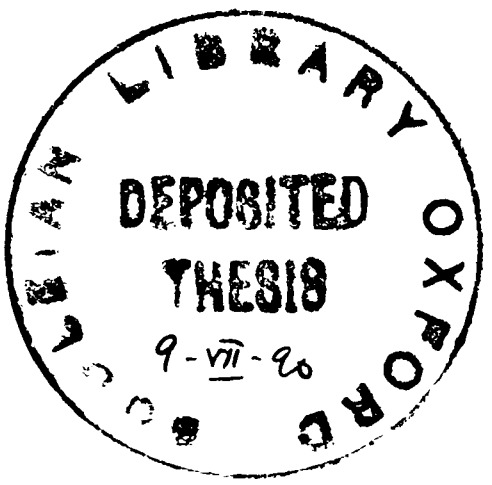
A Functional Database



Phil Trinder

Wolfson College and
Programming Research Group,
University of Oxford.

A Thesis submitted for the degree of Doctor of Philosophy
Michaelmas Term, 1989

Abstract

A Functional Database

Phil Trinder
Wolfson College

D.Phil.
Michaelmas Term, 1989

This thesis explores the use of functional languages to implement, manipulate and query databases.

Implementing databases. A functional language is used to construct a database manager that allows efficient and concurrent access to shared data. In contrast to the locking mechanism found in conventional databases, the functional database uses data dependency to provide exclusion. Results obtained from a prototype database demonstrate that data dependency permits an unusual degree of concurrency between operations on the data. The prototype database is used to exhibit some problems that seriously restrict concurrency and also to demonstrate the resolution of these problems using a new primitive. The design of a more realistic database is outlined. Some restrictions on the data structures that can be used in a functional database are also uncovered.

Manipulating databases. Functions over the database are shown to provide a powerful manipulation language. How to make such functions atomic is described. Such atomic transaction-functions permit consistent concurrent transformations of a database. Some issues in the transaction model are also addressed, including nested transactions.

Querying databases. Others have recommended list comprehensions, a construct found in some functional languages, as a query notation. Comprehensions are clear, concise, powerful, mathematically tractable and well integrated with a functional manipulation language. In this thesis comprehensions are proved to be adequately powerful, or relationally *complete*. Database and programming language theories are further integrated by describing the relational calculus in a programming language semantics. Finally, the mathematical tractability of the notation is used to improve the efficiency of list comprehension queries. For each major conventional improvement an analogous comprehension transformation is given.

Contents

I Introduction **1**

1 Introduction **2**

1.1 Ethos 2

1.2 Contributions 4

1.3 Organisation 5

1.4 Authorship 7

2 Referential Transparency **8**

2.1 Definition 8

2.2 Consequences 9

2.3 Centrality 11

2.4 Impact on Data Languages 12

3 Related Work **13**

3.1 Implementation Language 13

3.1.1 Requirements 13

3.1.2 Current Languages 14
3.1.3 Persistence 16
3.2 Data Manipulation Language 17
3.2.1 Requirements 17
3.2.2 Current Languages 19
3.2.3 Declarative File Systems 19
3.2.4 Comparison 22
3.3 Query Language 23
3.3.1 Requirements 23
3.3.2 FQL 24
3.3.3 FDL 24
3.3.4 Query Language Work 24

II Implementation 26

4 Bulk Data Management 27

4.1 Introduction 27
4.2 Bulk Data Structures 28
4.2.1 Non-destructive Update 28
4.2.2 Trees 29
4.2.3 Efficiency 31
4.3 B-trees 33

4.3.1 Description 33
4.3.2 Example 36
4.4 Bulk Data Manager 37
4.4.1 Transaction Functions 37
4.4.2 Manager 38
4.4.3 Multiple Users 39

5 Concurrency **41**
5.1 Introduction 41
5.2 Prototype Database 42
5.2.1 Hypothetical Machine 42
5.2.2 Metrics 44
5.2.3 Database Architecture 45
5.3 Bulk-Data Operations 45
5.3.1 Potential 45
5.3.2 Lookups 47
5.3.3 Updates 49
5.3.4 Effect of Database Size 53
5.3.5 Data Dependent Exclusion 57
5.3.6 Typical Mix 62
5.4 Transaction Processing 64
5.4.1 Drawbacks 64

5.4.2 Optimistic If 65
5.4.3 Friedman and Wise If 73
5.4.4 FWOF 76
5.4.5 Balancing 76

6 Database Support **79**
6.1 Introduction 79
6.2 Multiple Classes 80
6.3 Views and Security 82
6.4 Data Structures 83
6.4.1 Conventional Structures 84
6.4.2 Graph Structures 84
6.4.3 Secondary Indices 85
6.5 Data Models 90
6.5.1 Relational Model 90
6.5.2 Functional Data Model 94
6.5.3 Closure Problem 98

III Manipulation **101**

7 Transactions **102**
7.1 Introduction 102
7.2 Atomicity 103

7.2.1 Serialisability 104
7.2.2 Totality 105
7.3 Transaction Issues 107
7.3.1 Long Read-Only Transactions 107
7.3.2 Long Transaction Restart 110
7.3.3 Non-Terminating Transactions 111
7.3.4 Nested Transactions 112

IV Interrogation 113

8 Queries 114

8.1 Introduction 114
8.2 Relational Calculus 116
8.3 List Comprehensions 119
8.3.1 Introduction 119
8.3.2 Relational Queries 120
8.3.3 Extra-Relational Queries 121
8.4 Syntactic Correspondence 124
8.5 Translation Rules 125
8.5.1 Outline 125
8.5.2 Relational Calculus Syntax 127
8.5.3 List Comprehension Syntax 127

8.5.4 Translation Functions 128
8.5.5 Example 131
8.6 Semantics 133

9 Improving Queries **135**
9.1 Introduction 135
9.2 Improvement Environment 136
9.3 Algebraic Improvements 137
9.3.1 Selections 138
9.3.2 Converting Product into Join 139
9.3.3 Combination of Unary Operations 141
9.3.4 Common Subexpressions 142
9.3.5 Projections 143
9.4 Algorithm Example 144
9.5 Implementation-based Improvements 148
9.5.1 Preprocessing Files 148
9.5.2 Evaluating Options 149
9.6 Extra-Relational Queries 151
9.6.1 Date's Example 152
9.6.2 Atkinson and Buneman's Example 153

V Conclusion **156**

10 Conclusion **157**

10.1 Summary 157

10.2 Future Directions 159

Appendices **161**

A Parallel Programs **161**

A.1 Bulk Data Manager 162

A.1.1 Standard Bulk Data Manager 162

A.1.2 Bulk Data Manager with Disk Delay 167

A.2 Bulk Data Operations 169

A.2.1 Lookups 169

A.2.2 Updates 170

A.2.3 Updates with Disk Delay 171

A.2.4 Read and Write Programs 172

A.2.5 Typical Mix 173

A.3 Transactions 174

A.3.1 Five Bank Transactions 174

A.3.2 Two Long Transactions 175

A.3.3 Two Transactions with a failing *Optif* 176

A.3.4 Long Read Transaction 177

A.3.5 Long Read Transaction and Deposits 178

B ML File Manager **180**

C Denotational Semantics **189**

C.1 Semantic Domains 189

C.2 Semantic Functions 190

Bibliography **192**

Acknowledgements

I am indebted to Phil Wadler and Carroll Morgan, my supervisors. Both provided invaluable encouragement and sound advice. I am also grateful to many others, all of whom cannot be named. Malcom Atkinson and John Hughes made it possible for me to work at Glasgow University for two years. Pat Terry started it all by suggesting a British doctorate. Paul Fertig interested me in declarative transaction processing. Colleagues in both Oxford and Glasgow provided stimulation and a pleasantly bizarre working environment. Gabre Baraki, Andrew Kay and John Launchbury all listened to half-baked ideas and made useful comments. The secretarial staff in both departments provided valuable assistance.

I would like to thank my family and friends for their help. Dad for financial support, James for reading the thesis and both sets of Grandparents for providing a peaceful retreat. Finally I am grateful to Oxford University and the Committee of Chancellors and Vice-Principals for the scholarships that made this degree possible.

Part I

Introduction

Chapter 1

Introduction

1.1 Ethos

This Thesis explores the use of functional languages to implement, manipulate and query databases. The advantages of functional languages are often cited and recited. This Thesis argues that the distinctive advantages and disadvantages of functional languages derive from their enforcement of referential transparency. The distinctive advantages are an amenability to reasoning, the promise of painless parallelism and lazy evaluation [11, 55, 88].

