary:* This representation makes sense for medium and large granularity designs, especially when we do not need the BE to manipulate the internal structure, or when the BE and the clients all use the same internal binary format.
- *Compressed at Frontend:* Frontend compression makes sense for large and medium granularity objects that will not be manipulated by the BE.
- *Compressed at Backend:* Backend compression makes sense for objects of all granularities that need to be manipulated in the BE.

In the next section we look at the possible schemas for the three data structures: source code, AST, and SYMTAB.

5.6.1. Program Text

For program text, the possible schemas are line per tuple, procedure per tuple, module per tuple, and relation per structure kind and line per tuple:

- Relation Per Structure Kind, Line Per Tuple (R-LPT): We have a different relation for each kind of source code structure like comments and data structure definitions, for example:
  lpt_comment (file_name, line_no, comment)
  lpt_datadef (file_name, line_no, datadef)
- Line Per Tuple (LPT): **lpt_text** (file_name, line_no, line)
- Procedure Per Tuple (PPT): **ppt_text** (file_name, line_no, proc)
  Where line_no refers to the starting position of the procedure in file_name.
- Module Per Tuple (MPT): **mpt_text** (file_name, module)

5.6.2. Abstract Syntax Tree

For the AST, the possible schemas are relation per node type and node per tuple, node per tuple, procedure per tuple, and module per tuple.

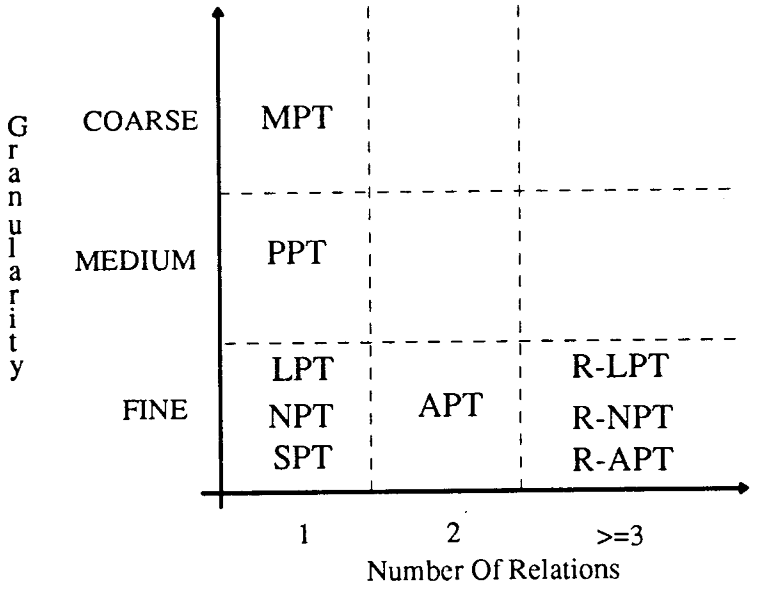
- Relation Per Node Type, Node Per Tuple (R-NPT): We have a different relation for each type of node, with 1 node per tuple, for example:
  id_node (file_name, line_no, col_no, parent, objid)
  var_node (file_name, line_no, col_no, parent, objid)
- Node Per Tuple (NPT): **npt_ast** (file_name, line_no, col_no, operator, parent, objid)
- Procedure Per Tuple (PPT): **ppt_ast** (file_name, proc_name, proc)

- Module Per Tuple (MPT): **mpt_ast** (file_name, module)

5.6.3. Symbol Table

For the SYMTAB, the possible schemas are relation per attribute type and attribute per tuple, attribute per tuple, symtab entry per tuple, procedure per tuple, and module per tuple.

- Relation Per Attribute Type, Attribute Per Tuple (R-APT): A different relation per type of attribute, with an attribute per tuple, for example:
  ste_type (sym_objid, value)
  ste_scope (sym_objid, value)
- Attribute Per Tuple (APT): Each SYMTAB entry is stored in 1 relation, and all its attributes are stored in another relation:
  apt_symtab (file_name, objid, symbol)
  apt_att (kind, value, sym_objid)



Key:

MPT: Module per tuple (all reps).	SPT: SYMTAB entry per tuple (SYMTAB).
PPT: Procedure per tuple (all reps).	APT: Attribute per tuple (SYMTAB).
LPT: Line per tuple (text).	R-NPT: Rel. per node type, node per tuple.
NPT: Node per tuple (AST).	R-LPT: Rel. per structure type, line per tuple.
	R-APT: Rel. per attribute type, attribute per tuple.

Figure 5.11: Graph of granularity and number of relations for the different program representation designs.

- SYMTAB Entry Per Tuple (SPT): each SYMTAB entry and its attributes is stored in a single relation:
  ste_symtab (file_name, objid, symbol, scope_value, type_value, status_value,...)
- Procedure Per Tuple (PPT): **ppt_symtab** (file_name, proc_name, proc)
- Module Per Tuple (MPT): **mpt_symtab** (file_name, module)

Figure 5.11 shows a graph in which the DB designs for the different representations are compared in terms of tuple granularity and number of relations. The dashed horizontal lines separate granularity into "classes", so that 2 designs in the same class have equivalent granularity. Figure 5.12 is a two dimensional graph of granularity and data representation. In it we indicate which of the database schemas that we use are appropriate for each data representation. Notice that the schemas appropriate for the dependent binary and FE compressed representations, the two rightmost columns, are those that do not require backend manipulation. Because fine granularity only makes sense if we want DBMS manipulation, there are no fine granularity designs in those two columns. We include BLOBs in both the coarse and medium granularity rows because binary objects come in different sizes. We do not show a graph that relates the number of relations and the data representations because they are independent of each other.

5.7. CASE Operations On Code

In this section we look at the kinds of operations that the software code documents need to support in a CASE environment. The objective is to illustrate the many different activities that tools expect from a data manager, the kinds of access to the data that these operations require, and to determine the performance requirements of the different operations. Code is worked on during the code generation, system testing, and system maintenance phases of the software lifecycle. The tools that are used during these phases include: text editors, syntax directed editors, compilers, debuggers, and profilers. We have classified the operations of tools on software into 5 categories: fetch, store, delete, update, and query (interrogation). We provide examples of each kind of operation on the SPT SYMTAB database design. A typical fetch query is:

```
retrieve (s.all) from s in ste_symtab
where s.scope_value = 3 and s.file_name = "screen.c"
```

A typical store query is:

```
append (ste_symtab.scope_value = 3, ste_symtab.type_value = integer,
        ste_symtab.file_name = "screen.c")
```

A typical delete query is:

```
delete ste_symtab where ste_symtab.scope_value = 3
```

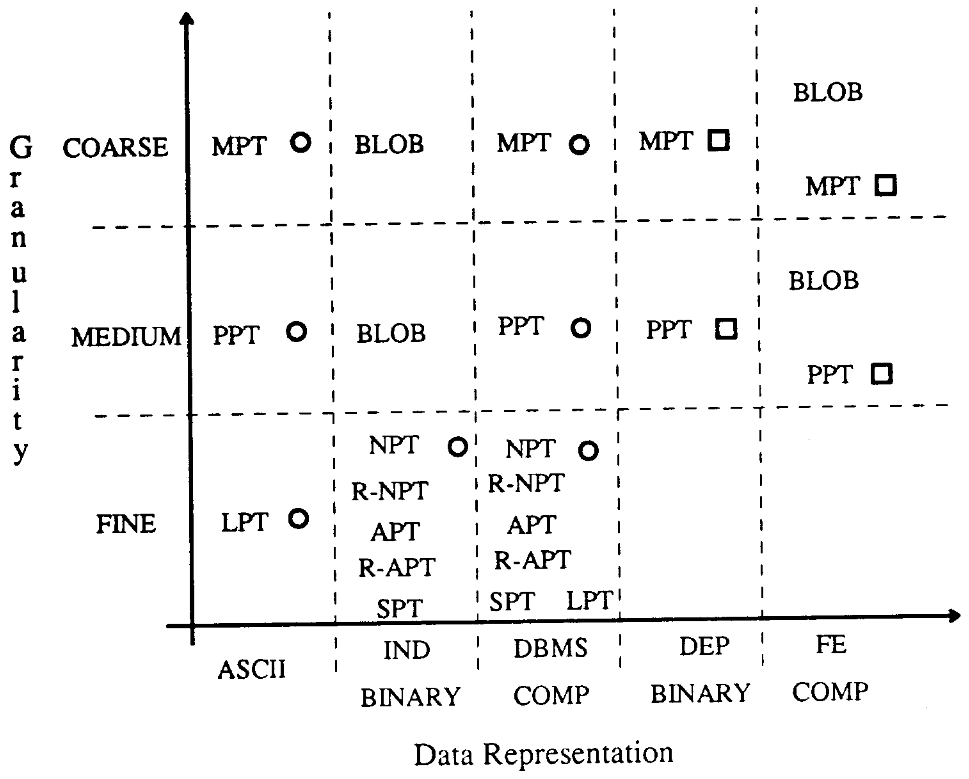
A typical update query is:

```
replace ste_symtab (status = "used")
where ste_value.value = "i" and ste_value.scope = 3
```

A typical interrogation query is:

```
retrieve (s.name) from s in ste_symtab
where s.scope_value = 3 and s.type_value = integer
```

The difference between fetch and query is that fetch brings to the FE sets of related



KEY:

○ ⟶	BE manipulation required	BLOB ⟶	No internal structure
□ ⟶	No BE manipulation required	PPT & MPT ⟶	Internal structure

Figure 5.12: Graph of granularity and data representation for the different program representation designs.

objects and requires much higher performance, whereas query provides any kind of information that the database supports and performance is less critical.

The kinds of information that CASE operations on code require include the comments and the names of program entities (which, unless they adhere to a grammar similar to that of a programming language as in [6], can be of any kind), the static semantics, the syntactic structure, the control and data flow graphs, and algorithmic complexity and correctness. The functionality demands that we have fine grained access for interrogation and intra-object references. The performance requirements on fetch, store, delete, and update operations are on the order of milliseconds (msecs) per fine grain entity, while on interrogation they are on the order of 0.1 to 1 seconds (secs) per query.

5.8. Summary

In this chapter we provide a framework for analyzing data structure characteristics in the context of a CASE DBMS environment. We first looked at data structures, and the characteristics that are important: their internal structure, their information content, their quantitative and qualitative characteristics, and their sharing characteristics. We next examined the three main database schema design dimensions for the data structures: number of relations, granularity, and data representation. We moved on to examine SE documents and the kinds of data that arise in SE: textual, tabular, graphical, and unstructured data classes. We then analyzed the program code data structures that we chose for our experiments, using the framework and criteria we derived in the earlier sections. We closed by examining the CASE operations on program code.

CHAPTER 6

DATABASE DESIGN EXPERIMENTS

6.1. Introduction

In this chapter we report on a suite of experiments that we designed and to see how different database designs for major CASE data structure classes perform in terms of time and space. The program representations that we choose, program source code from the textual class of data structures, an abstract syntax tree from the graphical class of data structures, and a symbol table from the tabular class of data structures, are the same that we analyzed in Chapter 5. This chapter, by focusing on the database aspects of these data structures, complements the in-core focus of Chapter 5. The database designs vary in three dimensions: granularity, number of relations, and data representation. They were chosen from the designs that we developed in Section 5.6. The experiments consist of storing and fetching the different database designs between a frontend process and a backend data repository. The environment simulates a client-server architecture, and the repository, a next generation DBMS, is like the one we believe that a CASE environment should have. We divide the database storage operation into three important phases: data extraction, data conversion, and database query. Fetching data from the database also has three main phases: database query, data conversion, and data structure reconstruction.

The rest of this chapter is organized as follows. Sections 6.2 to 6.6 set the stage for the experiments. Section 6.2 provides the rationale for conducting them, Section 6.3 states what we expect the results to show, while in Section 6.4 we detail what the experimental environment is like and how its different variables might affect our results. In Section 6.5 we explain what baselines we use for comparing our results, and in Section 6.6 we compute the data structure generation information in the FE process. Sections 6.7 to 6.9 explain, report, and analyze the results of experiments run on a suite of C source files from actual programs. The objective of these three sections is to record the space and time characteristics of the different database designs with real data. They report the results on the AST data structure only, because the results that we obtain for the text and SYMTAB are very similar, lead to the same conclusions, and would add bulk but little substance. We use the results of the source code when isolating the granularity effects, and of the symbol table when isolating the number of relations effects. Section 6.7 focuses on storing the structures to the DBMS, Section 6.8 on bringing them in from the DBMS, and Section 6.9 on the space usage characteristics of the different designs. Sections 6.10 to 6.12 report on experiments where we isolate the effects of the different database parameters. Section 6.10 reports on experiments that isolate the effects of granularity, Section 6.11 isolates the effects of data representations, and Section 6.12 isolates the effects of the numbers of relations. Section 6.13 compares the performance of our environment with that of several others and discusses the implications for our conclusions, Section 6.14 discusses what our findings imply across the spectrum of CASE data structure classes, and we close in Section 6.15 with a brief summary of our results.

6.1.1. Contributions

We know of no other work that focuses on evaluating the differences that number of relations, data representation, and granularity of storage have on space usage and operation performance.

In an experimental setup that simulates the TULUM architecture, we design a series of experiments that allow us to evaluate the performance and space utilization of the database designs that we developed in Ch. 5. We break down the storage and fetch operations into three phases, and measure the relative contribution of each phase to the total time cost in the face of different database designs. One suite of experiments measure the performance of different designs with real programs. A second suite isolates the effects of each database parameter on both space and performance. We end by generalizing the results to realistic environments, as well as to other data structure classes.

6.2. Rationale

As computers become cheaper and more powerful, new application areas are opening up for them, and databases are being called upon to store a greater volume and a wider variety of data. Our experiments are designed to explore in detail the practical aspects of the storage and retrieval of large volumes of complex data between applications and sophisticated data repositories in a CASE context. The specific goals are:

- To provide a methodology that can be used by CASE environment administrators to decide if, when, and how to store a specific data structure in the DBMS. The experimental design is simple to implement, yet yields the information needed to decide to store in the DBMS and to choose among different representations.
- To break down the save and fetch processes into their constituent parts, and to pinpoint how time is distributed among them and why. In addition, to see how the time distribution changes with the schema dimensions.
- To pinpoint how space is distributed both in the FE and in the data repository, and how both the space utilization and distribution change with the different schema dimensions.
- To determine which environment variables are important and why.
- To answer the ultimate question: can a data repository satisfy **both** the performance and functionality requirements of CASE data, at what cost in terms of space, and if so how?

Furthermore, the implementation and running of the experiments gave us many insights into the steps, factors, and techniques that are important in saving and fetching data structures to a repository.

6.3. Predictions

We make the following general predictions about the functionality, space utilization, and performance of different representations with respect to the three database schema parameters that we measure: granularity, data representation, and number of relations. As tuple granularity increases:

- Space utilization diminishes: Due to lower database overhead per entity, and potentially more compact data encoding.
- Speed of operations increases: Due to smaller DB sizes, leading to faster queries.

- Types of queries that can be asked diminish: Since we cannot use the DBMS built-in query system to answer small granularity queries.

As the number of relations increases:

- Space utilization increases if the total number of tuples increases, and shows no significant change otherwise. This is because most space usage comes from per tuple overhead.
- Speed of operations decreases if the total number of tuples increases, and shows no significant change otherwise. This is as predicted by equation 6.11.
- Types of queries that can be asked does not change. The way the queries are formulated would change, but the same operations and information are available.

As data coupling increases (i.e., the difference between data representations diminishes):

- Space utilization: Depends on the specific data representation. ASCII will probably be larger, and binary more compact.
- Speed of operations increases: Since we minimize data conversion costs.
- Types of queries that can be asked: May increase if the data representation can be handled directly by the BE, or diminish otherwise unless we create a new ADT with operators to handle the chosen data representation (leading to extra complexity).

In Figure 5.11, which compares granularity and number of relations, this means that operation speed increases as we go up to larger granularity and left to fewer relations; that space utilization increases as we go down to smaller granularity, and right to more relations; and that functionality increases as we go down to finer granularities. In Figure 5.12, which compares granularity with data representation, this means that operation speed increases as we go up to larger granularity; that space utilization increases as we go down to smaller granularity and right to more compact representations; and that functionality increases as we go down to finer granularities and left to the independent representations.

6.4. Experimental Environment

In this section we describe the hardware and software environment where we conducted our experiments. In both the software and hardware components we emulate a realistic CASE environment, like those described in Chapter 4. The architecture, illustrated in Figure 6.1, is a straightforward client-server setup, with a powerful server running the data manager code. The server has a large and fast disk that holds the database. The client has a small local disk for swapping, and uses UNIX sockets via an ethernet LAN to communicate with the server.

In the client we run a frontend (FE) program that simulates the CASE application. The FE program implements a compiler frontend that includes a scanner, a parser, and a static semantics analyzer for C programs written in Common-lisp [88]. It takes as input C source code files, and produces a parse tree and a symbol table for each file. Our system, illustrated in Figure 6.2, includes the three first phases in a typical compiler (scanner, parser, static semantics), with two DB specific modules at the end: the DB Send module takes the text, AST, or SYMTAB and sends them to the DBMS, while the DB Receive module fetches the text, AST, or SYMTAB from the DBMS. We elaborate

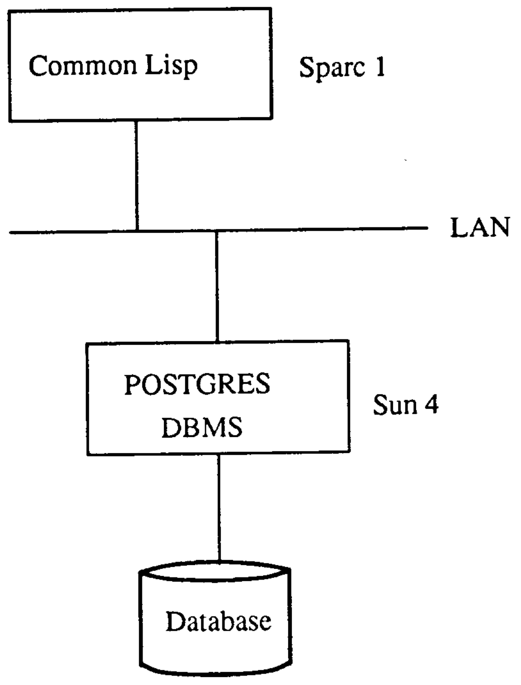


Figure 6.1: Experimental environment.

more on the DB send and DB receive processes in sections to follow.