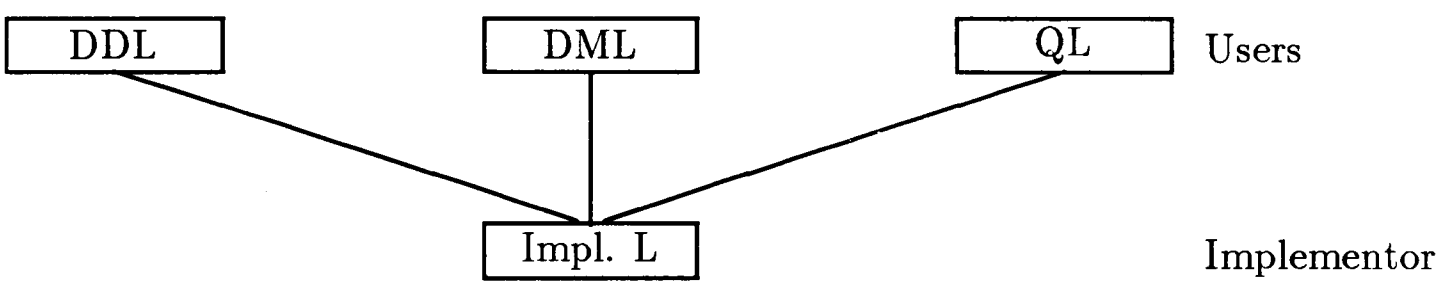
These advantages lead the author to believe that, if the outstanding problems with functional languages can be resolved, they will form the next class of high-performance, general purpose, languages. Current problems with some functional languages are their slowness, large memory requirements, determinism and poor data storage facilities.

Data storage is important because almost all non-trivial programs manipulate permanent data. At present most declarative languages are guest languages on single-processor procedural machines, and able to preserve their data in the file system provided by the host machine. However, the interface to the file system that is provided by the guest languages is primitive and often not referentially transparent. A guest functional language that

adopted the proposals made in this Thesis would have rich, but declarative, data manipulation and query facilities.

Further, there are several machines, and more under development, that are designed to evaluate functional languages in parallel [3, 22, 28, 69]. To integrate with the declarative framework and thus be efficient on these machines, the manipulation and query languages must be implemented in a parallel functional language. From a database viewpoint, if parallel functional languages become a fast alternative to procedural languages, then it is desirable to implement databases in them. There is already some evidence that declarative multi-processors, with their large memories, have potential for fast data manipulation [75].

There are four main kinds of database language. Databases are designed using a data definition language (DDL). Databases are created and maintained using data manipulation or data transaction language (DML). Databases are interrogated using a query language (QL). Database management systems are built using an implementation language. While the data definition, transaction and query languages are all employed by database users, the implementation language is used by the constructor of the database management system. The following schematic describes these relationships.



In this Thesis an implementation, a transaction and a query language are described but no data definition language. The reason for this is that data definition languages are the subject of much work in type theory. Type theory, however, is outside the scope of this Thesis. It should be easy to use the results of this Thesis together with the type theory results.

1.2 Contributions

The following contributions are made in the fields of databases and declarative programming languages.

- A database manager that allows efficient and concurrent access to data is constructed in a pseudo-parallel functional language.
- In contrast to the locking mechanism found in conventional databases, the functional database uses a novel exclusion mechanism, namely data dependency. Data dependency is shown to provide an unusual degree of concurrency between operations on the data.
- Some problems that seriously restrict concurrency are demonstrated and overcome using primitives, including a new primitive, 'optimistic if'.
- Some restrictions on the data structures that can be used in a functional database are also uncovered.
- Functions over the database are shown to provide a powerful manipulation language. Such transaction-functions are made atomic and permit consistent and concurrent database transformations.
- List comprehensions, a construct found in some functional languages, are proved to be an adequately powerful, or relationally *complete*, notation for expressing database queries.
- Database and programming language theories are further integrated by describing the relational calculus in denotational semantics, a programming language semantics.
- The mathematical tractability of list comprehension notation is used to develop a suite of transformations to improve the efficiency of comprehension queries. For each major conventional improvement an analogous list comprehension transformation is given.

1.3 Organisation

The remainder of this Thesis is made up of five parts. There is an introductory part and a part to describe each of the implementation, transaction and query languages. Finally there is a concluding part, and some Appendices.

The language parts have the following dependencies. The transaction language is dependent on the implementation accepting functions over the database as transactions. In contrast, while the query language is cleanly integrated with our implementation it is not restricted to it. Indeed the query notation may be used both in procedural languages and in conjunction with conventional databases.

Part I Introduction

Chapter 2 covers referential transparency. A definition is given and the consequences of forcing a programming language to be transparent are described. It is argued that enforcing referential transparency is the fundamental difference between functional and procedural languages. By applying functional languages to databases, the impact of referential transparency is being investigated. The impact on each of the data languages in this thesis is described.

Chapter 3 covers functional data languages. The requirements of implementation, manipulation and query languages are outlined. Other functional data languages are briefly described and related to the approach taken in this Thesis.

Part II Implementation

Chapter 4 covers the implementation of a bulk data manager. It is shown that a functional language can be used to implement efficient operations on trees. An overview of B-trees, a common type of tree in databases, is presented. A bulk data manager that uses trees and supports transactions from multiple users on a shared database is described.

Chapter 5 covers the introduction and control of concurrency within the database. A pseudo-parallel data manager is used to demonstrate concurrent bulk-data operations and concurrent transactions. In contrast to the

locking mechanism found in conventional databases the prototype manager uses data dependency as a novel exclusion mechanism. Data dependency is shown to permit an unusual degree of concurrency between transactions. It is also demonstrated that, within certain limits, the rate of processing transactions is independent of the size of the database. Some problems that severely restrict concurrency are identified and illustrated. Three primitives are proposed to resolve these problems. Two of the primitives are new and one of these, optimistic if, has been implemented. Optimistic if is used to illustrate concurrency both within and between transactions.

Chapter 6 covers the design of a more realistic functional database. The facilities illustrated are access to multiple classes, views, security, alternative data-structures and support for data models. A class of data structure that cannot be maintained under a non-destructive update regime is also encountered.

Part III Manipulation

Chapter 7 covers the use of functions as transactions that manipulate the database. How transaction-functions are made atomic is described and the techniques are compared with a conventional logging-and-locking approach. Some issues in the transaction model are also addressed. These include processing long read-only transactions, restarting long transactions, evicting non-terminating transactions and the provision of nested transactions.

Part IV Interrogation

Chapter 8 covers theoretical work on the expression of queries in programming languages. Queries are written as list comprehensions, a feature of some programming languages. Relational queries are demonstrated, as are queries requiring greater power than the relational model provides. It is argued that comprehensions are clear because of their close resemblence to the relational calculus. The power, or relational *completeness*, of list comprehensions is proved. Database and programming language theories are further integrated by describing the relational calculus in a programming language semantics.

Chapter 9 covers the improvement of list comprehension queries. For each major improvement strategy identified in the database literature an equiva-

lent improvement is given for comprehension queries. This means that existing database algorithms that improve queries using several of these strategies can be applied to improve comprehension queries. Extra-relational queries can also be improved. An example of each improvement is given.

Part V Conclusion

Chapter 10 summarises the results reported in the Thesis and concludes that functional languages have potential as database implementation, manipulation and query languages. Further research directions are also identified.

1.4 Authorship

All of Chapter 4, and some of Chapter 5, describes joint work reported in [5]. The parts of Chapter 5 that are joint work are as follows. Data dependency was identified as the exclusion mechanism, but the unusual degree of concurrency it permits was not recognised. The concurrency restrictions introduced by total transactions and tree balancing were identified. Two primitives, *optif* and *fwif* were proposed to overcome these restrictions. The suggestion in Chapter 7 that a non-deterministic primitive can be used to evict a non-terminating transaction is also reported in [5].

The work reported in Chapter 5 that was done solely by the author is as follows. A pseudo-parallel data manager has been implemented. The concurrency possible between bulk-data operations is demonstrated. It is shown that, within certain limits, the rate of processing transactions is independent of the tree size. Data dependency is used as a novel exclusion mechanism and is shown to permit an unusual degree of concurrency between transactions. The restriction that total transactions place on concurrency is demonstrated. To overcome this restriction the new primitive *optif* has been implemented and is used to provide concurrency both within transactions and between transactions. A space allocation problem with *optif* is encountered and a solution is proposed. A new primitive *fwof* that combines the behaviour of *fwif* and *optif* is proposed. Finally a new means of rebalancing the tree is proposed.

Chapter 2

Referential Transparency

This Chapter covers referential transparency. A definition is given and the consequences of forcing a programming language to be transparent are described. It is argued that enforcing referential transparency is the fundamental difference between functional and procedural languages. By applying functional languages to databases, the impact of referential transparency is being investigated. The impact on each of the data languages in this thesis is described.

2.1 Definition

Readers familiar with referential transparency and its consequences for programming languages may wish to omit both this Subsection and the next. Referential transparency is a fundamental property of mathematical notations. It was first described for propositions by Whitehead and Russell [98]. Quine defines when part of an English sentence is referentially transparent [73] and Stoy gives the following definition suitable for computing notations [82].

The only thing that matters about an expression is its value, and any subexpression can be replaced by any other equal in value. Moreover, the value

of an expression is, within certain limits, the same whenever it occurs.

Referential transparency allows a simple definition of equality: two expressions are equal if they denote the same value. Because the value is the only important feature of an expression, any two equal expressions may be interchanged. For example, sin(1+5) can be replaced by sin(6).

The qualification 'within certain limits' refers to the context that an expression occurs in. An expression may have different values in contexts in which the values of its free variables differ. For example, if $x = 6$, then sin(1+5), sin(6) and sin(x) are all interchangeable, provided that sin(x) is not placed in some context where x has been defined to be some value other than 6.

The clause stating that 'any sub-expression can be replaced by any other equal in value' can be deduced from the first clause of the definition. An immediate consequence of the clause 'The only thing that matters about an expression is its value' is that the value of a composite expression depends only on the *values* of its constituent expressions. Hence any sub-expression may be replaced by any other equal in value. The ability to substitute one expression for another is termed *equational reasoning* and is central to mathematical thought. Substitution facilitates proof, and the transformation and derivation of programs [17].

2.2 Consequences

Let us examine the impact of enforcing referential transparency on a programming language. We start by considering some aspects of conventional, referentially opaque, languages in order to have a basis for comparison. In a referentially opaque language, programs compute by effect [48]. A program proceeds by repeatedly computing a value and assigning it to a location in the store. Because this behaviour is so close to a von Neumann architecture, such programs are efficient on conventional machines.

The first consequence of enforcing referential transparency is freedom from a detailed execution order. As the effect of a conventional language statement may depend on the current contents of the store, the order in which the

statements are executed is crucial. This execution order must be specified by the programmer.

In contrast, an expression in a transparent notation must always have the same value in the same context. In other words, the value of an expression is independent of the history of the computation. Hence the order in which the expressions within a program are evaluated is not significant. This determinism implies that a program is simpler because it need not contain detailed sequencing information. Further, both lazy and parallel evaluation become natural alternatives.

Laziness is an evaluation strategy where the arguments to a function are evaluated only if they are required to compute the result of the function. Furthermore, the arguments are evaluated at most once. This strategy contrasts with most procedural languages which are strict: the arguments of a sub-program are always evaluated. In a lazy language a program can process infinite structures because only as much of the structure as is needed is generated. Lazy evaluation also aids modularity by separating data from control. A more complete description of laziness and its costs and benefits can be found in [48].

A second consequence is that referentially transparent programs may no longer have *side effects*. Side effects are those actions a sub-program performs in addition to computing the desired value. Such actions include assignment to global variables and performing input or output. Side effects allow some actions such as input and output to be expressed neatly. However, unless the additional effects are well documented, they may not be anticipated by users of the sub-program. For this reason, the presence of unnecessary side effects is regarded as undesirable [48, 90, 101].

The third consequence of enforcing referential transparency is that assignment may no longer be used. This is because a variable is a simple expression. As such, the definition of referential transparency states that a variable must always have the same value in the same context. Assignment violates this requirement by changing the value associated with a variable. The loss of assignment is perhaps the most significant result of constraining a language to be referentially transparent. Assignment is a fundamental operation in a von Neumann machine and disallowing its use in a machine carries a heavy

penalty.

Consider, for example, the task of decrementing the balance of a bank account. In a referentially transparent language, a new name must be associated with the new value. A new account record must be constructed containing the new balance, the unchanged information must be copied from the original account, and the result given a new name. This approach to modifying data is termed *non-destructive update*. A destructive update or assignment could simply change the balance part of the existing record. The non-destructive update requires additional space to be allocated for the new account record. Non-destructive update also requires more time, because the unchanged information must be copied from the original account into the new one.

The situation is even worse for database applications where large data structures are frequently modified. Consider the task of updating a bank account record that is part of a file containing thousands of similar records. A destructive update can simply alter the balance part of the specified record. A non-destructive update must create a new copy of the entire file. Chapter 4 demonstrates how this can be done efficiently and the uses to which the multiple copies of the file can be put.

Further discussions of referential transparency and its significance can be found in Bird and Wadler [17], Stoy [82] and Turner [88]. A full description of the suitability of functional languages for parallelism can be found in [70].

2.3 Centrality

Preserving referential transparency is the fundamental difference between functional and procedural languages. As described above, transparency facilitates reasoning about programs using proof, transformation and derivation. Transparent programs are free from a rigid evaluation order, making parallelism easier and lazy evaluation possible. Side effects are eliminated and a non-destructive update regime is enforced.

Some other advantages often claimed for functional languages include data

abstraction, pattern matching and higher-order functions. While these features are supported, they do not seem to be specific to functional languages. Data Abstraction originated in the procedural world [14], and is widely used there [2, 4, 37]. Pattern matching had its origins in procedural languages [34]. Many procedural languages also treat procedures as first class objects [8, 45].

2.4 Impact on Data Languages

Because referential transparency is the significant property of functional languages, when functional languages are used in databases the impact of referential transparency is being investigated. The consequences of preserving referential transparency in each of the data languages in this thesis are as follows. This theme is developed further in [86].

In the implementation language, referential transparency allows lazy lists, or streams, of requests to be directed to the database manager. The choice of data structures is limited because only a few can be updated efficiently under a non-destructive update regime. The multiple versions of the database generated by non-destructive update do, however, permit an unusual degree of concurrency. Further, referential transparency guarantees that the concurrent operations have simple semantics.

In the manipulation or transaction language the multiple versions of the database arising from non-destructive update make guaranteeing transaction properties easy. Because transaction-functions are transparent they can be reasoned about easily.

The lazy evaluation strategy underlying the query language makes database queries faster. Because the query notation is transparent it is amenable to transformation. In Chapter 8, transformations are given that improve the efficiency of queries.

Chapter 3

Related Work

This Chapter covers other functional data languages. The requirements of implementation, manipulation and query languages are outlined. Other functional data languages are briefly described and related to the approach taken in this thesis.

3.1 Implementation Language

3.1.1 Requirements

The implementation language is used to construct a database management system (DBMS). The most obvious requirement of an implementation language is the ability to store data permanently. Values must be named so that they can be retrieved by subsequent programs. What is actually preserved is a binding of names to values, or keys to records. A name, value pair is called an *entity* [40].

It must be possible to define efficient operations on the stored data in the implementation language. That is, it must be possible to implement operations to lookup, update, insert or delete entities that have a low space and

time cost. Any one of these operations is termed an *action* [40]. The implementation language must make it possible for many processes to perform actions on different entities concurrently.

A DBMS constructed in the implementation language will need a data model, such as the relational model. Such models enable the user to reason about the data at a higher conceptual level than that of files and records. The ability to support more than one data model is clearly desirable.

The implementation language should allow a rich set of data structures to be used to model the real world objects and relationships being represented. The DBMS built using the implementation language must also provide some facilities for recovering from system failure: should the database be damaged, it should be possible to retrieve the lost data as quickly and simply as possible.

In summary, the implementation language must provide permanent storage, efficient and concurrent actions, data model support, a rich set of data structures and support for failure recovery. As already described, functional languages seem well adapted to concurrency. They also provide a rich set of data structures. Chapter 4 demonstrates efficient actions and Chapter 5 concurrent actions. Chapter 6 demonstrates data model support. Current functional languages, however, do not provide adequate data storage facilities. This problem is explored in the following Subsections.

3.1.2 Current Languages

Most current functional languages provide two mechanisms for permanent storage, one primarily for programs and the other for data. This distinction is unnatural in functional languages.

Programs and small amounts of data are preserved as text in a conventional filing system and modified using an editor. At the 'top level' of the system the user is required to issue a sequence of commands that manipulate the store in an imperative manner. Effectively the declarative system has been embedded in an imperative shell, and this seems symptomatic of the guest

status of functional languages on procedural machines. Examples of this type of mechanism include KRC's and Miranda's scripts [15, 89].

Large amounts of data to be processed by a program ~~is~~ are kept in files. Most functional systems provide functions to read and write these files. These functions may be embedded within other functions, they need not be used at the top level. Examples of these functions include KRC's *read* and *write* [15] and Mirandas *read* and *tofile* [89].