In choosing the BE data manager we have been heavily influenced by the current research directions in the DBMS community, more concretely by the inclusion of object and knowledge support in the latest generation of systems. As we show in Chapter 3, the features offered by several of these systems overcome many of the traditional DBMS shortcomings while providing enhanced capabilities. Several research prototypes of DBMS's that incorporate rules systems and object support are currently available [59, 36, 37] , and commercial relational systems with these features are expected soon [79]. We considered many of the requirements of CASE environments, like an unlimited storage space, concurrency control, and version support (refer to Chapter 3 for a more complete list). Our choice is the POSTGRES third generation DBMS research prototype [13], that fulfills a large subset of the requirements. In particular, it possesses the three main attributes that our system requires: advanced data, object, and knowledge (rules) services. Our examples use the POSTGRES rules system PRS2 [12] and the POSTGRES query language, POSTQUEL [13]. POSTGRES can be considered both an extended relational and an object oriented system since it is grounded in the relational model and also includes the main features of object oriented systems. In POSTGRES, and in our examples, classes are equivalent to relations and instances (objects) are equivalent to tuples.

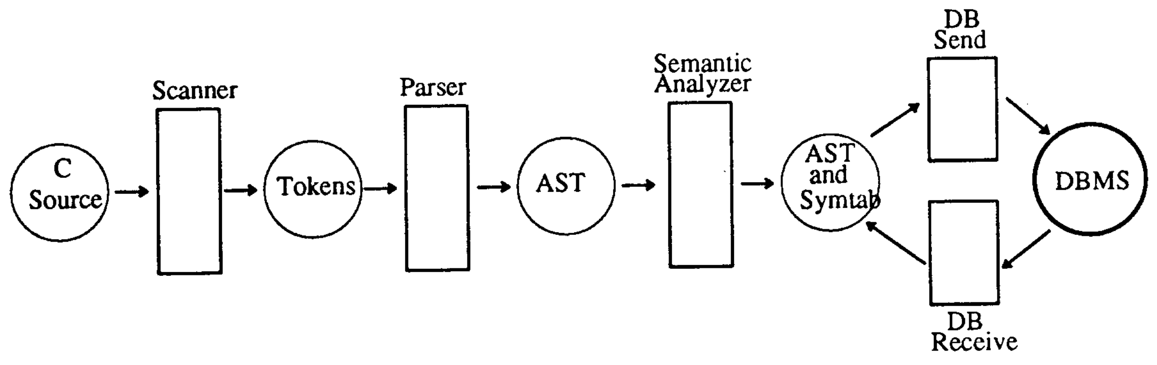


Figure 6.2: Frontend software organization.

Here we explain why we do not use the TULUM architecture (which we covered in Chapter 4), in particular the feature where we have a DBMS+ running on the server and a DBMS- running on the client, and where the DBMS- caches local database data in the client. First, the great majority of systems implement the traditional client-server architecture. In addition, the main advantages of having a DBMS+ and DBMS- are not related to single client performance, but rather they are increased client autonomy and increased transaction throughput under concurrent access. Neither client autonomy nor transaction throughput is a factor in our experiments, and both are outside of the scope of our research. The two factors that can have an impact are the difference in CPU speeds between the client and the server, and the network transmission overhead. To eliminate the CPU speed effect, both our client and server run at the same speed. The network transmission overhead is not very significant (we justify this statement with performance data in Section 6.13).

We use two different data sets. The first one, *c-samples1* consists of a set of 30 C code files that belong to programs that run under UNIX and the X Window System (like POSTGRES and the TWM window manager). They were selected in the range of 50 to 2500 lines. The second one, *c-samples2*, is designed to isolate the effects of granularity and consists of 10 files, each with the same total number of nodes for the AST, and the same total number of lines for the source code. Each file consists of a series of identical procedures (the only difference is one character in the procedure name so as to avoid compiler errors). The net effect is to have files with the total same number of elements, but with different number of elements per procedure.

The FE and BE can reside on the same, or on different machines. The server is a Sun 4-330, the client is a SparcStation 1, both running SUN OS 4.1. In the next section we cover the important environment variables that can impact our results.

6.4.1. Environment Variables

In our software and hardware environment, there are a number of variables that can have a significant impact on the performance of the operations we measure. Most of them have to do with the BE component. We explain each in turn.

- *DBMS buffer pool size*: The buffer pool is a main memory cache for the disk based database managed by the DBMS. In essence, the larger the better. If we deal with databases that are too large for main memory, the buffer pool size (assuming reasonable buffer management) heavily influences the number of I/O's per query.
- *Warm or Cold Queries*: We say that a query runs on a cold system when the relations it accesses are not in the buffer pool, and they have to be brought in from disk during query processing. After the query has been run several times, and the buffer pool cache stabilizes, the numbers are called warm. In POSTGRES, this has no effect during storage operations since, in both cases we do the same processing and the same number of I/O's.
- *Space overhead per tuple*: In order to provide good performance and sophisticated services, each DBMS tuple has some information attached. Different DBMS's represent tuples differently, as well as provide different amounts and kinds of functionality which may require space for each tuple. The overhead per tuple can have a tremendous impact on the space utilization (and therefore on the performance) of a given db schema.
- *Cardinality of relations*: Relations with large numbers of tuples may, depending on the access methods used, take longer to process.
- *Data distribution on disk*: Clustering together data that is accessed in a single query lowers the numbers of I/O's.
- *Compiled queries*: Compiled queries have lower DBMS overhead than interpreted ones because they skip the parsing and optimization phases. In addition, the query plan might be different.
- *Access methods*: A DBMS access method is an algorithm used to locate the desired data. The kind of access method used to process a query can have a strong impact on how it performs. In addition, the presence of an index on a given field can cause different formulations of the same query to perform very differently. Finally, indexes constitute another form of space overhead.
- *Support for binary large objects (BLOBs)*: This is important for the large and medium granularity schemas. If the DBMS limits the size of objects, large objects have to be "broken up" at some cost. In addition, if the DBMS does not support binary data, there will be a conversion and space cost.
- *Support for multiple-append*: Because of the high overhead associated with query processing, a significant way to improve performance is to transfer as much data as possible per query. Without multiple-append, 1 query per tuple is necessary to send new data to the DBMS. With multiple-append we can use a single query to append many tuples to a relation.
- *Query formulation*: A given query can be formulated in many different ways, some more efficient than others depending on the available access methods as well as the optimizer used by the DBMS.
- *The CPU to I/O speed ratio*: All the DBMS operations that we measure have both a CPU and an I/O component. Using a given disk on a 1 MIPS machine will yield different results than using the same disk with a 10 MIPS CPU.

The value of the relevant environment variables in our experiments is:

- The buffer pool size is 64 8k pages. This means that the server devotes 0.5 Mbytes to cache the database. For our experiments, this is enough to hold the *working set* of

our queries, which means that a page only needs to be brought in once during a given query (there is no thrashing).
- All query results reported are warm, that is after the buffer pool cache stabilizes.
- The overhead per tuple is 78 bytes. This is pretty large, especially for small granularity designs when the overhead may be larger than the data. But it is necessary to support sophisticated POSTGRES features like historical data.
- For all fetch queries, the database that we use is identical.
- There is no disk data clustering, since POSTGRES v2.1 does not support it. Nevertheless, since we fetch data pretty much in the order it was saved, and POSTGRES v2.1 stores contiguously the data saved at the same time, most of it is in effect contiguous on disk.
- There are no compiled queries, since POSTGRES v2.1 does not support them. This adds a small overhead to the BE times.
- We use the B+ Tree access method on the fields that we use to fetch the data. We do not count the B+ Tree update overhead when appending data since we do it via the file copy command which does not support index updates.
- There are no BLOBs, since POSTGRES v2.1 does not support them. POSTGRES imposes a size limitation of 8 Kbytes, and does not support binary data for large objects. We therefore limit the size of our objects to 8 Kbytes, store large objects in ASCII format, and obtain the binary representation results by extrapolation.
- We implement multiple append through the file copy command, in which we write the stream of tuples to an intermediate file, send a *copy* command to the BE, and the BE reads the tuples from the file[12] This forces us to adjust our results to account for the extra file handling operations.
- All queries are formulated in the simplest and most efficient way, using the B+ Tree access methods.
- We use a disk with average seek time of 25 msecs, and a client and server rated at 10 MIPS. We therefore have a 10 / 25 (MIPS per msec) CPU to I/O ratio.

6.5. Baselines

In order to evaluate the results of our experiments we need some yardsticks. For fetching and restoring data structures from the DBMS, we have chosen the time to create the data structure from source. That is, for the AST the time to scan and parse the source code, and for the symbol table the time to perform static semantic analysis over the AST. This makes sense because, in the absence of persistence, any tool that needs to use the data structure has to create it (ie., the AST is recreated during each compilation even if only a single line has changed), and also because, except for the source code, it is a CPU bound operation of linear algorithmic complexity. It therefore provides both a realistic baseline, and one that improves as CPU speeds increase and memories get larger. There is no comparable measure for the source code since source code is produced manually by the software engineers at much slower and widely varying rates [87].

[12] We can use files to exchange data between the client and the server since they share a file system via SUN's Network File System (NFS).

For the storage of data structures to the DBMS, the fastest permanent save operation that is generally available is to a local file[13]. We therefore use the time to save the data structures to a file as our yardstick. But, since the results of our storage experiments show that DBMS storage and file storage times are virtually identical, we also use the time to create the data structure to compare the data storage times.

Finally, to compare the space usage of the different schemas and representations we use as a yardstick the amount of space occupied by the data structures when they are in-core in the FE in several representations. This allows us to see how significant the space overhead is for each representation in the different media. We measure the FE space usage in two ways: space used by the data only, and the space used by the data plus the data structure overhead (like headers and pointers). We present the actual measurements in the relevant sections ahead.

6.6. Data Structure Generation

In this section we describe how we measured the AST and SYMTAB generation parameters, and derive formulas for the number of elements and for the cost, in milliseconds, per element. The cost per element equations are computed by dividing the time to create the structure by the number of elements at each measurement point. We also compare the performance of our FE environment with that of a standard commercial system. We have several objectives: to show how we can compute the quantitative characteristics of a data structure (which were described in Section 5.2), to compute the quantitative characteristics for inclusion in the data structure analysis forms for the program text, AST, and symbol table presented in Figures 5.8, 5.9, and 5.10 respectively, and to use as a baseline for the rest of the experiments. Our suite of generation experiments measures the time and space required to create the data structures from the source code. They consist of running the parser over the files in the *c-samples1* data set. We measured the time to create the data structure, the number of lines per file, and the number of elements per data structure for each file.

• **AST:** Our parser uses a simple bottom-up LR(1) parsing algorithm, with a single pass over the tree. Given that the number of tree nodes produced is always the same for a given code fragment, and that program metrics studies [87] have shown that programmers tend to use similar programming structures with regularity, we postulated that the number of nodes in an AST can be approximated by a linear function on the number of code lines. And, since parsing time is a linear function of the number of nodes, the time to create the tree can be approximated by a linear function on the number of code lines[14]

To check our hypothesis, we performed linear regression analysis of both nodes as a function of lines, and time to create as a function of lines. In both cases, we got

[13]Saving to non-volatile memory would be faster, but non-volatile memory is neither widely available nor cheap. Therefore this is not a realistic option.

[14] Lines of code does not include empty and comment lines [87]. A file will yield the same AST regardless of the number of empty and comment lines present in its source code. In addition, the overhead to "scan" over the empty and comment lines tends to be rather small.

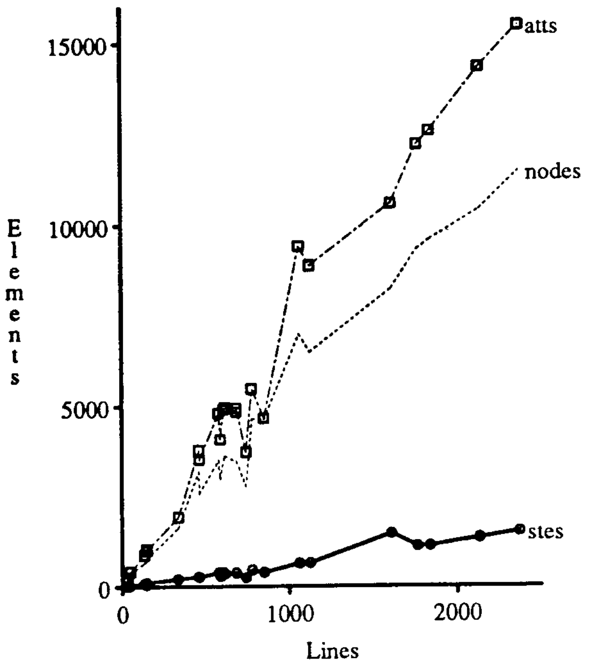


Figure 6.3: Number of elements as a function of file size (lines).

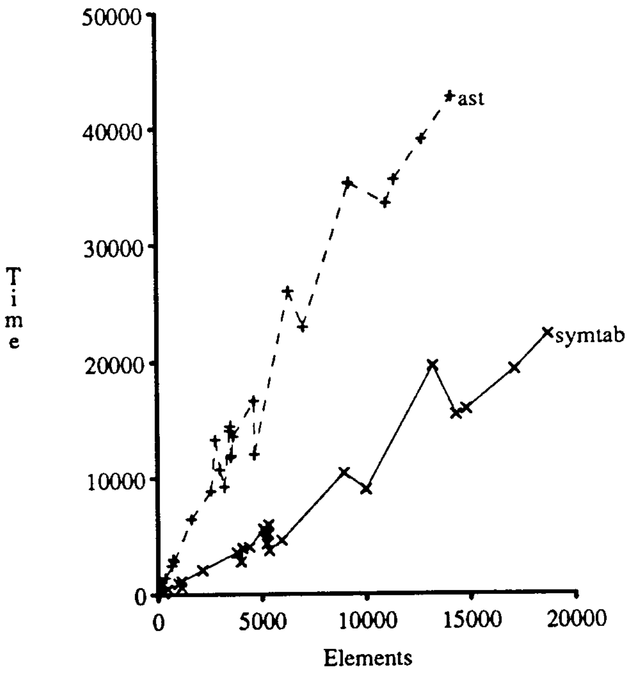


Figure 6.4: Time to create the data structures as a function of file size.

equations with correlation coefficients > 0.98, and significant at the 0.005 level. We also got the linear fit for cost per node as a function of lines, and found it to be constant[15] Equation (6.1) describes the number of nodes as a function of number of lines, equation (6.2) describes the time to create the AST as a function of the number of nodes, and equation (6.3) describes the cost, in milliseconds (msecs), per node:

$$N=5.0L+247 \tag{6.1}$$

$$T=3.1N+1481 \tag{6.2}$$

$$C=4.1msecs \tag{6.3}$$

Where:

N is the number of nodes,
C is the cost per node,
T is the time to create the AST, and
L is the number of lines.

The constant term in equation (6.1), 247, is due to a fixed number of node overhead per AST generated. The constant term in equation (6.2), 1481, is due to the generation of those nodes plus some constant processing overhead per file. Figure 6.3 shows the graph of number of elements as a function of code lines, and Figure 6.4 shows the time to create the structures as a function of elements, nodes for the AST, and attributes (atts) and symbol table entries (stes) for the SYMTAB. As you can see, the curves are linear both for the number of elements per line, and the time to create per element. The curves and equations therefore provide a solid baseline against which we can compare the times to fetch and store the structures from the DBMS.

• **SYMTAB:** In a similar way as with the AST, the number of symbol table entries and attributes is always the same for a given code fragment. We therefore postulate that the number of symbol table entries and attributes in a SYMTAB can be approximated by a linear function on the number of source lines. In addition, since semantic analysis is a linear function of the number of symbol table elements and attributes, the time to create the symbol table can be approximated by a linear function on the number of source lines.

To check our hypothesis, we performed linear regression analysis of both symtab entries and attributes as a function of lines, and time to create as a function of lines. In all cases, we got equations with correlation coefficients > 0.98, and significant at the 0.005 level. We also got the linear fit for cost per element as a function of lines, and found it to be constant. Equation (6.4) is for the number of attributes, and (6.5) for the number of entries:

$$A=6.7L+251 \tag{6.4}$$

[15] This is true as long as the FE process is small enough to fit in the available RAM. When the process grows larger than memory, paging distorts these results. In our environment, this occurs for programs larger than 2500 lines.

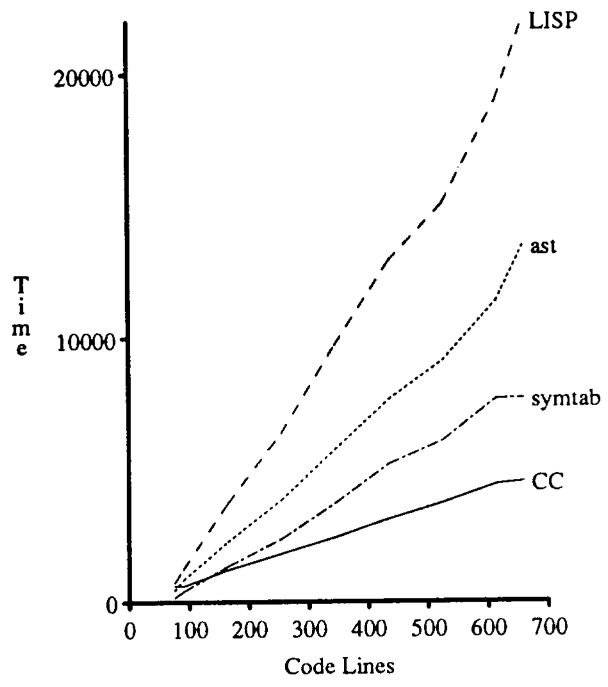


Figure 6.5: Time to compile with /bin/cc and our LISP environment.

$S=0.7L-38$ **(6.5)**

Equation (6.6) is the time to create the symbol table from the AST, and (6.7) is the cost per entity:

$T=1.2E-883 msecs$ (6.6)

$C=0.8 msecs$ (6.7)

Where:

A is the number of attributes,
S is the number of entries,
$E = A + S$, is the number of entities,
C is the cost per entity,
T is the time to create the SYMTAB, and
L is the number of lines.

As in the case of the AST, the close fit between number of elements and number of lines allows us to use the time and cost to create per element as a baseline.

• **Common LISP vs the C Compiler:** Finally, in order to get a more realistic number for the time and cost to create the data structures, we compare the times to create the AST and the SYMTAB with our LISP environment, to the time to perform a full compilation with the standard UNIX C compiler, /bin/cc. In Figure 6.5 the CC curve is the total compilation time with the C compiler, the symtab curve is the time to create the symbol table in our LISP environment, the ast curve is the time to create the abstract syntax tree, and the LISP curve is the sum of the ast and symtab times. From the curves in Figure 6.5 it is clear that /bin/cc is significantly faster than our LISP environment, beating our environment even when we only create the AST or the SYMTAB. Equation (6.8) shows the LISP times, and (6.9) shows the C compiler times:

$$Y=34.8L-2161 \tag{6.8}$$

$$Y=6.9L+39 \tag{6.9}$$

If we assume that parsing and static semantics consumes about 50% of the total compilation time, /bin/cc is about 1 order of magnitude faster than our LISP code. This is due to the highly optimized nature of the C compiler, and the high overhead of the CommonLisp environment. We will take this into account when we make the comparisons with the database fetch and store times ahead.

6.7. DBMS Storage

In this section we report how we store the data to the DBMS, the important steps in the storage process, the results we obtain, and analyze those results. The steps involved in storing a data structure to the DBMS are:

(i) Extraction of relevant data from the internal data structure. This usually includes traversing the individual elements, and arranging them in a way suitable for shipment to the DBMS. We may also eliminate unnecessary fields (in the case of the AST we use an end-line-no field in the FE that is irrelevant in the DBMS), and add new ones (a *file_name* field that we need to reconstruct the AST in the FE). We elaborate more on the specific fields in Section 6.8 on space considerations.

(ii) Conversion from internal to external format. This can be achieved during

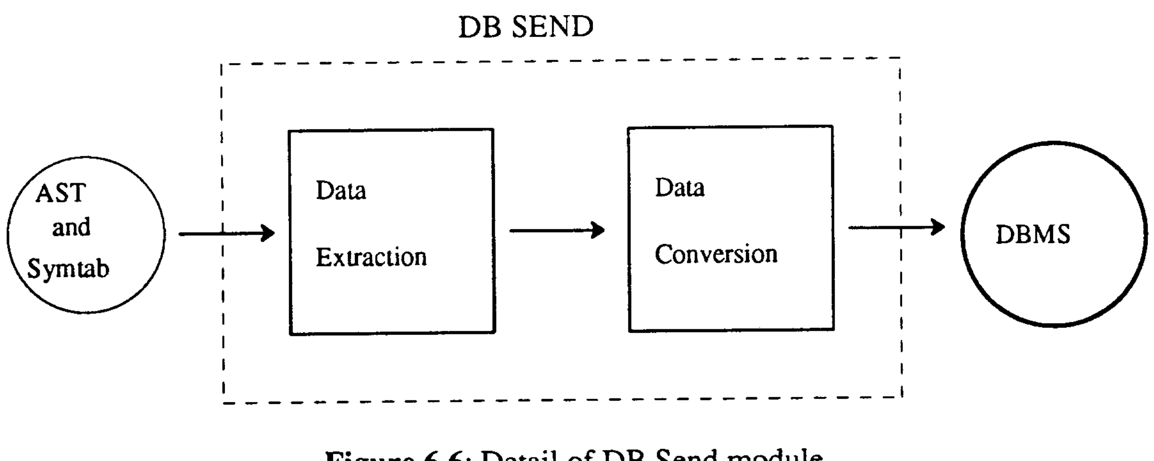


Figure 6.6: Detail of DB Send module.

extraction, or as a separate, latter phase.
(iii) Transmission from FE to BE: This can be done via shared memory, over the network, or via intermediate files. We use intermediate files to simulate multi-append.
(iv) BE receives query and data, parses and optimizes query.
(v) BE writes the data to the corresponding relations on disk[16].
(vi) BE sends acknowledgement to FE.

In our store experiments, **extract** includes step (i), **convert** includes step (ii), **fwrite** includes step (iii) as writing an intermediate file, and **query** includes steps (iv), (v), and (vi) treating the BE as a black box. Figure 6.6 illustrates this arrangement, with the data extraction module extracting the data, passing it to the data conversion module which converts it to the desired representation and sends it the DBMS. Extraction and conversion involve a constant number of passes over the data, getting those data items we need, and converting them (individually or in batch) to the chosen external representation. Both extract and convert are linear functions of the number of nodes and therefore of the number of source lines as in equation (6.10):

$$Y = mX + b \tag{6.10}$$

The fwrite and query times are a function of the number of tuples and the number of disk pages as in equation (6.11):

$$Y = f(t) + w * f(p) \tag{6.11}$$

Where:
w is a weight factor derived from the relative weights of I/O and CPU costs
t is the number of tuples, and
p is the number of pages.
Notice that *f()* is the same function for both terms. The weight factor w equalizes their impact. As we mentioned before, the fine granularity schemas do not require FE data conversion, since the FE and BE fine granularity types are equivalent. From the measurements, we compute the times to save to file and to the BE, for fine and coarse granularities, for binary data representation at all granularities, and for ASCII data representations at PPT granularities. The time to save to file is in equation (6.12) and to save to the DBMS in (6.13):

$$extract + convert + fwrite \tag{6.12}$$

$$extract + convert + query \tag{6.13}$$

To save to the DBMS, we do not count the *fwrite* time because of the way our store experiments are designed. With POSTGRES v2.1, we need to use an intermediate file to get multi-append. With direct FE to BE multi-append, the BE reads the data while the FE is transmitting it, so that we get I/O parallelism. Reading the data from file is a good approximation to reading directly from the pipe, with some extra file handling

[16]This statement is not strictly true. Most current DBMS's optimize disk write operations by just writing the database log page to disk at transaction commit time, and waiting to write the *dirty* pages until a certain threshold is reached.

overhead (namely file open and file close operations). But, given the large amounts of data we transmit, this overhead is not significant. For the PPT granularities, the binary storage times are computed by substracting the *convert* times from their totals, and adjusting the query and fwrite times in proportion to the size difference with the ASCII representation. For example, there are two external PPT AST representations sent by the DB-SEND module to the DBMS: AST_PPT_bin and AST_PPT_ascii. If we define *S(X)* to be the size of the data structure[17] X, then we compute the query time to store the AST_PPT_bin representation to the DBMS as equation (6.14):

$$query * \frac{S(AST\_PPT\_bin)}{S(AST\_PPT\_ascii)} \tag{6.14}$$

6.7.1. Storage Measurements

In this section we report on the times to save the AST to POSTGRES using the *c-procs1* data set (described in Section 6.4 on the experimental environment). Our objective is to see how different representations and designs vary with real life data when it is stored in the DBMS. We concentrate on the AST because the SYMTAB and program source results are similar and do not change our conclusions. We do not intend to isolate the effects of the different database dimensions here, that we do in subsequent sections. We look at the applicable dimensions here, granularity and data representation, but their impact is not as clear as in the sections where we isolate them. Because our AST designs do not vary in the number of relations, it is not discussed here.

We measured the times spent in the four phases we mentioned above: data extraction, data conversion, write to file, and multi-append query. Figure 6.7 shows the curves we obtained for the time to store versus number of nodes for the AST in the NPT schema, with a curve for each of the phases: fwrite, query, and extract. The create curve is the AST creation time, included for comparison purposes. Figure 6.8 shows the cost per node curves for each of the three phases, also for the AST in the NPT schema. The cost equations are computed by dividing the time at each phase by the number of elements at each measurement point. Figures 6.9 and 6.10 show the corresponding time and cost per node curves to store the AST in the PPT schema respectively. We computed the linear regression curves for the cost per node, and we list them in Table 6.1. Note that NPT convert is not listed, since the NPT schema does not require data conversion. As you can see from both the Figures and the Table, in both the PPT and NPT schemas *fwrite* is greater than *query* This is due to the relative efficiency of the CommonLisp and the C file I/O operations (since POSTGRES v2.1 is written in C). If we took the fwrite values as is, it would seem that saving to a file is slower than saving to the database. This is, of course, nonsense since storage to the DBMS includes saving to file, plus transmission costs, plus CPU processing costs. In addition, the POSTGRES file copy operation is essentially i) file open, ii) file read, iii) relation write, iv) file close, with the steps ii) and iii) repeated and overlapped until the end of file. If anything, the file read operations incur additional I/O costs. Therefore the query time is the

[17] We explain how we compute the ASCII and binary sizes in Section 6.9 on space considerations.

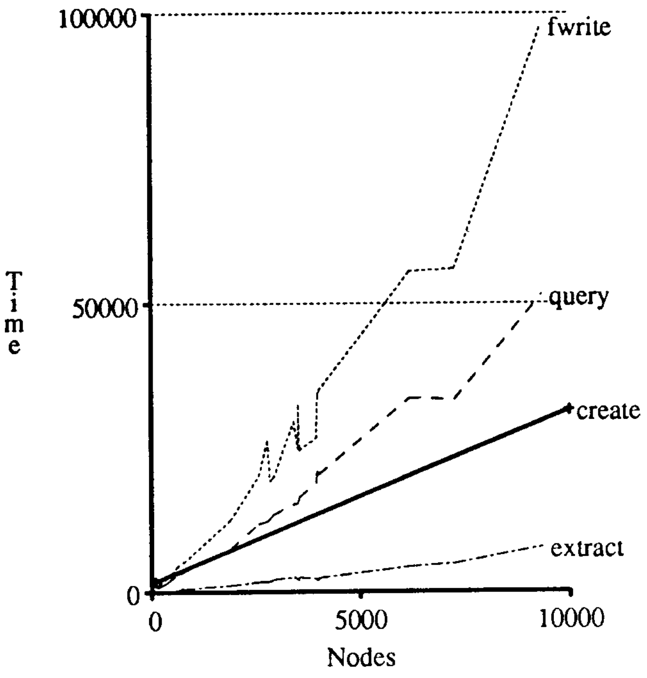


Figure 6.7: Time to store the AST in the NPT schema as a function of number of nodes.

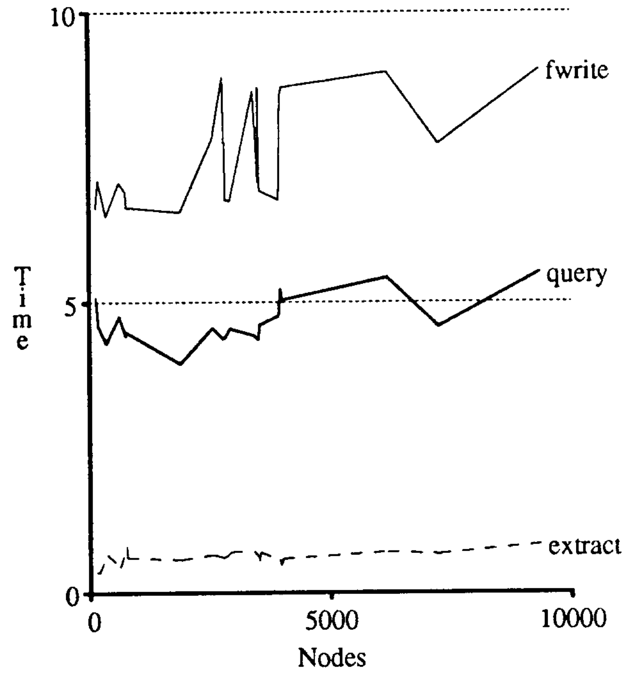


Figure 6.8: Cost per node to store the AST in the NPT schema.

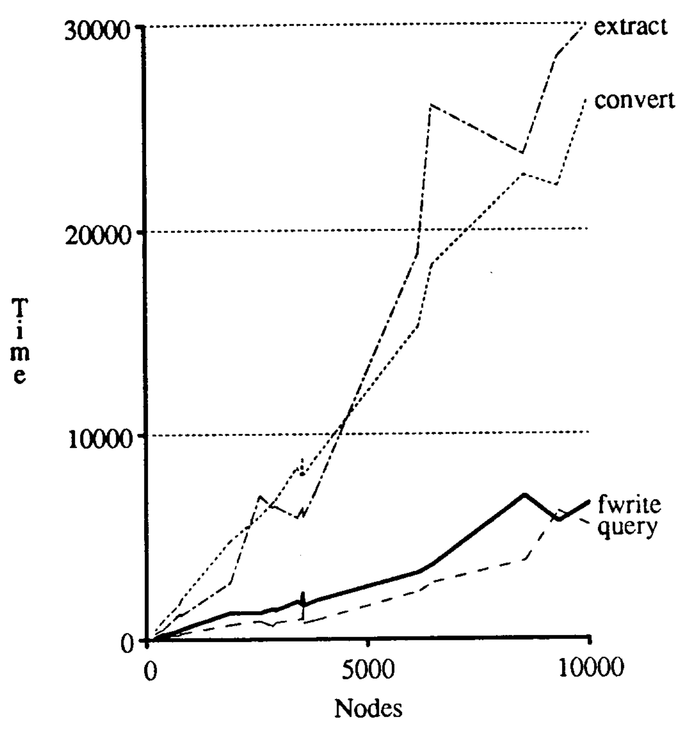


Figure 6.9: Time to store to the DBMS the AST in the PPT schema.

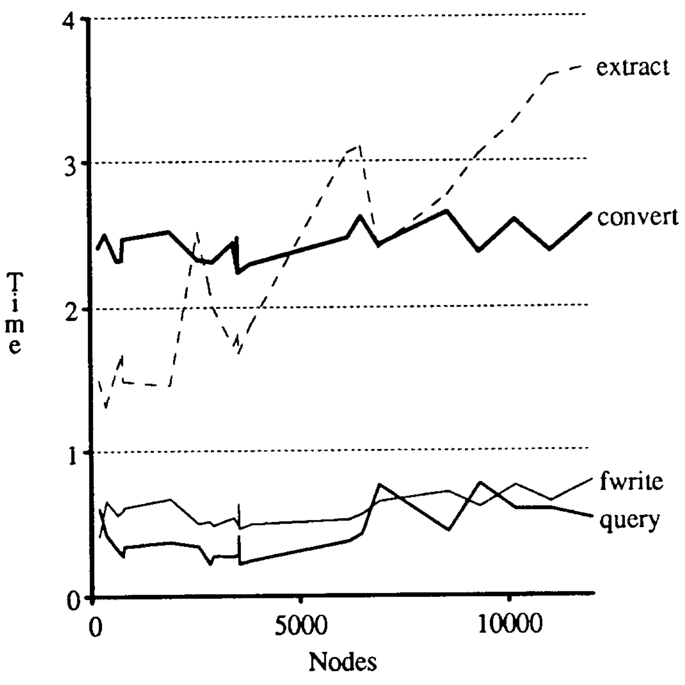


Figure 6.10: Cost per node to store to the DBMS the AST in the PPT schema.

fwrite time with some small CPU overhead. From this we conclude that the times and costs to save to a file are essentially identical to the times and costs to save to the DBMS using multiple append.

Table 6.2 shows the relative time distribution of saving the AST to POSTGRES in the different phases for the NPT and PPT schemas in the ASCII and binary representations. It was computed by using the linear regression cost per tuple for each phase, adding up the contribution of each phase to get the total storage cost per tuple, and finally computing the percentage contribution of each storage phase to the total. All costs are constant, meaning that their corresponding time curves are linear on the number of nodes. The exception, as you can see from Figure 6.10, is the *extract* cost for the PPT schema, which increases linearly with file size. In reality, it increases with granularity, which also increases with file size in our sample files. This is because the data extraction algorithm is quadratic on procedure size. We explain this in the section on granularity effects, and here we use the value for the median procedure size in our samples. In addition, the *Total* column reports the ratio of the total cost per tuple to that of creating the AST from scratch (the *Create* row in the Table), the baseline that we use since we cannot use file storage.

6.7.2. Storage Analysis

From Table 6.2 we can see that storing to the DBMS in all representations using multiple append is about as expensive as creating the AST, ranging from 0.6 to 1.2 times the create cost. Without multiple append, storage is several orders of magnitude slower, since we are limited to one query per tuple. If we use the fastest query time reported with POSTGRES v2.1, 0.1 secs, the query cost escalates to 100 msecs per node for the NPT granularity, and to 2 msecs per node for the PPT granularity (using the rough Figure of 50 nodes per procedure). The difference is two orders of magnitude

Schema	Extract	Convert	FWrite	Query
NPT	0.6	-	7.5	4.5
PPT	2.0	2.5	0.5	0.4

Table 6.1: Cost per node (msec) of different phases to store the AST.

Representation	Extract	Convert	Query	Total
Create	-	-	-	1
NPT bin	12%	-	88%	1.2
PPT bin	88%	-	12%	0.6
PPT ASCII	41%	51%	8%	1.2

Table 6.2: Relative time distribution to store the AST to POSTGRES.

for the NPT granularity (100 vs 4 msecs), and one order of magnitude for the PPT granularity (2 vs 0.4 msecs). Without multiple append, DBMS storage is prohibitively expensive.

The cost to extract the data is much higher for the PPT granularity because it involves creating and keeping track of many small temporary structures to store the data corresponding to each procedure, as opposed to a single large structure to store the nodes at the NPT granularity. When we need to convert the data from binary to ASCII with PPT ASCII, *convert* is as expensive as the query and extract phases combined, doubling the per tuple storage cost from 2.5 to 5.0 msecs. We would expect that at the NPT granularity data conversion would have an even more pronounced cost effect, increasing it by 4 times (as suggested by the convert cost in NPT fetch, Table 6.3). Finally, the contribution of *query* is about 10% for the PPT granularities, and dominates storage (88%) for the NPT design. This is due mainly to a much larger number of tuples and relation size of the NPT schema as opposed to the PPT schema. As we can see from the *Query* column in table 6.1, the actual cost difference between NPT and PPT is an order of magnitude.