The behaviour of these functions is far from desirable. Let us consider KRC's well documented *write* and *read* functions.

- *write fname x* will print the value of x into the file called *fname*.
- *read fname* will return the contents of the file *fname*.

The *write* operation is not performed until the *write* function is itself printed. If a program attempts to *read* the same file, evaluation order becomes important. In fact, the Miranda manual contains the solemn warning "Users who write Miranda programs which read and write the same file are warned that they are treading on dangerous ground and that the behaviour of such programs is unlikely to be reliable" [89].

The contents of the file created by *write* is the result of output being redirected to the file. The file contains a printable representation of the data. For example, numbers are represented as characters, and must be transformed back into the original type before programs can manipulate them.

The *read* function is inadequate because it provides no way to deal with a file whose content changes over time. Consider the function

$$getbal = converttonum\ (read\ \text{“}Balfile\text{”}),$$

that reads the string in a file and converts it into a number. When *getbal* is first called it will return the number currently stored in a file. Even if the contents of the file are changed it will continue to return the same value.

3.1.3 Persistence

Persistence [7] seems to offer a solution to the problem of long-term data storage in functional languages. Atkinson observed that in most existing languages only certain data structures may be permanently stored. Much of the effort in writing programs that manipulate permanent data is expended in unpacking the data into a form suitable for the computation and then repacking it for storage afterwards.

The idea behind persistent programming languages is to allow entities of any type to be permanently stored. The length of time that an entity exists, or its persistence, becomes an orthogonal property of the entity. Thus programs manipulating persistent entities need no longer pack and unpack the data.

Mathews [63] describes a persistent store as a cross between a virtual memory and a database. Entities in a database are structured — they may refer to other entities. Transfers to and from the database are usually made by explicit calls to special procedures. A virtual memory transfers unstructured chunks, or pages, between memory and backing store. It performs these transfers without explicit requests from the user. A persistent store contains structured data, performs transfers automatically and garbage collects any entities no longer in use.

Poly and ML are two near-functional languages that have been made persistent [63]. Either Poly or ML could support efficient data operations, provide data model support and allow arbitrary data structures to be used. Unfortunately Poly and ML are only near-functional and hence inherently sequential. As a result, locking or exclusion occurs at a database level. Many programs may read the contents of a database, but only one may write to it.

Chapters 4 and 5 show that persistence and a few parallelism primitives are all that need to be added to a lazy functional language before it can be used as an implementation language.

3.2 Data Manipulation Language

3.2.1 Requirements

A Data Manipulation Language (DML) is used to create, modify and inspect a database. After being manipulated a database is required to satisfy some consistency constraints [33]. In a mythical simple-minded bank, a consistency constraint could be that

$$Moneyheld = \Sigma Deposits - \Sigma Withdrawals$$

It is not sufficient to guarantee the constraints after every action because it is necessary to violate the constraints when modifying the database. For example, in a transfer of money between two accounts, money is withdrawn from the source account before being deposited in the destination account. After the withdrawal, but before the deposit the above consistency constraint is violated.

A transaction is a collection of actions that preserves consistency. Any transaction executed alone on a consistent database will transform it into a new consistent state. Maintaining consistency becomes difficult when concurrency is introduced. In a bank, for example, the fact that one account is being examined should not preclude other accounts from being accessed.

The problems that arise between concurrent processes with shared memory are well known. For example, to withdraw money from a bank account, the current balance is read and then a new balance is written. If two processes wish to withdraw money from the same account they may *interfere* in the following way. They may both read the current balance, then the first may write its new balance, followed by the second. In this case the effect of the first update will be ignored. To prevent interference in databases an exclusion mechanism such as locking is used. The reader is assumed to be conversant with the concept of record locking. Expositions can be found in [32, 33].

A transaction may perform some actions before discovering that, for some reason, it cannot complete, or *commit*. For example, an account entity it

accesses may not exist. In these circumstances the transaction must *abort.* The database is likely to be in an inconsistent state as a result of the actions the transaction has already performed. In aborting the transaction must return the database to its original, consistent, state. Transactions with this behaviour are termed *total.* To be more precise, a transaction is total if, providing it returns, it was carried out completely or (apparently) not started [61].

The qualification that a transaction must return refers to machine failures. Within a transaction the consistency constraints may be violated. Hence, if the machine fails during a transaction, the database may be left in an inconsistent state. In the event of a failure a DBMS should provide a recovery procedure that will return the database to a consistent state with a minimum of lost information. It may do so either by completing the transaction or by removing the effects of the transaction. This is where the implementation languages recovery mechanism from Subsection 3.1.1 is used. A DBMS that is able to recover from machine failures may guarantee a stronger property than totality, namely *reliability.* Totality guarantees that a transaction is executed zero or one times. Reliability guarantees that a transaction is evaluated exactly once.

When executing several transactions concurrently their actions may be ~~are~~ interleaved in time. To preserve consistency all of the actions of a transaction must occur against the same state of the database. A property strong enough to guarantee this is *serialisability.* The actions of a transaction are serialisable if, when a collection of transactions are carried out concurrently, the result is as if the individual transactions were carried out one at a time in some order [61].

It is desirable for transactions to be both total and serialisable. The conjunction of these two properties is termed *atomicity.* Logically the multiple actions of atomic transactions occur "instantaneously". To summarise, a transaction is a collection of actions that is atomic [61].

Once atomic transactions have been constructed, it is desirable to reuse them within other transactions. For example, a transfer transaction might be built out of a withdrawal and a deposit transaction. The use of a transaction as a *subtransaction* of another is termed *nesting.*

A transaction language should be amenable to reasoning. In particular it is desirable to transform transactions into a more efficient form. It is also desirable to prove properties of transactions. For example, it might be proved that a deposit immediately followed by a withdrawal of the same amount has no effect. In summary, a data manipulation language must be capable of supporting concurrent atomic transactions. It should also have a clean semantics and be able to nest transactions.

3.2.2 Current Languages

In Subsection 3.1.2 the file manipulating operations found in current functional languages were described. These are inadequate as a DML for several reasons. No provision is made for more than one process to access the files concurrently. Only one file may be accessed by these operations — there is no provision for consistent groups of updates. As described earlier, in some languages some combinations of read and write operations do not have well defined semantics. As a consequence programs that use these operations cannot be reasoned about . It also makes composing file manipulating functions to construct new functions, or nesting, unsafe.

3.2.3 Declarative File Systems

Friedman and Wise

Friedman and Wise have described a lazy applicative file system [36]. In their system file manipulation was restricted to top-level functions. In a given collection, several functions can take a file as an input parameter, but only one function can modify a file. It is envisaged that, in the presence of errors, such lazy file systems have unpleasant behaviour. It is well known that allowing an error to propagate from its point of occurrence complicates debugging [42, 48]. In an entirely delayed system an erroneous expression may be encountered that is the legacy of an unknown program evaluated at some unknown time in the past.

Lispkit

The Lispkit operating system was described by Henderson [49, 50] and Jones [57]. The file system allows files to be read and written by the user. A user communicates with the file system by sending a stream, or lazy list, of commands to it, and receiving a corresponding stream of responses. Lispkit is a distributed operating system and files may be shared between users. The Lispkit file system does not provide a DML and hence does not support atomic transactions.

Flagship PRM

The goal of the Flagship project was to build a multiple processor machine suited to evaluating declarative languages [3]. In the design of the operating system, or programmers reference model (PRM), transactions are provided as primitives. This ensures that update is implemented efficiently, correctly, consistently, securely and only once. PRM transactions have the side-effect of updating many entities atomically. Within a transaction, however, referential transparency is guaranteed. PRM transactions may be nested and are not restricted to top-level functions. Many transactions may occur concurrently.

The PRM has been implemented on a machine with 15 68020 processors. The Flagship machine achieves some impressive results for the DebitCredit benchmark [75]. DebitCredit is a bank transaction-processing benchmark [76]. Against a database of 30000 account records, 3000 teller records and 1000 bank records, the Flagship machine can process 45 transactions per second (TPS). Some comparative figures are as follows. It is not clear whether all of these figures are based on a database of the same size.

Machine	TPS
Flagship	45
Sun	5
IBM 4381-P22	22
DEC VAX 8830	27

PRM transactions do provide the database manipulating behaviour required

of a DML. However, the loss of referential transparency in much of the language is serious. The semantics of the transactions is complex, reasoning about programs becomes hard and evaluation order becomes significant.

DL

DL is a language developed by Breuer to support data definition, data manipulation and queries [19]. Data is viewed as definitions, and may be manipulated by top-level redefinition or assignment. Because DL is essentially single user, concurrency and consistency issues are not addressed.