6.8. Fetch from DBMS

In this section we report how we fetch the data from the DBMS, the important steps in the fetch process, the results we obtain, and their analysis. The steps to fetch a data structure from the DBMS are:

(i) FE issues query.
(ii) BE gets query, parses it, and produces query plan.
(iii) BE reads DB from buffer pool and disk if necessary.
(iv) BE computes result and transmits to FE.
(v) FE receives result (overlapped with (iv)).
(vi) FE converts data from external to internal format.

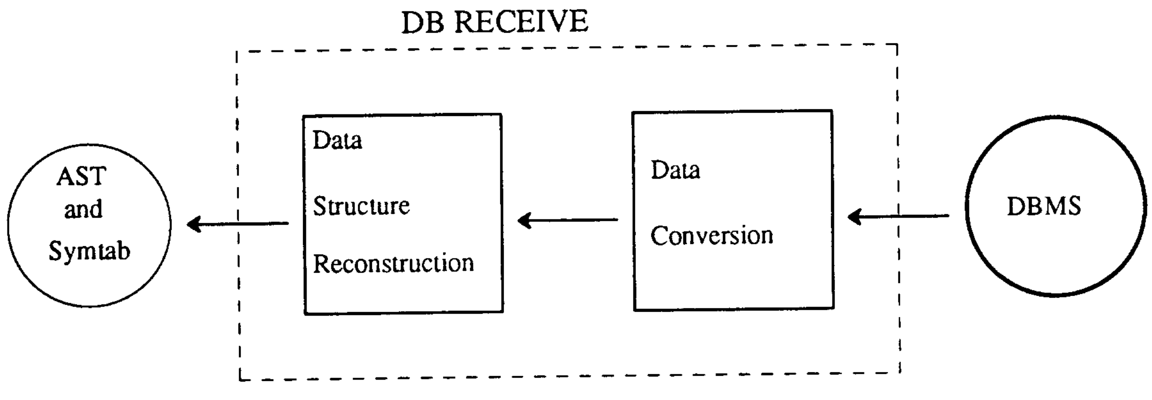


Figure 6.11: Detail of DB Fetch module.

(vii) FE reconstructs the binary data structure.

In our fetch experiments, **query** includes steps (i) thru (v), **convert** includes step (vi), and **build** includes step (vii). Figure 6.11 shows the process: DBMS, at the far right, executes the **query** phase and sends the data to the data conversion module which executes the **convert** phase. It converts the data to internal format and passes it on to the data structure reconstruction module, responsible for the **build** phase, which reconstructs the original data structure. As in the storage experiments, we treat the DBMS as a black box.

Since we use a B+ tree index to access the database, the query cost is a function of the number of tuples and the number of disk pages read, expressed in equation (6.15):

$$Y = t/d + w*(2 + p/d) \tag{6.15}$$

Where:
w is the I/O and CPU weight factor,
2 is the number of B+ tree pages read (assuming a large tree),
t is the number of tuples,
d is the number of distinct keys (the number of files for us), and
p is the number of disk pages read.

Both the time to convert and the time to build are a linear function of the number of nodes and therefore of the number of lines as in equation (6.10). Even though fine granularity does not require data conversion, in our measurements we include data conversion for the fine granularities, due to the FE to BE interface, which returns all values in ASCII representation. We do not count this when computing the total fetch times for the fine granularity representations, but nevertheless use it as an additional data point. From our measurements, we compute the times to fetch and recreate the data structures from the DBMS for fine and coarse granularities, for binary representation at all granularities, and for ASCII representation at PPT granularities. Equation (6.16) expresses the total time to fetch and recreate the data structure:

$$query + convert + build \tag{6.16}$$

Where *convert* = *0* for the binary representations. As we mentioned above, POSTGRES v2.1 does not support BLOBs, so we are forced to store the PPT schemas in ASCII representation. The query times for the binary representations are computed by adjusting them in proportion to the size difference with the ASCII representation, in the same manner as we do when storing data (e.g., for PPT query times we use $query*\frac{S(AST\_PPT\_bin)}{S(AST\_PPT\_ascii)}$ where $S(X)$ is the size of data structure X). Our approximation makes sense because, as equation (6.15) shows, the fetch query cost is a factor of the number of tuples and the number of I/O's. The major contribution comes from the I/O component, which is the one directly affected by the difference in tuple size from ASCII to binary.

6.8.1. Fetch Measurements

In this section we report on the times and costs to fetch the AST from POSTGRES. As with the storage measurements in Section 6.7, our objective here is to see how different representations and designs vary with real life data when fetching and restoring from the DBMS. We concentrate on the AST because the SYMTAB and

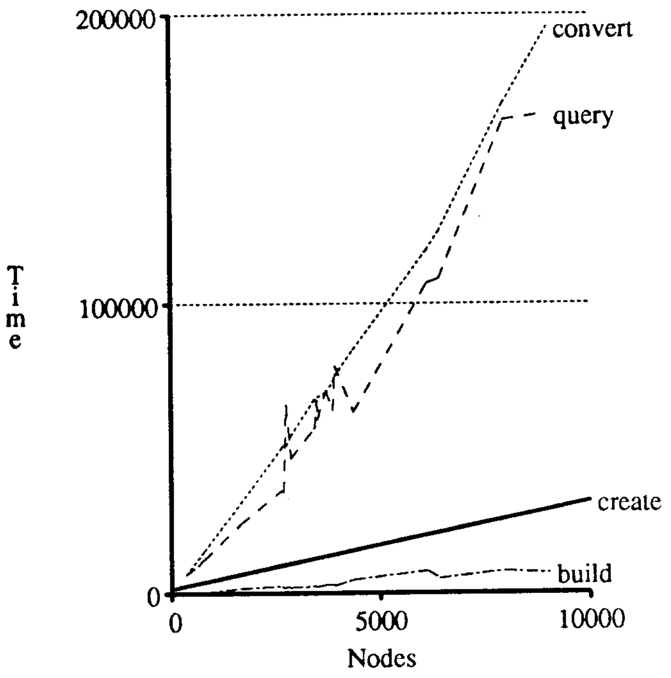


Figure 6.12: Time to fetch the AST from the DBMS in the NPT schema.

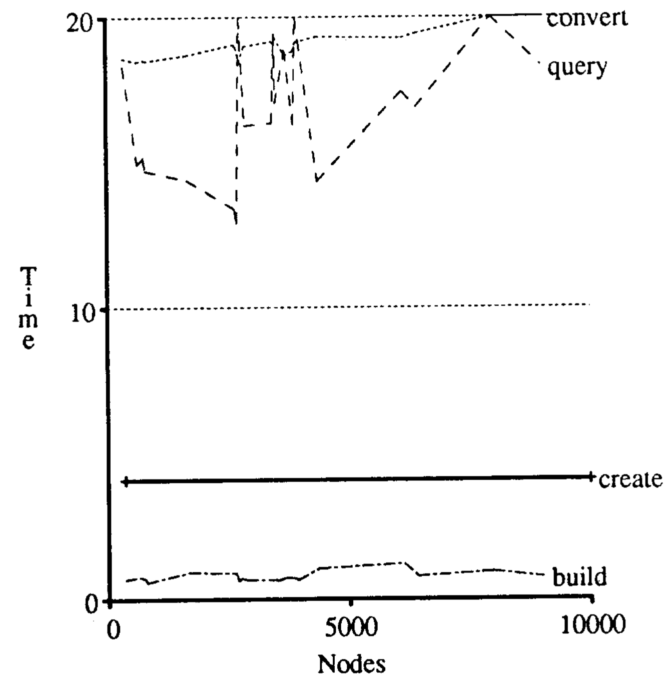


Figure 6.13: Cost per node to fetch the AST from the DBMS in the NPT schema.

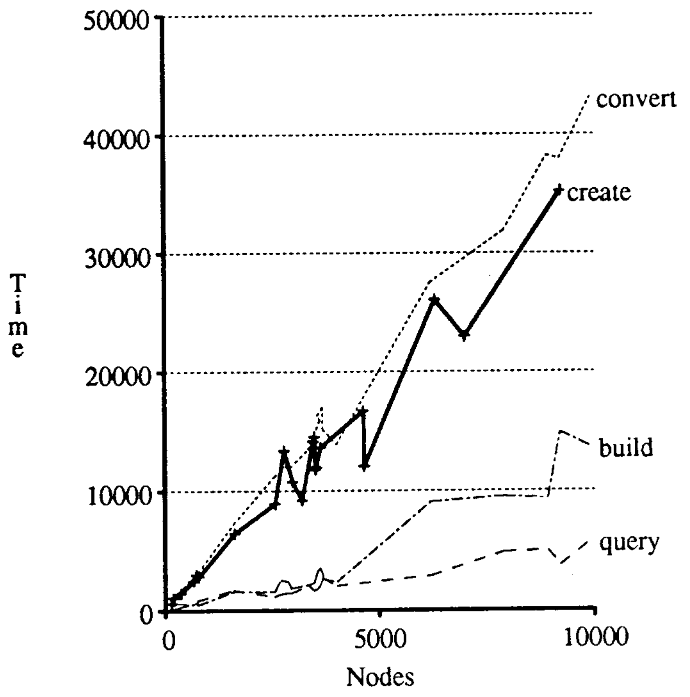


Figure 6.14: Time to fetch the AST from the DBMS in the PPT schema.

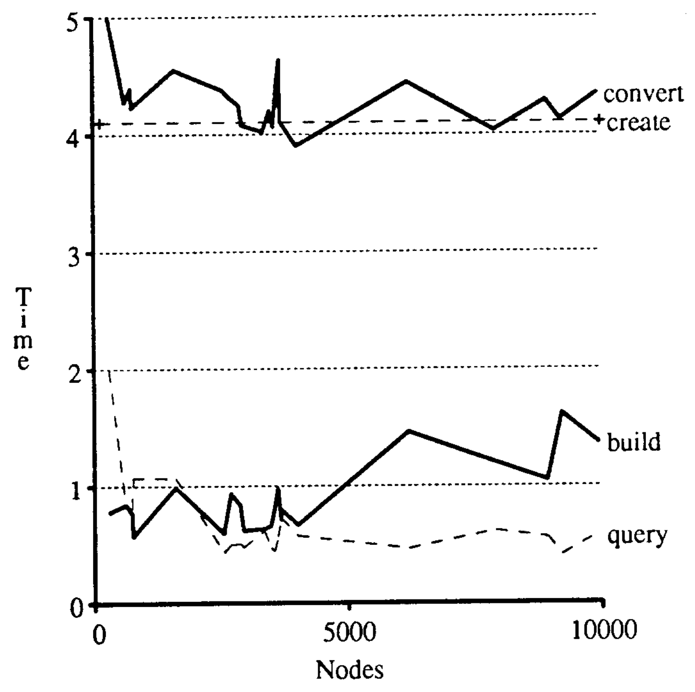


Figure 6.15: Cost per node to fetch the AST from the DBMS in the PPT schema.

program source results are similar and do not change our conclusions. We do not intend to isolate the effects of the different database dimensions here, that we do in subsequent sections. We look at the applicable dimensions here, granularity and data representation, but their impact is not as clear as in the sections where we isolate them. Because our AST designs do not vary in the number of relations, it is not discussed here.

The cost equations are computed by dividing the time in each phase by the number of elements at each measurement point. We measured the times spent in the three phases we mentioned above: DBMS query, data conversion, and data structure reconstruction. Figure 6.12 shows the curves we obtained for the time to fetch in terms of the number of nodes for the AST in the NPT schema, with a curve for each phase: query, convert, and build. The create curve is the AST creation time, included for comparison purposes. Figure 6.13 shows the cost per node curves for each phase, also for the NPT in the NPT schema. Figures 6.14 and 6.15 show the corresponding time and cost per node curves to fetch the AST in the PPT schema respectively. For the cost curves, we computed the linear regression, and list them in Table 6.3. As you can see from Figures 6.13 and 6.15, the costs per node listed in Table 6.3 are constant.

Table 6.4 shows the relative time distribution of fetching the AST from POSTGRES in the different phases for the NPT and PPT schemas in the ASCII and binary data representations. It was computed by using the linear regression cost per tuple for each phase, adding up the contribution of each phase to get the total storage cost per tuple, and finally computing the percentage contribution of each storage phase to the total. In addition, the *Total* column reports the ratio of the total cost per tuple to that of creating the AST from scratch (the *Create* row in the Table). Notice that we include the convert time for the NPT schema, even though the data is stored in binary format in the BE. This is because the BE-FE interface only handles ASCII values, therefore it returns the NPT tuple fields in ASCII format and we have to convert them to binary in the FE. We include the NPT ASCII representation in Table 6.4 as an extra point of comparison, to see the effect of data conversion on fine granularities.

6.8.2. Fetch Analysis

In Table 6.3 we can see that, for both NPT and PPT, convert is very expensive. It is fortunate that data conversion is not usually necessary for fine granularities, since data conversion doubles the already very expensive fetch times. NPT query is very expensive, while PPT query is cheap. The build costs are essentially the same.

From Table 6.4 we can see that fetching the AST from the DBMS can be cheaper than creating it (PPT bin), or up to an order of magnitude more expensive (NPT

Schema	Query	Convert	Build
NPT	17.0	19.0	0.8
PPT	0.6	4.3	1.0

Table 6.3: Cost per node (msec) of different phases to fetch the AST.

Representation	Query	Convert	Build	Total
Create	-	-	100%	1
NPT bin	93%	-	7%	4.3
NPT ASCII	46%	52%	2%	9.0
PPT bin	28%	-	72%	0.4
PPT ASCII	10%	73%	17%	1.4

Table 6.4: Relative time distribution to fetch the AST from POSTGRES.

ASCII). As we would expect, the binary representations are significantly more efficient than the ASCII representations, about 4 times cheaper for the PPT and 2 times cheaper for the NPT granularities. The culprit is the very expensive data conversion operation. The *Total* column shows the ratio of the corresponding representation to the time to create the data structure. We can see that NPT is expensive, an order of magnitude greater if in ASCII representation, and about 4 times greater if in binary representation. The PPT representations have total times very close to the Create time, with the ASCII representation 1.4 times larger than the time to Create, and PPT binary 0.4 times the time to Create. For NPT binary, query dominates overwhelmingly, with over 90% of the time. For the PPT representations the query component is not dominant, with build dominating in the binary case and convert in the ASCII case.

To summarize, compared to the time to create the AST, the coarse granularities perform quite well; data representation has a large impact, with binary representations being much faster; the fine binary granularity is significantly slower than Create, but its performance is acceptable. We never recommend using fine granularities in ASCII representation because of its awful performance and lack of purpose.

6.9. Space Considerations

In this section we discuss the space requirements and distributions of the different data representations both in-core and in the database. We look at the size of the data structures, in Kbytes, in several representations: internal binary, external binary, external ASCII, DB binary, and DB ASCII. In the section on Frontend space usage we cover the internal and external representations. In the section on DBMS space usage we cover the DB representations and how they compare with the frontend representations. As with the previous sections, we concentrate on the AST data structure, but the results obtained with the symbol table and source code are similar. With the source code there is a difference in magnitude, since the fine granularity *granule* size, the source line, is larger than that for the SYMTAB or AST (e.g., our data set averages 5 nodes per line).

6.9.1. Frontend Space Usage

In this section we consider the in-core size of the data structures, in Kbytes, in several representations: internal binary, external binary, and external ASCII. Internal ASCII does not make sense and is not used, therefore we do not include it. First we describe how we obtain our measurements. When in the FE, a data structure has

Data Representation	NPT		PPT		$\frac{NPT}{PPT}$
	Size	Ratio	Size	Ratio	
Internal Binary	11007	3.4	11007	6.1	-
External Binary	3281	1	1803	1	1.8
External ASCII	3281	1	2755	1.5	1.2
POSTGRES Binary	18833	5.7	2570	1.4	7.3
POSTGRES ASCII	18833	5.7	3870	2.1	4.9
INGRES Binary	8930	2.7	-	-	-

Table 6.5: AST space utilization data. Sizes in Kbytes.

several different space components:
i) Data structure overhead: The space consumed by the headers and other information associated with the data structures that *contain* the data. For example, on a record data structure we have an 8 byte header.
ii) Redundant data: The space used by data that is used for efficiency or convenience, like back pointers in a doubly linked list, but that does not add to the information content of the data structure.
iii) Schema overhead: The space used by data that is needed in the data repository (file or DBMS), but not in the FE. For example, we need a *file_name* field with each tuple to map it to the corresponding FE data structure.
iv) Net data: The size of the minimum set of data that contains all the information in the data structure. In other words, the minimum set of data from which the whole data structure can be regenerated.

We make explicit the distinction between the redundant and the net data because, in a Database context, the redundant data does not have to be stored in the DBMS. The schema overhead is part of the external representations, even when they are in the FE. We calculate the size of the internal representation as:

$$ND + RD + DO \qquad (6.17)$$

Where:
ND is net data,
RD is redundant data, and
DO is data structure overhead.
We calculate size of the external representations as:

$$ND + SO \qquad (6.18)$$

Where:
SO is schema overhead.

The schema overhead is included since it is the information needed to restore the original data structure. To adjust between an ASCII and a binary representation, we use the simple formula: ASCII is one byte per printing character, and we counted the characters of the printed representations of the binary data. The adjustment affects three components only: RD, ND, and SO. For example, the AST PPT external representation of each procedure is a list with 5 components per node. In binary form the 5 components use 17 bytes, and in ASCII form they use 26 bytes.

Table 6.5 shows the space utilization of the different representations for the AST in both the NPT and PPT schemas. The entries with a dash (-) are not applicable. For both NPT and PPT the size column is the net size of that representation in Kbytes, and the Ratio column is the ratio between the corresponding representation and the smallest representation, External Binary. The last column, NPT/PPT is the ratio of the corresponding NPT to PPT sizes for each row. For the internal binary representation the PPT schema does not make sense because when the data structure is in-core we need to manipulate it at fine granularities, but we repeat the same entry as under NPT so we can see how it compares with the external PPT representations. Therefore for internal binary the NPT/PPT entry does not apply.

The internal binary size is very large, 3 times greater than the external NPT binary and 6 times greater than the external PPT binary as we can see from the Ratio columns. By coincidence, the NPT binary and ASCII representations are the same size, while the PPT ASCII is 50% larger than the PPT binary. Finally, from the NPT/PPT column we can see that the NPT external representations are between 20 and 80% larger than their corresponding PPT representations. Apart from the ASCII to binary space difference, we need further information to explain the size differences. In order to do so we computed how the space is distributed among different components for each representation.

Table 6.6 shows how the space is distributed among the different components for each of the FE representations. Notice how, for the NPT internal representation, the data structure overhead occupies almost two thirds of the space, while the redundant data is a small contributor to the total (8%). In addition, the redundant data is also a small contributor (21% to the data component (redundant + net data). The external representations have virtually no data structure overhead since we extract the data, organize it, and put it in a large, low overhead list, for storage to files or the DBMS. They hold no redundant data since it is not needed in the BE or files. The redundant data is replaced by the schema overhead, which allows the data to be accessed quickly and to reconstruct the FE data structure from the BE.

At fine granularity, NPT, the schema overhead is about 1/3 of the total, while for the coarse granularity, PPT, it is only 2% to 3%. The reason is that the schema overhead accrues on a per tuple basis. NPT has 1 node per tuple, whereas PPT has (in our samples) about 50 nodes per tuple.

For the external representations, the space usage in files is the same as the FE space usage, since files do not add any additional overhead. If we use the internal binary representation, like the "Fasl Write" facility in Franz Inc's Allegro Common Lisp (Fasl is their proprietary internal binary format), we increase space utilization but eliminate the *extract* time when saving to the file and the *build* time when reading from the file. On the other hand, the extra size increases the number of I/O's. Given that I/O is so expensive, that the space difference can be as large as 3:1, and that both *extract* and *build* are relatively inexpensive (see Tables 6.1 and 6.3), we do not think that it is a good idea to store in internal binary format, either to files or the DBMS. Among the

Space Component	NPT Internal Binary	NPT External Binary	NPT External ASCII	PPT External Binary	PPT External ASCII
Net Data	30%	65%	68%	97%	98%
Schema Overhead	-	35%	32%	3%	2%
Data Struct. Overhead	62%	-	-	-	-
Redundant Data	8%	-	-	-	-

Table 6.6: AST relative space distribution in the frontend.

external formats, we can choose between the ASCII or binary representations. Using external binary is the best option in terms of space and efficiency since it minimizes I/O, since it is significantly smaller, and because it avoids the very expensive *conversion* phase. The external ASCII representation should only be used if it is required by a heterogeneous hardware or language CASE environment, if there is no support for binary data, or when implementation speed is more important than execution efficiency.

6.9.2. DBMS Space Usage

When the data is stored In the database, the relations that represent the data structure add the following space components:
i) Index overhead: The space occupied by the access methods and auxiliary relations to speed access to the data. In our environment, this overhead consists of a B+ tree index per relation.
ii) Tuple overhead: The space occupied by the tuple headers and other information associated with the relations that contain the data. For example, in POSTGRES v2.1 each tuple includes a 76 byte header, and each relation includes an entry in the *pg_relation* relation.
iii) Relation overhead: The space occupied by the information associated with the relations that contain the data. For example, in POSTGRES v2.1 each relation includes an entry in the *pg_relation* relation.

The relation overhead is negligible compared to the other space components, so we do not take it into account in our calculations. Using the same terminology as in

Representation	Tuples	Net Data	Relation Overhead	Tuple Overhead	Index Overhead	Total
POSTGRES NPT bin	105845	13%	6%	43%	38%	7.3
INGRES NPT bin	105845	25%	11%	36%	28%	3.6
PPT bin	1923	89%	1%	6%	4%	1
PPT ASCII	1923	92%	1%	4%	3%	1.5

Table 6.7: AST relative space distribution in the database.

equations (6.17) and (6.18), we calculate the size of the database representations as:

$$ND + SO + TO + IO \qquad (6.19)$$

Where
TO is tuple overhead, and
IO is index overhead.

Table 6.7 shows the absolute and relative sizes of the database representations of the AST. For comparison purposes we include the sizes in the POSTGRES DBMS, as well as in commercial INGRES. The NPT database representations are very large, with POSTGRES NPT 5.7 times larger than NPT external binary. The difference between POSTGRES and INGRES is quite dramatic, with POSTGRES twice as large. (As a coincidence, the NPT INGRES size is very close to the NPT internal binary size). For the coarse granularities we find that the PPT binary is 40% larger than PPT external binary, and PPT ASCII is also 40% larger than PPT external ASCII. The NPT representation is 5 times larger for ASCII and 7 times larger for binary than PPT. Since commercial INGRES has a small maximum record size, 2 Kbytes, we were unable to run the PPT experiments on it. The database to external and internal to external representation size differences are due to the same reason: a tradeoff of space for speed and functionality.

Table 6.7 shows the relative space distribution between the different components of the database representations. The entries column shows the number of tuples for each representation, while the total column shows the ratio of the respective representation to the smallest DB representation, PPT binary. The database overhead (IO + TO) is quite sizable for the fine granularity representations, 80% for POSTGRES and 64% for INGRES, while it is pretty small for the PPT representations (7 to 10%). The reason is, again, the large number of tuples for fine granularities. The INGRES to POSTGRES difference is substantial (POSTGRES is twice as large), but they both exhibit the same small granularity results: large size and domination of the DB overhead. There is a much higher amount of space overhead in the fine granularity representations, leading to large overall size differences. This fact, together with the observed speed advantage of large granularities, makes coarse granularity representations very attractive, especially when speed is important and/or space is limited. Under these circumstances, one of the intra-object reference schemas that we propose in Chapter 5 may compensate for the loss of granularity.

6.10. Granularity Effects

To isolate the effects of granularity on performance and space utilization we designed an experiment with the program source and the AST. We use the *c-samples2* data set (described in Section 6.4 on the experimental environment), chosen to provide sample points with the same number of elements that are divided into sets of different sizes. For example, with 1000 AST nodes per file, we have a file that contains 10 identical procedures which yields a data point at 100 nodes per procedure, and another file that contains 25 identical procedures which yields a data point at 40 nodes per procedure. The cost equations are for elements per procedure. The results are illustrated in Figures 6.16 and 6.17 for the source code, and Figures 6.18 and 6.19 for the AST. In the Figures granularity increases from left to right. For the AST, the NPT granularity is illustrated by the costs at the 1 (one) node per procedure data point. For the source

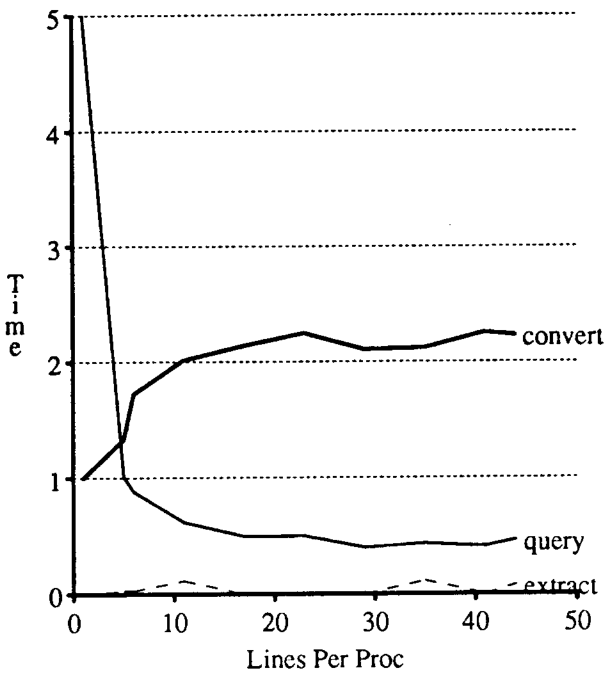


Figure 6.16: Cost per line to store the source code to the DBMS in the PPT schema.

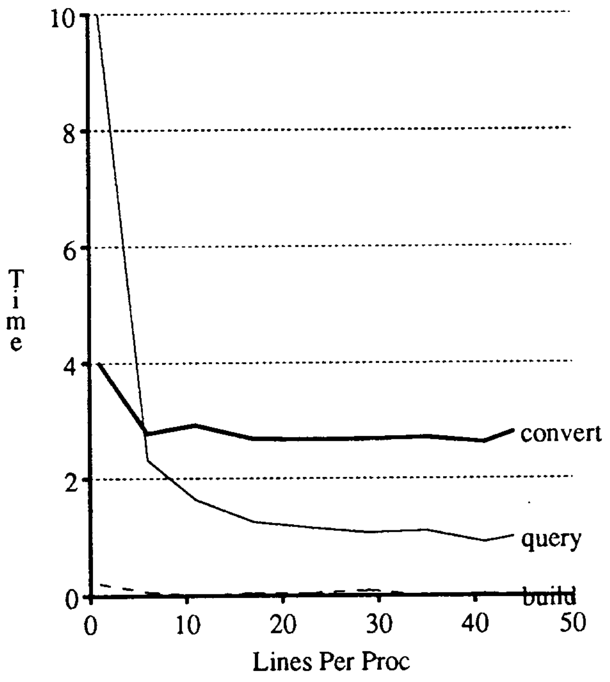


Figure 6.17: Cost per line to fetch the source code from the DBMS in the PPT schema.

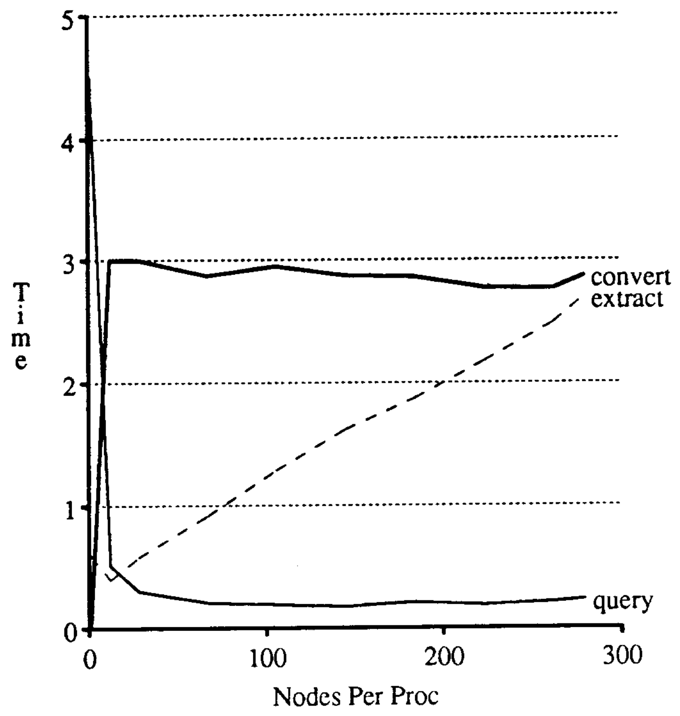


Figure 6.18: Cost per node to store the AST to the DBMS in the PPT schema.

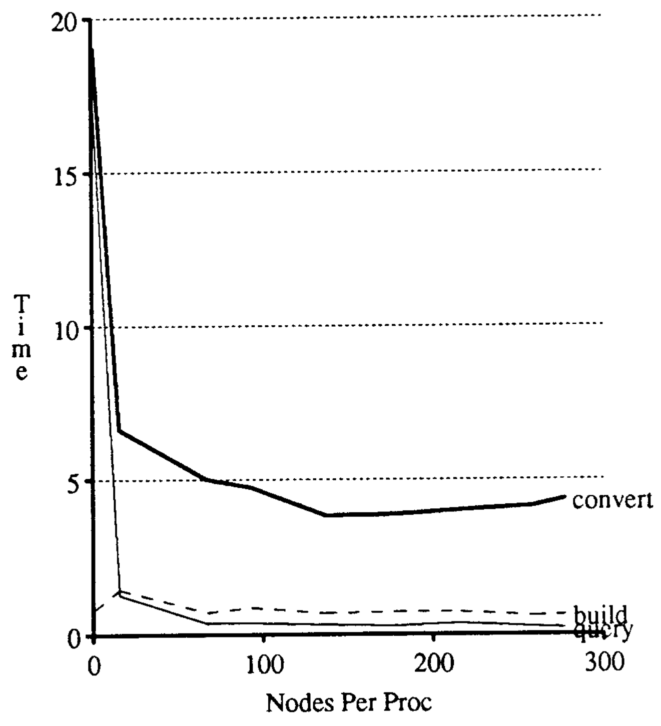


Figure 6.19: Cost per node to fetch the AST from the DBMS in the PPT schema.

code, the LPT granularity is illustrated by the costs at the 1 (one) line per procedure data point. As we mentioned before, POSTGRES v2.1 limits the object size, so we could not measure the MPT granularities. Nevertheless, we discuss the MPT granularity later on in this section. The absolute values tend to be larger for the source code (e.g., for query in the fetch operation the source code converges on 1 msec per line and the AST converges on 0.2 msecs per node) because our data has, on average, 6 nodes per text line and the figures show the cost per node for the AST and cost per line for the source code.

For the source code representation, the convert cost measures some pre- and post-processing that is needed to substitute characters that POSTGRES cannot handle. This is different from the convert cost for the AST and SYMTAB, which measure the actual conversion between binary and ASCII representations.

The effect of granularity on query cost is the same for both the store and fetch operations, and for both the text and the AST representations. It asymptotically converges, in a very monotonically decreasing fashion, on a constant cost per element. Query costs are very high at the NPT and LPT granularities, rapidly decrease at small granularity increases, but the rate of improvement also quickly levels off. The point where the cost per element gains are no longer significant is at 100 nodes per procedure for the AST, and at 25 lines per procedure for the text. The reason can be traced back to equation (6.11), which states that query time is a function of the number of tuples and the number of disk pages. Both decrease as granularity increases. But, once the number of nodes per procedure reaches 100, and the number of lines per procedure 25, the rate of change of the number of pages and tuples slows and is no longer significant (for example in going from 12 to 72 nodes per procedure we get a 6:1 tuple ratio and a 1.2:1 page ratio, and going from 72 to 122 nodes per procedure we get a 2.5:1 tuple ratio and a 1.04:1 page ratio).

We have a similar situation, asymptotic convergence on a constant point in a monotonically decreasing fashion, for the convert and the build costs per element during the fetch operation, For convert the cutoff point is 150 nodes per procedure, and it behaves like this because the data conversion algorithm is linear on the number of nodes and the number of procedures, but is affected very little by the procedure size. It is dominated by the number of nodes, not by how the nodes are grouped. For build, the cutoff point is at 70 nodes per procedure, and it behaves like this because its algorithm is linear on the number of nodes only, and the higher cost at finer granularities are due to high constant costs (the b term in equation (6.10)).

The convert cost, during the store operation, also asymptotically converges for both representations, but the AST does it from above and the text from below. The AST convert algorithm is the same algorithm used for data conversion during fetch. Convert for the source code behaves differently because, as we mentioned above, it measures a different thing: character substitution to get around the POSTGRES string interface. The string substitution algorithm is directly proportional to the number of characters, and, by coincidence, the average number of characters per line has exactly the same shape of the convert curve. If the lines are all the same size (in characters), the text convert cost is constant at all granularities.

During the storage operation, data extraction is negligible for the source, but the AST extract curve is monotonically increasing, with an equation Y=0.007X+0.6. The reason is that our node extraction algorithm is quadratic $O(n^2)$ on the number of nodes in the procedure. Using a linear algorithm, easy to implement, yields a constant data extraction value close to the NPT extraction value of 0.6 msecs per node. Using a quadratic algorithm was a mistake on our part which we could have corrected in time to include in this dissertation. We did not, on purpose, to make a very important point. The asymptotic behavior of the algorithms used in each phase is very important, and should be carefully evaluated using representative samples of the data they will handle. Otherwise, a relatively small cost (data extraction as opposed to data conversion) may become the dominant factor in practice, as extraction would be on procedure sizes larger than 300 nodes (equivalent to 50 lines). Our *c-procs1* data set had many procedures of larger size.

As mentioned above, we did not perform any experiments on the module per tuple (MPT) granularities due to POSTGRES v2.1 limitations. But, the MPT schema is simply an increase in granularity. And, as you can see from the results in our granularity experiments, the cost per element equations become constant at relatively small granularities, well under the size of most procedures. Therefore, scaling up to the MPT granularity would not significantly improve either space utilization or operation speed.

6.11. Data Representation Effects

In this section we isolate the effects of data representation on both space and efficiency. As we mentioned before, changing data representations does not make sense for fine granularities, so we concentrate on the PPT schema of the SYMTAB attributes, with ASCII and independent binary representations. Table 6.8 shows the costs to store and fetch the PPT attributes. We can see that the query cost in both the fetch and store operations is relatively small, and that the data conversion cost is very significant. These two observations are corroborated by the AST PPT data (Tables 6.2 and 6.4).

Data representation affects time efficiency at the conversion phase, and at the query phase. Its impact on query costs comes directly from the different sizes, since we have the same number of tuples but a different number of disk pages (equation 6.2 shows the query cost). The ASCII and binary size differences are also relatively small (ASCII is 25% larger for the PPT attributes, 50% larger for the PPT AST). Therefore the impact on query cost is not significant. On the other hand, the performance impact at the conversion phase is major. As we can see from Table 6.8, the conversion phase doubles the storage cost (50% of total), and triples the fetch cost (70% of total) for the attributes. This is similar to what happens to the AST PPT.

To summarize, the space difference between ASCII and binary representations ranges from 25% to 50%, while the cost differences are mainly due to the conversion

Operation	Extract	Build	Convert	Query	Binary	ASCII
Store	3.8	-	3.3	0.8	4.6	7.9
Fetch	-	0.8	3.5	0.5	1.3	4.8

Table 6.8: Costs to store and fetch the attributes in the PPT schema.