FDL

FDL is a functional data language that supports the functional data model [72]. In most existing DMLs there is an uneasy coexistence of data model and computation model. FDL provides a single data and computation model, that of the lambda calculus. In FDL there are three types of function: computational, data and a combination of the two. Data functions are essentially definitions and database updates occur at a meta-level and are viewed as function redefinitions. FDL is also single-user and hence ignores concurrency and consistency issues. Although the consistent data and computation model simplifies a database programmers world, the meta-level for update reintroduces some complexity.

Id

To facilitate data manipulation some extensions to the dataflow language Id have been proposed [67]. A new type of function called an index function is introduced. Index functions are initially undefined everywhere. Information about the function is added incrementally, defining it over a larger and larger domain. The transaction model is derived from the model presented in this Thesis. A transaction is an expression evaluated in the context of the current database and produces some output and a new database. Serialisability is

guaranteed because only one transaction is manipulating the database at any one time.

The dataflow model underlying Id provides adequate concurrency within a transaction. It is not clear, however, whether parallelism between transactions is possible. Having two types of function also introduces some complexity into the programmers world.

3.2.4 Comparison

In the transaction language presented in this thesis the database is represented as an abstract data type. Transactions are functions that take the database as an argument and return an updated database and some output. A process adds a transaction to a stream of requests to the database and receives a response from the database on an input stream. Chapter 5 demostrates parallelism both within transactions and between transactions. Chapter 7 describes how transaction-functions are made atomic. Because transactions are functions they can be nested simply by invoking one function within another. The problem of machine failure is not addressed in detail.

Like the Friedman and Wise file system the transaction language allows read-only sharing of parts of the database. The streams of requests and responses are similar to those found in the Lispkit operating system. Unlike the PRM transactions the transaction-functions are pure — they have no side-effects. As a result they have simple semantics.

Viewing the database as an abstract data type is in contrast to both DL and FDL which view data as definitions. Concurrency issues that are not relevant in DL and FDL are also addressed. Some of the Id proposals are derived from work presented in this thesis. They differ in being based on a dataflow model and by introducing a new type of function.

3.3 Query Language

3.3.1 Requirements

A query language is used to interrogate the database. A bank manager, for example, might wish to discover which customers have overdrafts. The query language should be clear — how to express a query should be intuitively obvious. Conversely, what a query means should also be immediately apparent.

A query language should be powerful. Codd defined a query notation to be relationally *complete* if it is at least as expressive as the relational calculus [27]. However, many queries require more power than that provided by the calculus. Typically these queries entail computation or recursion. A bank teller might, for example, wish to compute the total of all a customer's accounts. A recursive language is needed, for example, to express queries over recursive data structures such as trees or graphs.

A query language should be concise. It should not be necessary to give a great deal of verbiage to specify a simple query. For example, if a query specifies only one attribute of an entity it should not be necessary to enumerate all of the other attributes of the entity.

A query language should have a sound and tractable mathematical basis. This facilitates reasoning about queries. For example, most queries can be evaluated in different ways, and some evaluation orders are more efficient than others. It is desirable that a simple specification of the query can be transformed into a more efficient form.

A query language should be well integrated with the data manipulation language. This often not the case. For example a query language may be based on relations and the manipulation language based on von Neumann machines. In summary, the notation is required to be clear, powerful, concise, mathematically sound and well integrated with the manipulation language.

3.3.2 FQL

FQL is an early functional query language developed by Buneman, Frankel and Nikhil [21]. It is based on the FP language [11]. In FQL a small set of functions are composed to process lists of entities. The lists are processed lazily, and this is shown to reduce the number of disk accesses required to evaluate a query. Several of the functions take other functions as arguments, i.e. they are higher-order. FQL is used as an intermediate language in a commercial database product.

FP is mathematically sound, with known identities [11]. The FQL notation is extremely concise. However, the author finds the parameterless notation makes it far from clear. For the same reason FQL will look strange to users. FQL is closely related to the functional data model, and hence reasonably powerful. The power of the language is, however, restricted by the fact that it is difficult to define new higher-order functions.

3.3.3 FDL

As described in Section 3.2.4, FDL supports the functional data model (FDM). The functional data model is based on functions and sets of entities. In FDL, Poulovassilis shows how these concepts can be cleanly integrated into a functional language. The FDM gives a clear meaning to queries. Poulovassilis recommends the use of list comprehensions for expressing database queries. She demonstrates how, used in conjunction with recursive functions, computation and recursion can be expressed using comprehensions.

3.3.4 Query Language Work

A query language was required to be clear, powerful, concise, mathematically sound, and well-integrated. Other workers have also shown that list comprehensions are clear, powerful, concise and well integrated [20, 67, 72]. In Chapter 8 the power of comprehension notation is proved. It is argued that clarity is aided by the close correspondence between comprehensions and

the relational calculus. Databases and programming languages are further integrated by describing the relational calculus in a programming language semantics. In Chapter 9 the sound mathematical basis of comprehensions is used to develop transformations to improve the efficiency of queries.

The power of comprehension notation is proved by giving a translation of relational calculus queries into list comprehensions. The use of translation between a relational formalism and a programming language has some precedent in the database world. For example, the semantics of SQL has been described by translation into the relational calculus [92] and into the relational algebra [23].

The task of improving queries has received much attention. Both Date [32] and Ullman [91] give surveys of the field and identify two classes of improvement techniques — algebraic and implementation-based. Seminal work on algebraic improvements can be found in [56, 43] and [81]. Seminal work on implementation-based improvements can be found in [18, 81] and [100].

Equivalent improvements are given for each major conventional improvement strategy. Some of these improvements are effective because, as demonstrated in FQL, a lazy evaluation strategy reduces the number of disk accesses required. Most of the improvements entail transforming a simple but inefficient query into a more complex, but more efficient form. Transformation is a well-developed technique in the functional programming community [16, 25, 29]. In particular, Freytag has shown how to transform a query evaluation plan into a form that minimises the traversals of the data and the number of conditional expressions [35]. The query evaluation plan is the output of an algebraic relational optimiser, so Freytag's optimisations occur at a lower level than the transformations given in Chapter 9.

Part II

Implementation

Chapter 4

Bulk Data Management

This Chapter covers the implementation of a bulk data manager. It is shown that a functional language can be used to implement efficient operations on trees. An overview of B-trees, a common type of tree in databases, is presented. A bulk data manager that uses trees and supports transactions from multiple users on a shared database is described.

Notation

Program fragments are presented in this and subsequent Chapters. Except where the fragments are in a specific language such as Standard ML, the fragments are not written in an existing language. Instead Bird and Wadler's non-specific syntax is used [17].

4.1 Introduction

A *class* is a homogeneous set of data; it may represent a semantic object such as a relation or an entity set. For example a class might represent a collection of bank accounts. For simplicity the bulk data manager described in this

Chapter supports efficient operations on a single class of data. The same principles apply for operations on a database containing multiple classes of data. Similarly the principles given in Chapter 5 for concurrent transactions against a single class apply equally to transactions against a multiple-class database. Because the same principles apply, both the transactions and the manager are described in terms of a database, although only a single class of data is supported. Chapter 6 gives the design of a more realistic, multiple-class database.

The remainder of this Chapter is structured as follows. Section 4.2 demonstrates that efficient tree-manipulating operations can be implemented in a functional language. Section 4.3 describes the features of B-trees that are significant in this Thesis. Section 4.4 describes the bulk data manager.

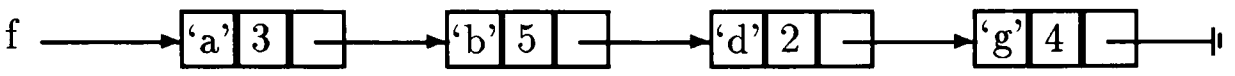
4.2 Bulk Data Structures

4.2.1 Non-destructive Update

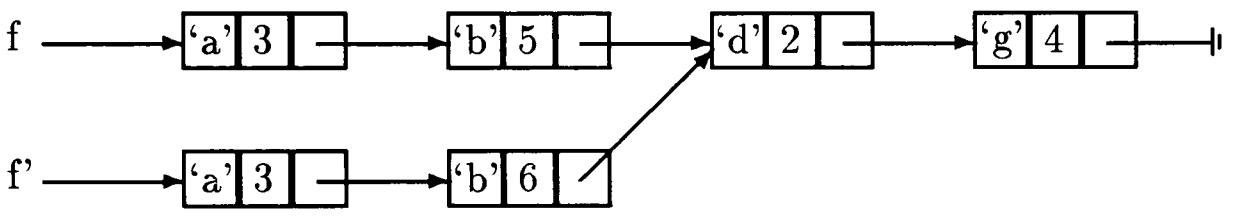
In a persistent environment a class can be represented as a data structure that persists for some time. Because of their size such structures are termed bulk data structures. Operations that do not modify a bulk data structures, for example looking up a value, can be implemented efficiently in a functional language. However, when a data structure is changed in a functional program a new version of the structure must be constructed. It appears to be prohibitively expensive to create a new version of a large data structure every time it is modified.

Updating large persistent data structures is related to the array, or aggregate, update problem. Aggregate update has received considerable attention [51, 68, 77, 94]. The problems are, however, different in several significant respects. Arrays are located entirely in primary memory whereas bulk data structures reside in secondary memory. Array updates are also not assembled into transactions. Because of these differences aggregate update proposals are not discussed further. A summary of their relation to the problem in hand can be found in [84].

It is expensive to construct new versions of many data structures. For example consider representing a class as a list. Fortunately it is not necessary to create a new copy of every element in the list when creating a new version of it. While the new version of the list is logically completely separate from the old version, most implementations allow the old and new versions to *share* the unchanged part. This is best illustrated by an example. Consider the following representation of f, a list of names and values.



Constructing a new list with a value of 6 associated with 'b' gives



On average, when creating a new version of the list, half of it will need to be reconstructed. If the list contains n entities, this gives a time and space cost of $n/2$. Such high modification costs effectively prohibit the use of lists as a bulk data structure in a functional language. Other structures are similarly prohibited and we return to this issue in Chapter 6.

4.2.2 Trees

A new version of a tree can be cheaply constructed. For simplicity a binary tree is considered. A class can be viewed as a collection of entities and there may be a key function that, given an entity, will return its key value. If *et* and *kt* are the entity and key types then an abstract datatype *bdt*, for a tree can be written

$$bdt = Node\ \ bdt\ \ kt\ \ bdt \mid Entity\ \ et$$

Alternately a polymorphic definition parameterised by the entity and key types, α and β, may be used:

$$bdt\ \alpha\ \beta = Node\ (bdt\ \alpha\ \beta)\ \beta\ (bdt\ \alpha\ \beta) \mid Entity\ \alpha.$$

Using one of these definitions, a function to lookup an entity can be written as follows. If the lookup succeeds the result returned is the required entity tagged *Ok*. If the entity does not exist, an *Error* is reported.

$$\begin{aligned} lookup\ k'\ (Entity\ e) &= Ok\ e, \textbf{ if } key\ e = k' \\ &= Error, \textbf{ otherwise} \end{aligned}$$

$$\begin{aligned} lookup\ k'\ (Node\ lt\ k\ rt) &= lookup\ k'\ lt, \textbf{ if } k' \leq k \\ &= lookup\ k'\ rt, \textbf{ otherwise} \end{aligned}$$

A function to update an entity is similar except that, in addition to producing an output message, a new version of the tree is returned.

$$\begin{aligned} update\ e'\ (Entity\ e) &= (Ok\ e,\ Entity\ e'), \textbf{ if } key\ e = key\ e' \\ &= (Error,\ Entity\ e), \textbf{ otherwise} \end{aligned}$$

$$\begin{aligned} update\ e'\ (node\ lt\ k\ rt) &= (m, node\ lt'\ k\ rt), \textbf{ if } key\ e' \leq k \\ &= (m, node\ lt\ k\ rt'), \textbf{ otherwise} \\ &\quad \textbf{where} \\ &\quad\quad (m, lt') = update\ e'\ lt \\ &\quad\quad (m, rt') = update\ e'\ rt \end{aligned}$$

4.2.3 Efficiency

Let us assume that the tree contains n entities and is balanced. In this case its depth is $\log n$ and hence the update function only requires to construct $\log n$ new nodes to create a new version of such a tree. This is because any unchanged nodes are shared between the old and the new versions and thus a new *path* through the tree is all that need be constructed. This is best illustrated by the following diagrams. If the tree depicted in figure 4.1 is updated to associate a value of 3 with x, then the result is depicted in figure 4.2.

Figure 4.1: Original Tree

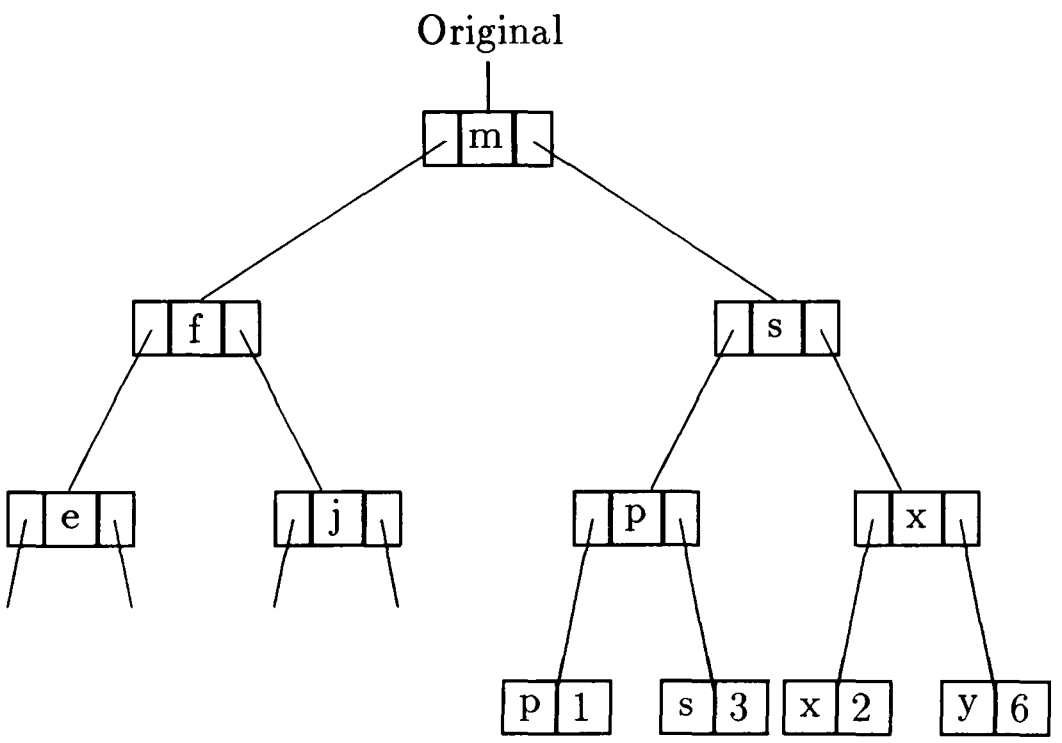
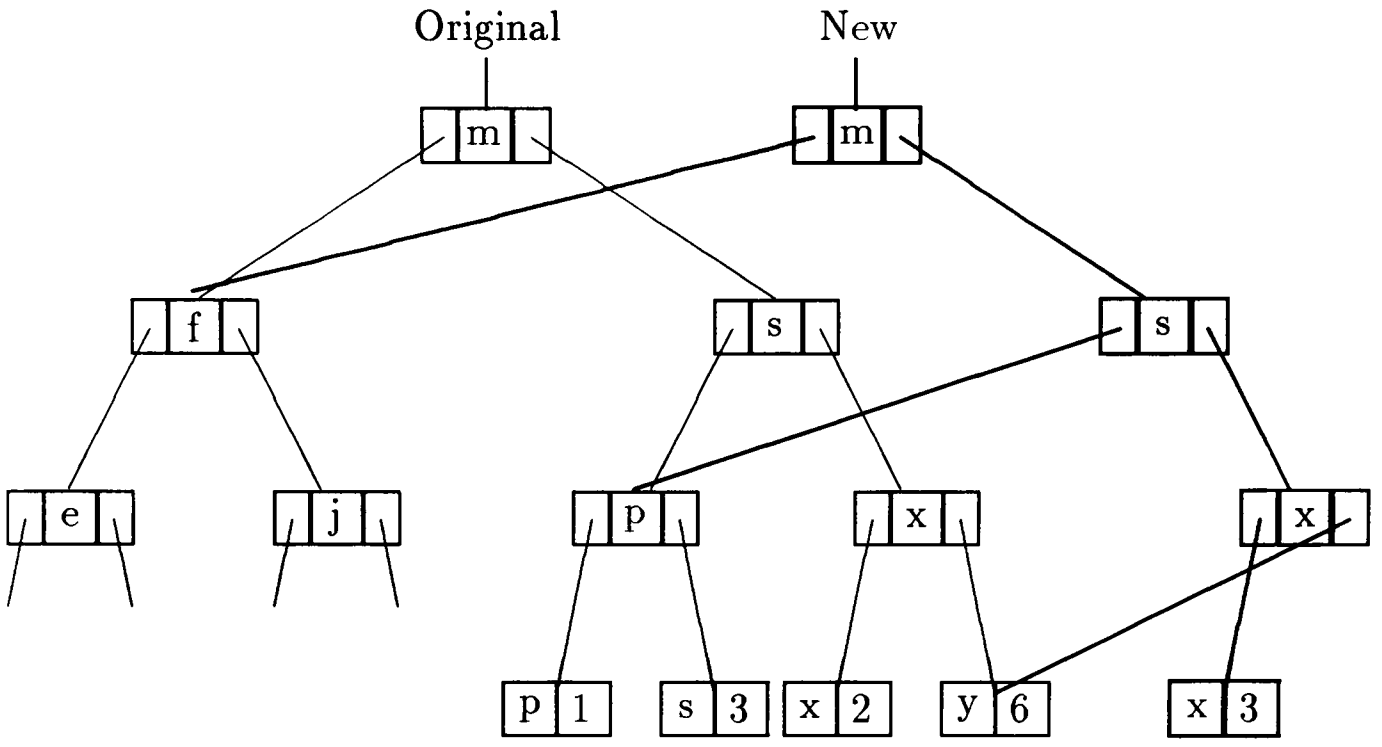