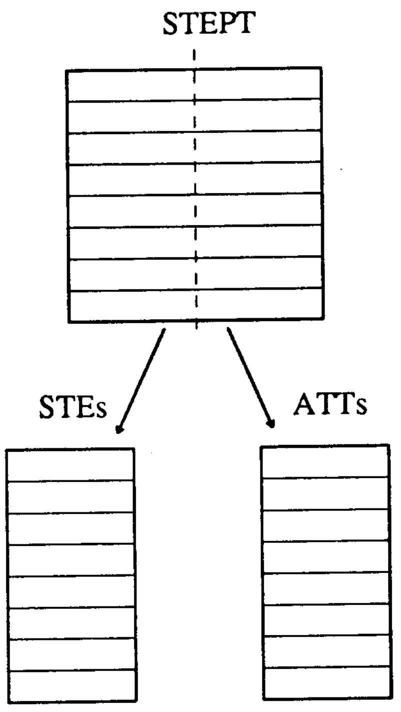


Figure 6.20: Vertical division of one relation into two.

overhead and range from doubling the storage times to tripling the fetch costs. Given that it costs more in both time and space to store the data in ASCII format, the reasons to use ASCII must come from functionality and convenience. In fact, they do. The ASCII format supports heterogeneous hardware and software environments, and it is usually easy and fast to implement and support.

6.12. Number of Relations Effects

In this section we isolate the effects of changing the number of relations on both space and efficiency. There are two main ways in which a single relation design can be divided into several relations, horizontally or vertically. As an example, we use the simple case of dividing one relation into two. In *vertical* division, illustrated in Figure 6.20, we divide the relation fields into two sets, create one new relation for each set, and insert the corresponding fields of each tuple into the new relations. We end with two relations, each with the same number of tuples as the original one. In *horizontal* division, illustrated in Figure 6.21, we divide the tuples into two groups, create a new relation per group, and insert each group into its corresponding relation. We end up with two relations, each with a fraction of the tuples in the original relation (both fractions add up to the original number). We now provide an extended example.

Suppose our symbol table has 1000 entries (STEs) and 2000 attributes (ATTs). We have two attribute kinds, 1000 of kind type (Type_Atts) and 1000 of kind size

(Size_Atts). The single relation schema, SPT, has 1000 tuples. Each tuple contains an STE, a Type_Att, and a Size_Att. In addition, the SPT relation has one 1000 entry index. To obtain the APT schema, we use vertical division, producing an STE relation and an ATT relation as illustrated in Figure 6.20. The result is an STE relation with 1000 STE entries and a 1000 tuple index, plus an ATT relation with 2000 ATT entries and a 2000 tuple index. To further obtain the R-APT schema (relation per attribute type), we do horizontal division of the ATT relation, producing a Type_Att relation with 1000 entries and a 1000 entry index, and a Size_Att relation with 1000 entries and a 1000 tuple index as illustrated in Figure 6.21.

Given the relatively small contribution of the data to space consumption, as illustrated by the NPT binary entries in Table 6.7, at fine granularities the amount of space and time efficiency of the chosen schema are mostly determined by the total number of tuples in a linear relationship. With this in mind, vertical division produces a decrease in space and time efficiencies that is linear with the increase in the number of records. Horizontal division should not increase space consumption significantly since it does not increase the total number of records, but should adversely impact performance when it produces a significantly larger total number of relations. To see why, we can consider how we might access the information in the relations. Either we use multiple queries, or a single complex query. If we use multiple queries, the per query overhead

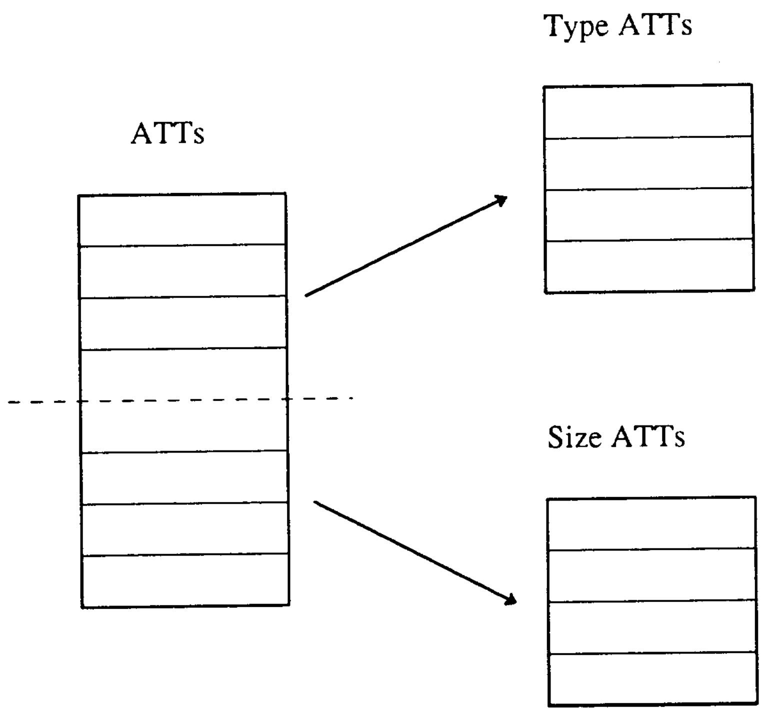


Figure 6.21: Horizontal division of one relation into two.

becomes much more significant and impacts performance. If we use a single, multi-relation query the database overhead for it (query plan selection in particular) is either very expensive in time, or will produce a sub-optimal plan otherwise, in both cases impacting performance.

For our experiments we looked at vertical division, using the symbol table program data structure, and comparing a single relation schema, SPT, with the APT two relation schema (S-APT stores the symbol table entries, A-APT stores the attributes). For both the fetch and store time comparisons, we use a two query approach with the APT schema. This is the only choice available when storing since there is no multi-relation append commands in POSTGRES (or any other DBMS that we know of). To fetch data from two relations we can use two single relation queries, or a two way join. We use the two relation approach. We have an average of 1.34 attributes per symbol table entry, therefore to compare the per phase costs we compute the total APT cost as:

$$S\text{-}APTcost + 1.34 * A\text{-}APTcost \qquad (6.20)$$

That is, each SPT tuple is equivalent to 1 S-APT tuple plus 1.34 A-APT tuples. Table 6.9 shows the costs of the different phases to store the symbol table to the DBMS. Table 6.10 shows the costs of the different phases to fetch from the DBMS. The rows labeled **S+A APT** have the total two query APT cost, computed as per equation (6.20). As with the NPT data above, we include data conversion with the fetch phases as another point of comparison.

For both the APT and SPT schemas in store and fetch, the query phase dominates the cost. The main difference between the two schemas is that the query time doubles for the two relation case, and with the query phase dominating the total cost, the two relation schema is almost two times slower than the single relation. The relative impact

Representation	Extract	Query
A APT	1.0	4.5
S APT	1.1	4.8
S+A APT	2.4	10.8
SPT	3.0	5.5

Table 6.9: Costs to store the APT and SPT schemas.

Representation	Build	Convert	Query
A APT	0.3	8.0	16.0
S APT	0.6	18.0	18.0
S+A APT	1.0	28.7	39.4
SPT	4.0	20.0	19.5

Table 6.10: Costs to fetch the APT and SPT schemas.

Representation	Entries	Net Data	Schema Overhead	Tuple Overhead	Index Overhead	Size (Kbytes)
APT	8334	16%	5%	34%	45%	1843
SPT	3566	22%	3%	25%	50%	1073

Table 6.11: DB size and relative space distribution for the APT and SPT schemas.

of data extraction is very small during fetch because of the high query cost (due to many small tuples and large database size). The SPT build cost is 4 times larger than for APT because we need to check each STE for all possible attribute kinds. During storage, the cost to extract is almost the same. As with the NPT data, if we do data conversion it is very expensive. This confirms our conclusion that data conversion is impractical for fine granularities.

Table 6.11 shows the size of the SPT and APT databases in the Size column. Notice that the APT size is 1.7 times the SPT size. Both are very large for the small data set that we use in these experiments. The space is similar in both cases, dominated by the index and tuple overheads just as for the NPT binary representation in Table 6.7. As we would expect, the net data is a relatively larger part of the SPT schema, and the schema overhead is also relatively smaller.

In coarse granularity schemas we expect the same overall effects as with fine granularity, but in much smaller proportion since both the number of records and total space utilization are smaller. Therefore, for coarse granularities, we expect a practically nil impact of horizontal division, and a relatively smaller impact of vertical division (as noted above, the impact is directly proportional to the total number of relations).

To summarize, when using vertical division, a two relation schema slows down and grows in linear proportion to the added number of tuples it creates compared to the single relation schema. Therefore, unless there is a compelling functionality reason to vertically divide the data in multiple relations, we recommend against it. Dividing horizontally has small performance and space implications when the division does not involve many new relations, and therefore other considerations must be used when considering it. If horizontal division produces a large number of relations, performance suffers significantly.

6.13. Comparative DBMS Performance

In this section we compare POSTGRES v2.1 performance with that of a wide variety of systems on two standard DBMS benchmarks. We use the results reported in [28]. Our objectives are several. First we want to compare the performance of POSTGRES with other DBMS's that use similar algorithms to see how fine tuning might improve our results. We also compare POSTGRES with systems that use different algorithms, to see how that would affect our results. In addition we look at the performance range for different kinds of queries (e.g., select and join). Finally we also look at the differences between *cold* and *warm* queries, and between queries that are run in a *local* or *remote* server. The first comparison is between POSTGRES, the

Query	Meaning	POSTGRES	UCB-INGRES
2	select 10% into temp, no index	9.8	10.2
3	select 1% into temp, clust. index	0.7	5.2
5	select 1% into temp, non-clust. index	1.2	5.3
6	select 10% into temp, non-clust. index	4.0	8.9
7	select 1 to screen, clust. index	0.3	0.9
9	joinAselB, no index	12.6	35.3
10	joinABprime, no index	17.0	35.3
11	joinCselAselB, no index	25.9	53.7
14	joinCselAselB, clust. index	24.1	56.7
17	joinCselAselB, non-clust. index	35.2	68.7
18	project 1% into temp	18.5	36.7

Table 6.12: A Comparison of UCB-INGRES and POSTGRES on the Wisconsin Benchmark. (Times in seconds. Reproduced from Table 2 in [28]).

University version of INGRES, and the commercial implementation of INGRES v5.0. They are compared on the Wisconsin benchmark [89], which consists of a comprehensive set of queries that can be used for systematic benchmarking of relational database systems. Table 6.12, reproduced from Table 2 of [28], reports a subset of the results with University INGRES and POSTGRES. They show that POSTGRES is about twice as fast as University INGRES. In addition, the authors report that POSTGRES is about 0.6 times as fast as commercial INGRES on the same benchmark (but on a different system configuration). In the Table the query times range from 0.3 to 35 seconds for POSTGRES, which shows the performance limitations of a relational DBMS. The performance increase with commercial INGRES still leaves it in the second to tenths of second performance range. In addition, queries 2 through 7 show how query performance improves as we move from no index, to a non-clustered index, to a clustered index. Further, we can also observe that more complex queries like joins and projections, queries 9 to 18, are very expensive.

The second benchmark compares POSTGRES with several other kinds of systems, including the Cattell in-house system, a commercial OODB, and a commercial relational DBMS. The Cattell/Skeen benchmark [90] is designed to measure the performance of engineering operations in a manner that reflects the access patterns of engineering applications (like CASE). Table 6.13, reproduced from Table 3 in [28], reports the results for the fine granularity version of the benchmark. The POSTGRES numbers are adjusted to compensate for speed differences between the different disks. The Table reports on *local* (client and server on same machine), and *remote* (client and server on different machines), and on *cold* (database on disk) and *warm* (relevant database pages in cache) queries. In addition POSTGRES uses database procedures to make direct access method calls and calls the procedures via the Fast Path feature to obtain maximum performance.

The very large tuple header size in POSTGRES led to very large database sizes, and its buffer pool size had to be increased to 13 Mbytes (the other systems use a 5 Mbyte cache) to be competitive with the other systems. The fast insert times of the

OODBMS are due to the fact that it writes different record types on the same page, therefore causing fewer I/O's. The OODBMS also does very well on *warm* queries because it caches records in main memory format, therefore avoiding data conversion between its buffer and the FE.

The cold-warm numbers for both POSTGRES and the RDBMS are virtually identical, showing that the network overhead is insignificant in a single client situation. The *cold* times are dominated by the time to get the data from the database disk into the DBMS buffer pool. The *warm* numbers are dominated by CPU processing times of the data already in memory, data conversions (if any), and inter-process communication (IPC). Since the OODBMS and in-house systems share memory space with the application their warm numbers incur no IPC overhead.

The in-house and OODBMS systems show a significant speedup between cold-local and cold-remote operations, while the RDBMS and POSTGRES do not. This is due to the nature of the benchmark and implementation technologies. The subset of the Cattell/Skeen benchmark reported measures the retrieval of an n-level tree. The in-house and OODBMS implement pointer chasing. That is, whenever the FE encounters a pointer, if the "pointed object" is not in memory, the FE fetches it from the BE. With a deep tree with many children, cold-fetching implies many FE-BE roundtrips. The network overhead then becomes significant. The same operation is achieved with a single query on a secondary key for both POSTGRES and the RDBMS, and since we get a single FE-BE roundtrip, the network overhead remains very small. For both POSTGRES and the relational system compared, the difference between warm-local and warm-remote, and cold-local and cold-remote are negligible. The reason is that the network transmission times are two orders of magnitude smaller than the DBMS and FE times.

The fastest numbers in this table are on the order of 1 second. Other commercial systems report *warm* numbers as low as 0.1 secs in the same benchmark [34,91]. From this data it is easy to infer that using a DBMS as our data manager imposes a lower limit on the performance of any query of about 0.1 secs on even the most highly tuned

Query	in-house	OODB	RDBMS	POSTGRES
cold-remote-lookup	7.6	20.0	29	17.5
cold-remote-traversal	17.0	17.0	90	36.8
cold-remote-insert	8.2	3.6	20	7.3
warm-remote-lookup	2.4	1.0	19	8.4
warm-remote-traversal	8.4	1.2	84	26.8
warm-remote-insert	7.5	2.9	20	4.5
cold-local-lookup	5.4	13.0	27	17.4
cold-local-traversal	13.0	9.8	90	36.7
cold-local-insert	7.4	1.5	22	7.3

Table 6.13: A Comparison of several systems on the Cattell/Skeen Benchmark. (Times in seconds. Reproduced from Table 3 in [28]).

data manager. This rules out using the data manager to fetch, store, update, or delete individual AST nodes or SYMTAB entries. The correct model of interaction between the data manager and the CASE tools is, apart from interrogation, at the procedure or module level. This does not mean that we rule out fine granularity designs, but rather that if we do have fine granularities in the DBMS, the FE should fetch and store from these designs related sets of tuples so that the high costs are amortized between many elements.

6.14. Differences Between Data Structure Classes

In Chapter 5 we classified CASE data structures into five classes according to the kind of data that they contain: textual, tabular, graphical, unstructured, and sequential binary. An objective of our experiments was to see if there were fundamental differences between the classes in a database context. We find one fundamental difference between the classes with internal structure and the unstructured class. For data structures in the textual, tabular, graphical, and binary sequential classes, the database designs should follow the following guidelines:

- When choosing the number of relations in the design, minimize the total number of tuples.
- Choose a data representation that minimizes the impedance between the FE and BE representations.
- Medium granularity designs optimize space and performance, but may produce a loss of functionality.

For the data structures that contain unstructured data, like pictures, the recommendation is to store them at the object per tuple granularity, in a single relation, and in their original data representation. For these data structures, the DBMS needs to provide special large object support like binary data, unlimited object size, and a special "file like" interface to the frontend.

6.15. Summary

In this section we summarize the major results of our experiments.

- The major determinant of both space usage and efficiency is the number of tuples in the chosen schema. The reason why space utilization goes up with number of tuples is the per tuple database and index overheads. The reason why speed diminishes with the number of tuples is that query cost is a function of number of records and number of pages, both of which are inversely related to granularity.
- For efficient database storage, multiple append is essential.
- Data conversion between binary and ASCII is extremely expensive, dominating other costs at large granularities and being just as expensive as querying at fine granularities.
- Data storage and fetching have well defined, distinct phases. The algorithms used at each phase need to be evaluated independently.
- Space usage in the FE carries a large data structure overhead.
- When considering the number of relations in a database schema, the greatest impact on space usage and performance comes from the total number of tuples in each alternate design. Minimizing the number of relations leads to less complexity and space overhead, and has a significant performance impact when the number of

relations diminishes significantly.

- Increasing granularity saves significant amounts of space and time up to a point, after which the gains become very small.
- The database space effects of different data representations are small because the tuple and index overhead does not change.

In general, the results are in accordance with the predictions that we made in Section 6.2. As far as granularity is concerned, the main difference is that there is a well defined point above which increases in granularity do not yield significant speed or space efficiencies. In practical terms for CASE, PPT granularity yields all the savings, and anything above that has practically no impact. As far as number of relations, the main difference with our predictions is that the number of relations has different effects depending on whether the tuples are divided horizontally or vertically. In addition, beyond the number of tuples, the number of relations adversely affects performance when the difference (in numbers) between designs is large (e.g., a 2 relation design versus a 10 relation designs). Our findings regarding the data representation are in accordance with our predictions.

An important point is that the database storage and fetching operations are independent of the algorithmic complexity of creating the data structure. In our experiments we looked at data structures with linear, **O(n)**, algorithmic complexity creation times. The more expensive the algorithmic complexity of creating a data structure, the more worthwhile it is to store it in the DBMS from the point of view of performance.

The main recommendation is to use a large granularity design with an independent binary data representation, unless fine granularity access is necessary. Even when we have a fine granularity database design, the FE should interact with the DBMS at coarse granularity levels by fetching and storing sets of tuples (procedures or modules) per each operation. When deciding on the number of relations in the database design, minimize the total number of tuples to optimize space usage and performance. In addition, a design with a large number of relations for related data will be slower than one with few. When deciding on a data representation in the database, use ASCII only when convenience (e.g., ease of implementation) or environment considerations (e.g., heterogeneous language or hardware architectures) are more important than performance.