Figure 4.2: Original and New Trees



A time complexity of $\log n$ is the same as an imperative tree update. The non-destructive update has a larger constant factor, however, as the new nodes must be created and some unchanged information copied into them. If the original version of the tree is no longer required the unused nodes will be reclaimed by garbage collector. Hence, although non-destructive update requires the allocation of additional nodes, the total amount of space utilised is the same as under a destructive regime.

The functional update can be made more efficient. A *reference count* is sometimes used in garbage collection to record how many pointers there are to a data structure. This corresponds to how many logical copies of the data structure exist. A reference count of one implies that there is only one copy and hence the original version need not be preserved if the data structure is updated. Some implementations [83] incorporate an optimisation whereby destructive update is used if there is only one reference to the data structure being modified. Clearly this optimisation can be used when the original version of a tree is not required and results in the functional update having

the same time and space requirements as its procedural equivalent.

If non-destructive update is used, a copy of the tree can be kept cheaply because the nodes common to the old and new versions are shared. These cheap multiple versions will be shown to be extremely useful in the following Chapters. To be more precise, retaining the original version after an update requires $\log n$ nodes. As a result, keeping a copy of a tree that has since been updated u times requires no more than $u \log n$ nodes in the worst case. In fact, as updates tend to occur in the same part of the database, or cluster, the average figure is probably well below this.

A more detailed analysis of the time and space costs of bulk data operations will be given once secondary indices have been introduced. The significant points are that when a version is required it is preserved automatically and cheaply. Further, if a version is not required the update can be automatically performed efficiently.

4.3 B-trees

4.3.1 Description

In the foregoing a binary tree was used for simplicity. B-trees [13] are the variant of trees widely used in databases. The distinction between binary and B-trees is not important for the techniques just described. A sketch of the motivations for B-trees, those of their features significant to this thesis and some example figures follow. Full descriptions can be found in [32, 91, 99]. Readers familiar with B-trees and their properties may wish to omit this section.

In the database world the unit of cost is a disk access. This is because the time required for an access is typically three or four orders of magnitude greater than the time required to execute a machine instruction. A disk access retrieves a fixed-sized chunk of data, or *block*, from the disk.

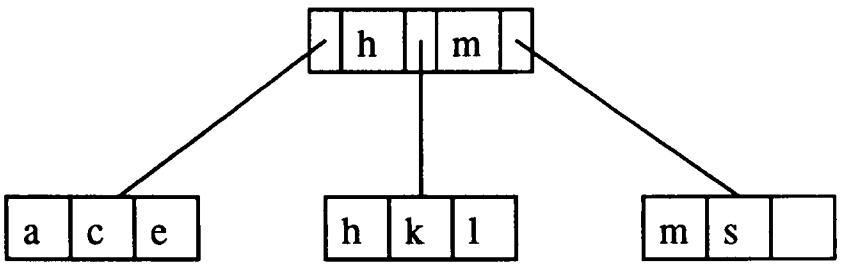
In a binary tree a node contains a key value and two pointers to sub-trees. To

access a node an entire block must be retrieved. Clearly a node may as well fill a block. Thus a node may contain as many keys and associated pointers, or entries, as will fit in a block. The *order* of the tree can be defined as half of the maximum number of keys possible in a node. Similarly, a leaf may contain as many entities as will fit in a block.

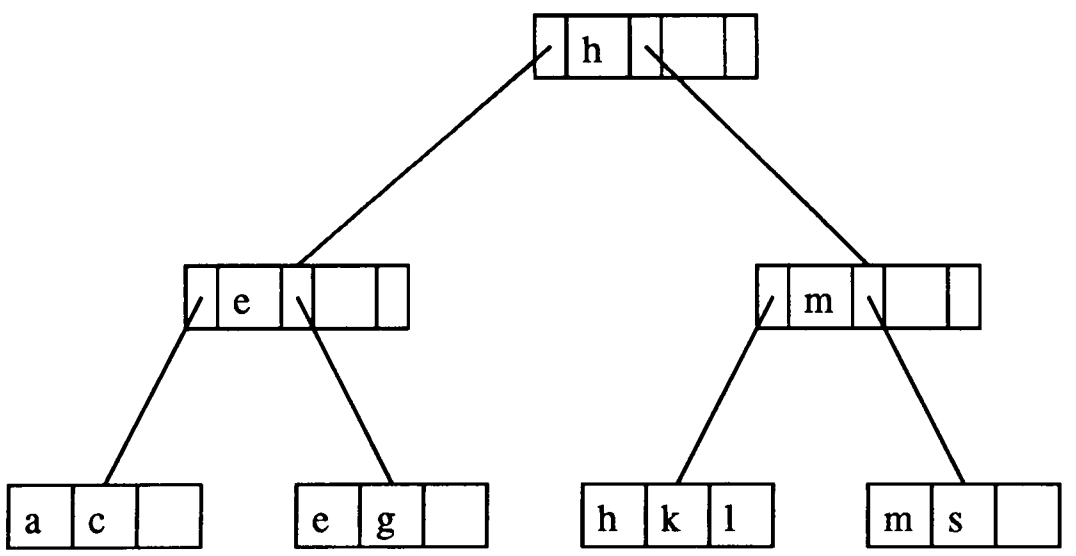
It is desirable that disk space is not wasted. This can be guaranteed by ensuring that every internal node of the tree contains a minimum number of entries and that every leaf contains a minimum number of entries. It is also desirable to give an upper bound on the number of disk accesses required to manipulate any entity. To do this the tree is *balanced*, so that every path from the root to an entity is of the same length. Thus a search or insertion in a B-tree of order m with n entities is guaranteed to require fewer than $\log_m n$ disk accesses. Sedgewick describes this bound as "constant for all practical purposes (as long as m is not small)" [78].

During insertion, the node into which an entry is to be added may be full. In this case a *rotation* is performed. A new node is created, the entries from the original node are split between the two nodes and the new entry is added. An entry for the new node must in turn be added to the original node's parent. In the worst case the root itself will need to be split. A similar process may occur during deletion. This behaviour is best illustrated by an example. Adding an entity with key value 'g' to the following B-tree

Figure 4.3 B-tree Rotation



results in the tree



The following definition, drawn in substance from [38], provides a useful summary. A B-tree of order m is a balanced multiway tree with the following properties.

- Every node has at most $2m + 1$ descendents.
- Every node except the root has at least $m + 1$ descendents.
- The entities all appear at the same depth.

4.3.2 Example

For concreteness let us make some assumptions typical of existing hardware and about some example B-trees. The cost of manipulating these structures can then be calculated. Two example trees are considered, one consisting of a few megabytes and the other of tens of megabytes. An important assumption made is that the top two levels of the tree will be cached. As the root of the tree will be used by every manipulation, and the nodes on the next level level will also be used frequently this is likely to be the case under almost any cacheing strategy.

Nodes		
Sizes in bytes.		
Block-size		512
Key-size		9
Pointer-size		3
Entry-size = Key-size + Pointer-size		12
Order of tree, m = ⌊Block-size/2*Entry-size⌋		21
Cacheing		
Number of levels cached		2
Maximum memory requirements		22k
Tree Sizes	**Example 1**	**Example 2**
Number of entities	10^4	10^5
Entity size in bytes	512	512
Megabytes of data, n	5	50
Depth of tree = $\lceil \log_m n \rceil$	4	6
Disk accesses to manipulate an entity	2	4

4.4 Bulk Data Manager

The preceding sections have shown that a new version of a B-tree can be cheaply constructed. Let us move on to consider a manager function that supports transactions on a bulk data structure.