CHAPTER 7

EXAMPLE TULUM APPLICATIONS

7.1. Introduction

In previous chapters we have spoken of CASE environments, how they have complex requirements, and the need for effective data management to help satisfy those requirements. In this chapter we show that the features that satisfy the CASE environment requirements can provide the kinds of sophisticated services required of the next generation of CASE environments. As with a programming language, the power to implement the sophisticated applications is there. Now we need to determine what we want to do with this power, and implement it. We show how to use some of the advanced features in TULUM, including the ADT capability, historical data, the rules system, and database procedures. We use both coarse and fine granularities in our examples to illustrate that a DBMS can provide very important services to both. The applications we illustrate are very practical: version control, support for software reuse, and support for derived data and multiple views of the same data. As with TULUM these are designs, not actual implementations.

7.2. CASE Version Control

In this section we briefly introduce the RCS [66] (Revision Control System) version control model, and then show how this model can be implemented for the program objects. In RCS, the object granularity is the file [66]. Files are manipulated to provide a version tree, write locking, to associate comments with versions, and to merge different versions. Among the most important RCS commands are:

- *check-in (ci)*: deposits versions into RCS objects, prompting for a log message summarizing the changes from the previous version.
- *check-out (co)*: retrieves a copy of the specified version.
- *merge*: merges different branches in the version tree into a new branch.

In addition there are commands to list the state of all RCS files in a directory, the log changes to those files, and facilities to tag an object with descriptive information like the author and the date created. Symbolic names can be assigned to different revisions.

7.2.1. Problems with DBMS Version Control

Version support in next generation DBMS's falls short in the following respects:

- Naming: A version of a relation has a new name

```
create version foo from bar
```

- Identity: A version of a relation has a different relation-id
- Granularity: Versions are provided at the relation, not tuple level.
- Modification: Unlike the RCS model, versions of relations can be modified at any time (with RCS, only the leaves of the version tree can be modified. To modify a non-leaf version, we create a new version from it).

- Derived Data: There is no direct support for creating new versions of derived data when a version of the original data is created. It is not even clear what should be done to the derived representations when we create a new version (eg, what to do to the symbol table when we create a new version of a procedure).

The next section proposes a design to provide the RCS model of version control that solves some of these problems.

7.2.2. Version Control Schema

In this section we provide a sample schema that provides the foundation for the RCS version control mechanism. The schema makes use of historical data, but not the POSTGRES versions of relations since they do not provide the required flexibility. We use PPT granularity to make two points: that TULUM can provide sophisticated services even at coarse granularities, and that we improve over RCS, which works on the file granularity. The schema, illustrated with the procs relation, is the following:

procs (oid, name, src)
procs_v (v_name, vid, abstime, log, lock)
procs_w (v_name, vid, abstime)

The relation procs_v records the relation history, and the relation procs_w allows versions "checked-out" with write permission to be updated. The fields are:

- *procs_v.abstime* is check-in time
- *procs_w.abstime* is last modification time
- *v_name* is an ADT that records the version number and optional symbolic name. It defines operations *vname* and *vnumber*
- *vid* is the oid of the first version
- *log* is a message
- *lock* is to enforce the single "check-out for update" rule

We also provide several database procedures, and register them as the new operators *next-version, previous-version, and last-version* This schema solves the problems of identity [83], naming, and modification of past versions. Figure 7.1 shows an example of how the relations are used. The version tree for procedure *bar* has versions 1.0, 1.1, 1.2, and 2.1 as the trunk and 3 branches: (1.1.1.1, 1.1.1.2), (1.1.2.1), and (1.2.1.1). The version tree for procedure *foo* has versions 1.0, 1.1, and 2.1, with both 1.1 and 2.1 checked out with write locks as the entries on procs_w indicate.

7.2.3. Configuration Management

In broad terms, a configuration is a set of versions of different modules. To support configurations with TULUM, we can have a configuration relation with the following fields:

config (name, number, date, modules)

where config.modules is an ADT that contains a list of modules and their version names

((mod_0 vname_0) ... (mod_i vname_i) ... (mod_n vname_n))

and with operations like *list_all(), mod_names(config), vname(mod), insert(config, mod, vname), and delete(config, mod, vname).* Although beyond the scope of this work,

procs_w		
vid	vname	abstime
123	2.1	19
123	1.1	37

procs_v				
vid	vname	abstime	log	lock
123	1.0	0		
456	1.0	2		
456	1.1	3		
123	1.1	5		Joe
123	2.1	7		Luis
456	1.2	10		
456	1.1.1.1	12		
456	1.1.2.1	13		
456	2.1	15		
456	1.1.1.2	20		
456	1.2.1.1	23		

procs		
oid	name	src
123	foo	procedure foo () .....
456	bar	procedure bar (x,y) .....

Figure 7.1: Example version trees for procedures "foo" and "bar"

TULUM can also provide support for the more sophisticated RCS and configuration functionality.

7.3. Software Reuse

As software systems continue to get larger, more complex, and to have ever more stringent requirements, the reuse of well tested, optimized subsystems becomes a necessity. Reuse has many advantages, from saving time and resources to fewer bugs and more confidence in the correctness of the components reused. In this section we show how we can use the TULUM repository to serve as a *library* of software that supports intelligent and efficient access, retrieval, and insertion. Since it is highly unlikely that users would want to reuse at a smaller granularity than procedures, we use PPT granularity.

A compelling case for reuse at all phases of the life-cycle is made in Genesis [92], where the authors identify two main ways of reusing software documents: 1) reusable building blocks like libraries and abstract data types, and 2) reusable patterns like application generators and transformation systems. They, as well as us, focus on the first way, in which the reusable components are *passive* and are used through composition. They also identify the four areas that a system needs to support to incorporate reusability into the life-cycle:

- Retrievability: Being able to locate desired components.
- Composability: Ways of combining reusable components to obtain bigger and more specialized components.
- Understandability: Components need to be understood and (possibly) modified to meet the desired needs.
- Development Methodology: Software developers need to develop new components with an eye to making them reusable.

In the rest of this section we show how the TULUM software repository can effectively support passive component reuse in the above four areas. The approach taken in Genesis is to provide support through tools and techniques independent of each other and the data repository. Our approach is superior because it: i) combines all the functionality in a single, central place, ii) it makes the support available throughout the software life cycle, iii) the query, rule, and procedure facilities provide considerably more power and generality than the individual tools (e.g., the range of queries that InfoView [6] provides is a small subset of the queries possible with POSTGRES).

Our schema has different classes (relations) for requirement specifications, functional specifications, and source code. We make them separate classes because there is a many-to-one relationship between requirement specifications and functional specifications, as well as between functional specifications and source code. The classes are defined as follows:

```
req_specs (func, perf, candidates)
func_specs (func, type, inputs, rets, candidates)
src_proc (name, language, module, src)
```

Where func is *functionality,* inputs is a list of parameters, perf is *performance,* rets is the type of the value returned, and src is the *source code.* The field *candidates* returns a list of candidate procedures that satisfy the values of the other fields[18] In the case of req_specs, all candidates are entries in the func_specs relation, while in the case of func_specs all candidates are entries in the src_proc relation. To show how TULUM supports retrieval operations we will consider the case of a square root procedure. The following query returns a set of candidate functional specifications:

```
retrieve (req_specs.candidates) where
      req_specs.func = Square_Root and
      req_specs.perf = Highest
```

Or, if we know what functional specifications we want and want to obtain a set of functions that meets them:

```
retrieve (func_specs.candidates) where
      func_specs.type = Proc and
      func_specs.inputs = (float) and
```

```
        func_specs.returns = (float) and
        func_specs.func = Square_Root
```

Composability is supported through the use of procedures. For example, an abstract data type (ADT) consists of a set of procedures and data structures, e.g.:

```
    ADT (name, procs, data_structs, no_lines, use_history)
```

Where no_lines returns the number of source lines and use_history contains the comments of users who have used the ADT before. A procedure for a stack ADT called stack_procs that is executed when accessing ADT.procs would consist of the following queries:

```
    retrieve (src_proc.src) where
        src_proc.name = "push"
    retrieve (src_proc.src) where
        src_proc.name = "pop"
    retrieve (src_proc.src) where
        src_proc.name = "new-stack"
```

That is, the TULUM repository allows reuse at the smallest "global entity" level and also at the program, module, library, or ADT levels. The support for understandability is most useful for large components, like whole programs. If a user wants to customize a program, he will need to understand it well. For example, what are the procedures that might be affected by a change to the number of arguments to the procedure Display? The procedures that call the Display procedure in the program Xpaint are obtained from the calls relation:

```
    calls (program, proc, called_by)
```

by the query:

```
    retrieve (calls.called_by) where
        calls.program = "Xpaint" and
        calls.proc  = "Display"
```

The area where the support of TULUM is not as significant is in the development methodology, since this area depends for the most part on programmer and management commitment to reusability considerations throughout the life-cycle. For example, when the system is being designed it is important to be aware of how a particular design choice will affect the amount of reusability, and when a decision has been made to produce code from scratch, the programmers need to write it with an eye on future reuse. Nevertheless, the repository can provide help through reusability metrics and by providing a history of previous uses of components. For example, to decide whether to develop a new hash-table ADT as opposed to modifying an existing one to fit the project requirements we could use:

```
    retrieve (ADT.no_lines, ADT.use_history) where
            ADT.name = "Hash Table"
```

[18] The implementation can be through a tuple-id list, a database procedure, or a rule.

7.4. Derived Data and Multiple Views

Given that the source code is the "original" source of information about a program, the attributed AST and the program interrogation information are both "derived" from it. Keeping the derived data up to date with the original information is a service that TULUM can perform. In addition, TULUM needs to fully support multiple views of the same data in a way that is transparent to the CASE tools and that provides the performance they require. This would allow us to keep a representation optimized for access by a compiler (an AST) as well as one for human consumption, while knowing that they will always be consistent. TULUM includes several mechanisms to provide such support: by using rules, views, procedures, or keeping the *alternate* representations materialized. By having more than one representation of the same data we get the problems of replicated data [93] in addition to the problems associated with the transformation between representations, including consistency and space utilization. The alternatives include:

- Keep both representations materialized by defining an ADT for the derived representation, and each time an update is made to any object on which it depends, update the ADT field. The update can be done through the use of rules, or performed manually by the application. The advantage of doing it manually may be efficiency (incremental update vs complete recalculation).
- Make the derived data a database procedure which, when accessed, derives and returns the required information.
- Make the derived representation a view, which means that the system automatically rederives it every time it is accessed.

In the rest of this section we give an example of a simple solution for two database designs of a symbol table using the materialized solution: keep the multiple representations as materialized objects, with rules that detect when a representation has been modified and trigger the appropriate action, while with database procedures we incorporate into the data manager the algorithms to convert from one database design to another[19] The designs we choose are fine granularity and coarse granularity respectively, to illustrate that TULUM can provide powerful support that spans granule size. The repository thus takes care of converting and keeping all designs up to date automatically. The relations are ept_symtab (entry per tuple symbol table) and ppt_symtab (procedure per tuple symbol table)

ept_symtab (proc, value, type, oid)
ppt_symtab (name, symtab, oid)

Where oid is the object-id. The repository has a procedure *Display* that contains an integer variable *i* as follows:

```
void Display ()
{
   int i;
     ...
}
```

If we change the type of *i* to char as follows:

```
void Display ()
{
    char i;
     ...
}
```

Assuming that we know the object-id of the symbol, the CASE environment would issue the query:

```
replace ept_symtab (type=char) where
    ept_symtab.oid = 123
```

The following rule keeps the representations synchronized:

```
define rule symtab_synch_1 on
    replace to ept_symtab then
    replace ppt_symtab (symtab =
    update_symtab_proc (CURRENT.symtab,
    NEW.value , NEW.type)) where
    ppt_symtab.name = NEW.proc
```

Where *update_symtab_proc* is a procedure that updates the ppt_symtab.symtab field with the current symtab, and new value and type as its arguments. CURRENT refers to the values in the tuple *before* any changes, NEW refers to values in the tuple *after* any updates. See[12] for the complete syntax and semantics of POSTGRES rules.

7.5. Summary

In this section we illustrated the power of the TULUM environment architecture by using some of its advanced repository features like rules and procedures to provide practical and important services to CASE software. These examples show that we can provide very sophisticated services with a combination of features available in the current generation of DBMS systems. In some cases with advantages over the systems after which they are modeled, as in the case of RCS version control where TULUM provides versions at finer granule sizes (procedure versus files) as well as configuration management. In addition, our examples show that coarse granularity designs can benefit from much of the power of the DBMS, and that coarse and fine granularity designs can be effectively used in combination.

[19] Notice that the representations vary only in the granularity of the unit of storage. Thus the algorithms to convert between them are straightforward, which is not always the case.

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

8.1. Goals

In Chapter 1 we argued that our society is suffering from a *software crisis*, that Software Engineering needs to improve several orders of magnitude in many respects to solve it, and that powerful CASE environments are an essential tool to that end. We pointed out that a major unresolved issue in CASE is how to manage the large amounts of complex data it generates in a way that maximizes its power and minimize its space and efficiency costs. The major goal of our dissertation has been to make a concrete contribution in the area of persistent data management for CASE data structures. We concentrate our efforts on optimizing the transfer of data between clients and servers, and on evaluating the space and efficiency characteristics of different database designs.

To achieve these goals we first looked at the CASE database requirements. We prepared an exhaustive list, explained what each requirement was and why it was needed, and developed criteria to classify and evaluate these requirements. In addition, we looked at the different data managers available, classified them, and evaluated how well each class satisfied the different requirements. Based on our requirement analysis, we designed a CASE environment architecture. We illustrate the architecture with a new CASE environment, TULUM. TULUM uses an advanced client server architecture, in which the data repository provides data, object, knowledge, and process programming services.

We performed an in-depth analysis both of CASE data structures in a database context, and of the three important database design dimensions: number of relations, data representations, and granularity. We used the above data structure criteria to analyze the main data structure classes found in CASE: textual, tabular, graphical, unstructured, and sequential binary. Based on this analysis, we designed and conducted a series of experiments that evaluate how the different database parameters affect the space, functionality, and performance characteristics for the different data structure classes. Finally, we provided extended examples of how TULUM can provide sophisticated CASE services.

8.2. Results and Accomplishments

In this section we summarize the most important accomplishments and results that we obtained in the course of our research. We divide them into three areas: CASE database requirements, CASE environment architectures, and database representations for programs.

CASE Database Requirements

We provide a list of 25 different requirements, and explain each according to its function and its necessity.

Classification: We classify the requirements according to whether they are inherent to the data model or whether they possess functional attributes that can be added to any system regardless of its data model. We further classify the requirements as to whether they provide data, object, knowledge, or process programming support.

Data Managers: We look at the different approaches to provide CASE data management, and classify them into seven classes. Further, we develop a criteria to evaluate how well each data manager class satisfies each requirement.

We conclude that the object oriented DBMS and extended relational DBMS data manager classes have the power to fully support a modern CASE environment, especially when they are coupled with careful system architecture choices and judicious database designs.

CASE Environment Architectures

We separate system architecture into three areas: logical architecture, physical architecture. and computational paradigm.

Logical Architecture: The logical architectures we discuss are tool centered, data centered, knowledge centered, and process centered. For TULUM we adopt a knowledge and process centered logical architecture, with a general purpose next generation DBMS providing data, object, knowledge, and process programming services.

Physical Architecture: The physical architectures we cover are independent tools with files, single user integrated environments, single system-multiple user, and client server. TULUM uses an enhanced client-server physical architecture that shifts much of the DBMS processing load, especially the I/O, from the servers to the clients by providing a DBMS process in the client that manages all local data and that caches the global data. We provide tool to tool communication via shared memory, and toolkit to toolkit communication via messages. This allows for much faster communication, and assures that we only go through the data repository when its services are needed.

Computational Paradigm: As its computational paradigm, TULUM integrates parallelism, incrementality and lazy evaluation into the architecture to minimize the toolkit to DBMS as well as the client server interaction (both the amount of information and the number of interactions).

Database Representations for Programs

We develop a framework to analyze the FE data structures, as well as another framework to analyze the different database schemas. With these frameworks we analyze several data structures, and propose database designs for them. We propose ways to solve the intra-object reference problem of large granularity designs, and run a series of experiments to measure the performance of different database designs.

Analysis of Data Structures: In order to understand the data structures that we might want to store in a CASE database, we develop five criteria: internal structure, information content, quantitative parameters, qualitative parameters, and data sharing characteristics. Each criteria has different components.

Database Design Dimensions: We identify and analyze three important dimensions in which a database design can vary: the number of relations, the granularity of the unit of storage, and the data representation.

Intra-object References: We propose three alternatives to get fine granularity functionality with coarse granularity designs: abstract data types, location based, and embedded identifiers. The example applications that we provide show the many benefits that coarse granularity designs can obtain from storage in TULUM.

Experiments: We design and run a series of experiments to measure the space and time characteristics of the different database designs. Our main conclusions follow:

- Use fine granularity only when necessary.
- The Frontend-Backend interaction paradigm should be via sets, either with coarse granularity designs, or by fetching and storing sets of fine granularity elements together.
- Use ASCII representations only when necessary, as in a heterogeneous language or hardware environment, or convenient, as when ease and speed of implementation are paramount.
- To enhance performance and minimize space usage, minimize both the number of relations and the number of tuples.
- Increasing granularity brings better efficiency and less space usage but the gains "level off" below the procedure per tuple level.
- We identified three important phases during storage: data extraction, data conversion, and database query. During fetch, the phases are: database query, data conversion, and data structure reconstruction. We find that in order to maximize performance, the algorithms used at each phase need to be individually tuned to prevent bottlenecks.

8.3. Suggestions for Further Work

Our work, despite its contributions, leaves many areas that need to be explored further with respect to CASE data management. In this section we point out what some of those areas are, and what direction future work might take. The areas are intra-object references, large object implementations, synchronization of multiple representations, an evaluation of different database designs under more realistic conditions, and a transaction model for software engineering.

Intra-Object Access: We proposed three different algorithms to provide intra-object access when we have coarse granularity designs for data structures with a fine granularity internal structure. The time and space usage gains as we move from fine granularity to medium grain designs are very significant, while at the same time the loss of power is also substantial. It is therefore natural that we investigate ways to "have our cake and eat it too"; that is to get the advantages of coarse granularity while avoiding or at least minimizing its disadvantages.

Large Objects: Next Generation database systems are implementing large objects in different ways. For example, POSTGRES v4.0 intends to provide three different large object interfaces:

1) User-file, in which the database stores the full path name to the file, and the large objects are owned and manipulated by the user directly via the operating system.

2) Postgres-file, in which the database creates and owns the file in which the large object is stored. Updates are kept as forward deltas in the buffer pool, and POSTGRES provides its own I/O routines.

3) Big-object, in which the object is stored as a relation in contiguous pages with sequence numbers, and POSTGRES simply provides the same interface as the operating system.

Other systems use different schemas. A comparative study of the most interesting implementations is needed.

Multiple Representations: An integral part of modern CASE environments are multiple representations of the same data, as well as maintaining multiple dependencies among different software documents. It is very important to maintain consistency among the different representations as well as among the interrelated documents. There are different mechanisms for accomplishing this, including database rules, database views, and database procedures. In addition there are a variety of policies that can be used, such as keeping all representations and documents materialized, materializing some and computing the rest from these, pre-computing and caching the derived representations, and using lazy evaluation. An in-depth analysis of the alternatives, as an investigation into the design and implementation of experiments to measure the space, functionality, and time characteristics of the different mechanisms and policies, is a promising area of research.

Database Designs: We would like to see an evaluation of the different database designs under real-life CASE environment use. Some of the interesting conditions under which we would like to evaluate the different database designs include many versions and configurations, long transactions, and multiple users.

Transaction Models: Although we do not directly address it in our research, one of the most important open issues in Software Engineering is that of transaction models. We need research that compiles a list of existing transaction models, proposes new ones, and classifies them. Further, sample implementations and careful experimental design would allow us to evaluate their performance under well defined criteria.

The more ambitious, long-term goal that we would like to see fulfilled is the full development of the CASE environment of the future. In particular, we would like to see the implementation of a TULUM-like environment. The use of the many powerful features of this environment could bring about the implementation of many complex subsystems: version and configuration management, process programming language, and reuse methodologies. We envision the integration of these subsystems into a powerful, coherent, simple-to-use CASE environment that relieves the software engineer of mundane and repetitive tasks, anticipates her actions when appropriate, filters out unnecessary information, and ultimately increases the quality and decreases the costs of the resulting software systems.

8.4. Concluding Remarks

We have reached the end of this long and winding road. We hope that our work can be useful to both practitioners and researchers in understanding the issues involved in storing large amounts of complex data in sophisticated data managers. A large part of our work can be generalized to **any** application that needs to efficiently store and retrieve complex data, in particular the work on database representations for programs and the results of our experiments.

The 1990's will be the decade of the database, which will be called upon to store ever larger volumes of data, to provide much enhanced performance, and more sophisticated services, and to do this in a manner that is easy to use and modify. This work is our small contribution to this exciting and practical arena.

References

1. B.W. Kernighan and R. Pike, *The UNIX Programming Environment,* Prentice Hall (1984).
2. W. Passman, ''Architecture of the Atherton Software BackPlane,'' *Proc. 1989 ACM SIGMOD Workshop on Software CAD Databases,* pp. 105-108 (FEB 1989).
3. Cadre Technologies, Inc., *Teamwork,* Cadre Technologies, Inc., Mountain View, CA (1990).
4. L. M. Masinter, ''Global Program Analysis in an Interactive Environment,'' PhD Thesis, Stanford University (January 1980).
5. W. Teitelman, *Interlisp Reference manual.* 1978.
6. Y.F. Chen and C. V. Ramamoorthy, ''The C Information Abstractor,'' *Tenth International Computer Software And Applications Conference (COMPSAC),* pp. 291-298 (October 1986).
7. M.H. Kim, Y.-C. Shim, and C.V. Ramamoorthy, ''APAS: The Ada Programming Assistant System,'' *Proceedings of ICCM 1988,* (October 1988).
8. D.G. Belanger, R.J. Brachman, Y.-F Chen, P.T. Devanbu, and P.G. Selfridge, ''Toward a Software Information System,'' *AT&T Technical Journal,* (March/April 1990).
9. A.V. Aho and J.D. Ullman, *Principles of Compiler Design,* Addison Wesley (1979).
10. T. Reps, ''Generating Language Based Environments,'' *ACM Doctoral Dissertation Award.,* (1983).
11. R.A. Ballance, M.L. Van-De-Vanter, and S.L. Graham, ''The Architecture of Pan I,'' Memo No. M88/409, Electronics Research Laboratory, U.C. Berkeley (August 1987).
12. M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos, ''On Rules, Procedures, Caching and Views in Data Base Systems,'' *Proceedings of the 1990 SIGMOD Conference,* (1990).
13. M. Stonebraker and L. Rowe, ''The Design of Postgres,'' *Proceedings of the 1986 ACM SIGMOD Conference* , pp. 340-355 (June 1986).
14. M. Linton, ''Queries and Views of Programs using a Relational Database System,'' PhD Dissertation, Computer Science Division, UC Berkeley, Berkeley, California (1984).
15. M.A. Linton, ''Distributed Management of a Software Database,'' *IEEE Software,* (Nov. 1987).
16. B. Velez, ''Database Representations For Programs,'' MS Report, University of California, Berkeley (November 1988).
17. W. Paseman, *personal communication,* Atherton Technology, (April 1989).
18. M.H. Penedo, ''Prototyping a Project Master Data Base For Software Engineering Environments,'' *Proceedings ACM SIGSoft SIGPLAN Software Engineering Symposium on Practical SDE's,* (December 1986).

19. G. Clemm and L. Osterweil, ''A Mechanism for Environment Integration,'' *ACM Transactions on Programming Languages and Systems* **12**(1) pp. 1-25 (January 1990).

20. R. Snodgrass and K. Shannon, ''Fine Grained Data Management To Achieve Evolution Resilience in a Software Development Environment,'' *ACM SIGSoft Software Engineering Notes* **15**(6)(Dec. 1990).

21. C. Rich and R.C. Waters, ''The Programmer's Apprentice,'' *Addison Wesley*, (1990).

22. L. Gilham, *KBSA-PMA Program Specification,* Kestrel Institute (Nov. 1986).

23. G.E. Kaiser, P.H. Feller, and S.S. Popovich, ''Intelligent Assistance for Software Development and Maintenance,'' *IEEE Software*, (May 1988).

24. R.N. Taylor, F.C. Belz, L.A. Clarke, L. Osterweil, R.W. Selby, J.C. Wileden, A.L. Wolf, and M. Young, ''Foundations for the Arcadia Environment Architecture,'' *Proc. ACM SIGSOFT 3rd Symp. on Software Development Environments*, (1988).

25. A. Silberschatz, M. Stonebraker, and J. Ullman, ''Database Systems: Achievements and Opportunities,'' *Communications of the ACM* **34**(10)(October 1991).

26. L. Rowe and S. Wensel, editors, ''Report on the 1989 Software CAD Databases Workshop,'' *11th World Computer Conference IFIP Congress '89*, (Feb. 1989).

27. N. Roussopoulos, L. Mark, T. Sellis, and C. Faloutsos, ''An Architecture For High Performance Engineering Information Systems,'' *IEEE Transactions on Software Engineering* **17**(1)(Jan 1991).

28. M. Stonebraker and G. Kemnitz, ''The POSTGRES Next-Generation Database Management System,'' *Communications of the ACM* **34**(10)(October 1991).

29. M. Stonebraker, E. Hanson, and S. Potamianos, ''A Rule Manager for Relational Database Systems,'' *Memo No. M86/85*, pp. 47-68 Electronics Research Laboratory, U.C. Berkeley, (June 1987).

30. M. Stonebraker, ''The Design of the POSTGRES Storage System,'' *Memo No. M86/85*, pp. 69-90 Electronics Research Laboratory, U.C. Berkeley, (June 1987).

31. L.A. Rowe and M.R. Stonebraker, ''The POSTGRES Data Model,'' *Proc. 13th VLDB*, (SEP 1987).

32. P. K. Garg and W. Schacchi, ''A Hypertext System to Manage Software Life-Cycle Documents,'' *IEEE Software* **7**(3)(May 1990).

33. P.A. Bernstein, ''Database System Support for Software Engineering -- An Extended Abstract --,'' *Proc. Ninth Intl. Conf. on Software Engineering*, (March 1987).

34. P. Butterworth, A. Otis, and J. Stein, ''The GemStone Object Database Management System,'' *Communications of the ACM* **34**(10)(October 1991).

35. Ontologic, Inc., *VBase,* Ontologic, Inc., Mountain View, CA (1990).

36. M. Stonebraker, M. Hirohama, and L.A. Rowe, ''The Implementation of POSTGRES,'' *IEEE Transactions on Knowledge and Data Engineering* **2**(1)(March 1990).

37. L.M. Hass *et. al.*, "Starburst Mid-Flight: As the Dust Clears," *IEEE Transactions on Knowledge and Data Engineering* **2**(1)(March 1990).

38. K. Wilkinson, P. Lyngboek, and W. Hasan, "The Iris Architecture and Implementation," *IEEE Transactions on Knowledge and Data Engineering* **2**(1)(March 1990).

39. C. J. Date, *An Introduction to Data Base Systems,* Addison Wesley, Reading, MA (1986).

40. P.P. Chen, "The Entity-Relationship Model: Toward a Unified View of Data," *ACM TODS* **1**(1)(MAR 1976).

41. C. J. Date, "Referential Integrity," *Proc. Seventh Intl. VLDB Conf.*, (1981).

42. M. Stonebraker, "Managing Persistent Objects In A Multi-Level Store," Memo No. M91/16, Electronics Research Laboratory, U.C. Berkeley (March 1991).

43. R. Katz, "Managing the Chip Design Database," *Computer Magazine* **16**(12)(DEC 1983).

44. Data Base Task Group, *Report to the CODASYL Programming Language Committee.* April 1971.

45. F. Cacace, S. Ceri, S.C. Reghiazzi, L. Tanca, and R. Zicari, "Integrating Object Oriented Data Modelling with a Rule-Based Programming Paradigm," *Tech. Report 90.008, Dipartamento di Ellectronica, Politecnico di Milano,* (Nov. 1989).

46. Chimenti et. al., "The LDL System Prototype," *IEEE J. Data and Knowledge Engineering*, (March 1990).

47. M. Stonebraker, *Class Notes for Computer Science 286.* Spring 1987.

48. M. Carey, D. DeWitt, and et. al., "The EXODUS Extensible DBMS Project: An Overview," *In Readings in Object-Oriented Databases*, Morgan Kaufmann, (1990).

49. D. Batory, T. Leung, and T. Wise, "Implementation concepts for an extensible data model and data language," *ACM Trans. on Database Systems* **13**(3)(Sept. 1988).

50. E.N. Hanson, T.M. Harvey, and M.A. Roth, "Experiences in DBMS Implementation Using an Object-Oriented Persistent Programming Language and a Database Toolkit," *OOPSLA'91 Conference Proceedings*, (1991).

51. S.S. Simmel and I. Godard, "The Kala Basket: A Semantic Primitive Unifying Object Transactions, Access Control, Versions, and Configurations," *OOPSLA'91 Conference Proceedings*, (1991).

52. S.B. Zdonick and M.A. hornick, "A Shared, Segmented Memory System for an Object Oriented Database," *ACM Trans. of Office Information Systems* **5**(1)(Jan. 1987).

53. Relational Technology Inc., *INGRES Reference Manual,* Version 3.0, VAX/VMS, Relational Technology, Inc., Berkeley, CA (MAY 1984).

54. Oracle Corporation, Inc., *ORACLE,* Oracle Corporation, Inc., Mountain View, CA (1990).

55. Relational Technology Inc., *INGRES Object Management Extension Users Guide,* Relational Technology, Inc., Berkeley, CA (JAN 1990).

56. G. Wiederhold, T. Barasalou, and S. Chaudhuri, "Managing Objects in a Relational Framework," *Stanford Univ, Tech. Report,* (1990).

57. D. Maier, J. Stein, A. Otis, and A. Purdy, "Development of an Object-Oriented DBMS (GemStone)," *OOPSLA'86 Conference Proceedings,* (Sept-Oct 1986).

58. M. Atkinson *et. al.*, "The Object-Oriented Database System Manifesto," *Altair Technical Report 30-89,* (August 1989).

59. M. Stonebraker, "Introduction to the Special Issue on Database Prototype Systems," *IEEE Transactions on Knowledge and Data Engineering* **2**(1)(March 1990).

60. O. Deux et al., "The O2 System," *Communications of the ACM* **34**(10)(October 1991).

61. G.M. Lohman, B. Lindsay, H. Pirahesh, and K. Bernhard Schiefer, "Extensions to Starburst: Objects, Types, Functions, and Rules," *Communications of the ACM* **34**(10)(October 1991).

62. S.C. Johnson, "Yacc: Yet Another Compiler-Compiler," *UNIX Programmers Manual - Supplementary Documents,* (March 1984).

63. D.B. Smith and P.W. Oman, "Software Tools in Context," *IEEE Software* **7**(3)(May 1990).

64. L. Miguel, M.H. Kim, and C.V. Ramamoorthy, "A Knowledge and Data Base for Software Systems," *Proc. of the 2nd Intl. Conf. on Tools for Artificial Intelligence,* (November 1990).

65. R.M. Stallman, "EMACS, the extensible, customizable, self-documenting display editor," *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation,* (June 1981).

66. W.F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," *Proc. IEEE 6th Int. Conference on Software Engineering,* (1982).

67. S.I. Feldman, "Make -- A Program For Maintaining Computer Programs," *Software - Practice and Experience* **9** pp. 255-265 (April 1979).

68. M.R. Stonebraker *et. al.*, "The Design and Implementation of INGRES," *ACM TODS,* pp. 189-222 (SEP 1976).

69. C.V. Ramamoorthy, L. Miguel, and Y-C. Shim, "On Issues In Software Engineering And Artificial Intelligence," *Intl. Journal of Software Engineering and Knowledge Engineering* **1**(1)(Spring 1991).

70. G.E. Kaiser and D. Garlan, "Melding Software Systems from Reusable Building Blocks," *IEEE Software,* (July 1987).

71. S.P. Reiss, "Connecting Tools Using Message Passing in the Field Environment," *IEEE Software,* (July 1990).

72. A. Goldberg, *Smalltalk-80: The Interactive Programming Environment,* ADDISON (MAY 1983).

73. C.W. Krueger, B.J. Staudt, and A.N. Habbermann, ''Scaling Up Integrated Software Development Environment Databases,'' *Proc. 1989 ACM SIGMOD Workshop on Software CAD Databases*, (Feb. 1989).

74. W. Rubenstein, M. Kubicar, and R. Cattell, ''Benchmarking Simple Database Operations,'' *ACM-SIGMOD Conference on the Management of Data*, pp. 387-394 (1987).

75. N. Roussopoulos and A. Delis, ''Modern Client-Server DBMS Architectures,'' *SIGMOD RECORD* **20**(3)(September 1991).

76. D. Dewitt et al., ''Implementation Techniques for Main Memory Data Base Systems,'' *Proc. ACM SIGMOD Conference on Management of Data*, (1984).

77. T. Lehman and M. Carey, ''Query Processing in Main Memory Database Management Systems,'' *Proc. ACM SIGMOD Conference on Management of Data*, (June 1986).

78. R.G. Cattell, ''Introduction to Special Issue on Next Generation Database Systems,'' *Communications of the ACM* **34**(10)(October 1991).

79. The Committee for Advanced DBMS Function, ''Third Generation Data Base System Manifesto,'' *SIGMOD Record*, (September 1990).

80. R.A. Ballance, J. Butcher, and S.L. Graham, ''Grammatical abstraction and incremental syntax analysis in a language-based editor,'' *Proc. SIGPLAN '88 Conference on Programming Language Design and Implementation*, (1988).

81. C. Mosher, *POSTGRES Reference Manual.* 1991.

82. D.A. Patterson, G. Gibson, and R.H. Katz, ''A Case for Redundant Arrays of Inexpensive Disks (RAID),'' *Proceedings 1988 ACM-SIGMOD International Conference on Management of Data*, (August 1988).

83. G.P. Copeland and S.N. Khoshafian, ''Object Identity,'' *Proc. 1986 OOPSLA Conf.*, pp. 406-416 (SEP 1986).

84. Sun Microsystems Inc., ''XDR: External Data Representation Standard,'' *Network Working Group Request for Comments 1014*, (June 1987).

85. B. Meyer, ''The Software Knowledge Base,'' *Proc. IEEE Conf .... ICASE ???*, (June 1985).

86. J. Smith and D. Smith, ''Database Abstractions: Aggregation and Generalization,'' *ACM TODS*, (June 1977).

87. S.D. Conte, H.E. Dunsmore, and V.Y. Shen, *Software Engineering Metrics and Models*, Benjamin/Cummings Publishing Company (1986).

88. G.L. Steele, *Common Lisp - The Language*, Digital Press (1984).

89. D. Bitton, D. DeWitt, and C. Turbyfill, ''Benchmarking Data Base Systems: A Systematic Approach,'' *Proceedings of the ACM 1983 Symposium on Very Large Data Bases*, ().

90. R. Cattell and J. Skeen, ''Object Operations Benchmarks,'' *ACM Trans. On Database Systems*, (Accepted for publication.).

91. C. Lamb, G. Landis, J. Orenstein, and D. Weinreb, ''The ObjectStore Database System,'' *Communications of the ACM* **34**(10)(October 1991).

92. C.V. Ramamoorthy, V. Garg, and A. Prakash, ''Support for Reusability in Genesis,'' *IEEE Transactions on Software Engineering* **14**(8)(August 1988).

93. P. Bernstein and N. Goodman, ''The Failure and Recovery Problem for Replicated Databases,'' *Proc. Second Symp. on the Principles of Distributed Computing*, (August 1983).