4.4.1 Transaction Functions

Before the manager can be described transaction functions must be outlined. Transactions are either read-only queries or modifications that update the database. A query transaction can be expressed as a function from the database to a domain of answers. A modifying transaction is a function that takes the existing database and creates a new version. In case the modification fails for some reason it must also return some output. Rather than distinguish between these two types of transaction a transaction is defined to be a function of type $bdt \rightarrow (output \times bdt)$. Let us call this type *txt*.

Transactions are built out of tree manipulating operations such as *lookup* and *update*. A function *isok*, that determines whether an operation succeeded proves useful.

$$isok\ (Ok\ e) = True$$
$$isok\ out = False$$

Another useful function is *dep* that increments the balance of an account entity.

$$dep\ (Ok\ Entity\ ano\ bal)\ n\ =\ Entity\ ano\ (bal + n)$$

Using *isok* and *dep*, a transaction to deposit a sum of money in a bank account can be written as follows.

$$
\begin{aligned}
deposit\ \ a\ \ n\ \ d &= update\ \ (dep\ \ m\ \ n)\ \ d, \textbf{ if } (isok\ \ m) \\
&= (Error, d), \textbf{ otherwise} \\
&\quad \textbf{where} \\
&\quad\quad m = lookup\ \ a\ \ d
\end{aligned}
$$

The arguments the *deposit* function takes are an account number a, a sum of money n and a database d. The database in this case is a simple tree of accounts. If the *lookup* fails to locate the account an error message and the original database are returned. If the *lookup* succeeds, the result of the function is the result of updating the account. The update replaces the existing account entity with an identical entity, except that the balance has been incremented by the specified sum. Note that *deposit* is of the correct type for a transaction function when it is partially applied to an account number and a sum of money, i.e. $deposit\ \ a\ \ n$ has type $bdt \rightarrow (output \times bdt)$.

The *deposit* function has a common transaction form. A test is made on the current contents of the database. If the test is satisfied the transaction proceeds. If the test fails the transaction aborts and the database remains unchanged. It is important to note that, until the test has been performed, it is not known whether the original or the updated database will be returned. Aborting a transaction is easily specified because the original version of the database is available, called d in *deposit*. As described in section 4.2, it is cheap to preserve the original version of the database.

4.4.2 Manager

The bulk data manager is a stream processing function. It consumes a lazy list, or stream, of transaction functions and produces a stream of output. That is, the manager has type $bdt \rightarrow [txt] \rightarrow [output]$. A simple version can be written as follows.

$$
\begin{aligned}
manager\ \ d\ \ (f : fs) &= out : manager\ \ d'\ \ fs \\
&\quad \textbf{where} \\
&\quad\quad (out, d') = f\ \ d
\end{aligned}
$$

The first transaction function f in the input stream is applied to the database and a pair is returned as the result. The output component of the pair is placed in the output stream. The updated database, d', is given as the first argument to the recursive call to the manager. The manager function is partially applied to an initial database d_0 to obtain the stream processing function. Thus the expression *manager* d_0 is of type $[txt] \rightarrow [output]$. Because the manager function retains the modified database produced by each transaction function it has an evolving state.

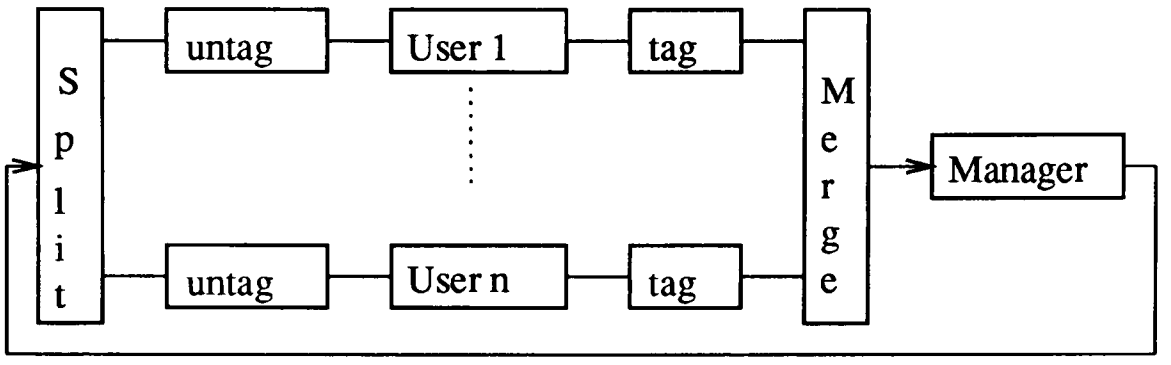
The simplest form of transaction-function that the manager can process is a single bulk-data operation such as *lookup* or *update*. Because these operations manipulate a single entity, a manager processing only bulk-data operations is equivalent to a file manager in a conventional database. The distinction between processing bulk-data operations and transactions becomes significant when concurrency is introduced in the next Chapter.

4.4.3 Multiple Users

The database must be available to many users simultaneously. This allows each user to query and modify the data. A user will also see the effects of other users' actions. The problem of combining asynchronous inputs from many sources has been addressed in work on functional operating systems [49, 50, 83]. A common solution is to employ a variant of Henderson's non-deterministic merge. Some solution of this type can be employed for the bulk data manager.

Figure 4.4 illustrates a possible configuration. A box represents a stream processing function. An arc represents a stream of values. Each user might be a bank teller and the database might be a class of account entities.

Figure 4.4 Multiple Users



A user sends a stream of transaction functions to the bulk data manager. A transaction is tagged to identify the user it originated from. The transactions from all of the users are merged together to form the single input stream to the data manager. The data manager generates some output as a result of applying the transaction function and this is routed back to the appropriate user by the split function. The output is untagged before being returned to the user.

This scheme provides a template for some concurrency. Logically each function in the above diagram could be evaluated on a separate processor. More realistically, a group of closely connected functions will reside on a processor. For example, a user function and associated tag and untag functions might cohabit. The concurrency provided by such a scheme is limited in a significant respect. Because the manager function is executing on only one processor, only one process can be modifying the database at any one time. In the next Chapter we describe how concurrent access to a shared database can be provided.

Chapter 5

Concurrency

This Chapter covers the introduction and control of concurrency within the database. A pseudo-parallel data manager is used to demonstrate concurrent bulk-data operations and concurrent transactions. In contrast to the locking mechanism found in conventional databases the prototype manager uses data dependency as a novel exclusion mechanism. Data dependency is shown to permit an unusual degree of concurrency between transactions. It is also demonstrated that, within certain limits, the rate of processing transactions is independent of the size of the database.

Some problems that severely restrict concurrency are identified and illustrated. Three primitives are proposed to resolve these problems. Two of the primitives are new and one of these, optimistic if, has been implemented. Optimistic if is used to illustrate concurrency both within and between transactions.

5.1 Introduction

It has been known for some time that functional programs have scope for automatic concurrency [24, 31]. Indeed several parallel functional machines are being constructed [3, 22, 28, 69]. Because of its referential transparency

the value of a functional program is independent of evaluation order. For the database manager this means that a transaction can potentially start before the previous transaction has committed, with no risk of interference. Such behaviour contrasts well with the imperative approach where the problem of ensuring that concurrent transactions do not interfere is difficult.

The essence of the concurrency model used in functional languages is as follows. A functional program, such as the database manager, can be viewed as an expression to be evaluated. The expression can be represented as a graph and evaluated by graph reduction [97]. At any stage in the evaluation there may be a number of subexpressions that could be evaluated next. Referential transparency guarantees that every subexpression can be safely evaluated simultaneously. Care is needed to ensure that work is not wasted evaluating expressions that are not required by the program.

The remainder of this Chapter is structured as follows. Section 5.2 describes the machine architecture and the prototype database. Section 5.3 describes the behaviour of concurrent bulk-data operations. Section 5.4 covers transaction processing, identifying some problems that restrict concurrency and illustrating the resolution of the problems using primitives.

5.2 Prototype Database

5.2.1 Hypothetical Machine

The prototype data manager is evaluated on a pseudo-parallel interpreter that emulates a hypothetical machine. The architecture of this machine determines the nature of the parallelism. In technical terms the hypothetical machine is an idealised MIMD super-combinator reduction engine with disk storage.

The memory hierarchy is as follows. It is assumed that the secondary storage underlying the persistent environment is based on disks. It is further assumed that the disk architecture is such that there are elapsed-time savings to be gained by retrieving many nodes in the database simultaneously. A caching

strategy is assumed. To eliminate locality issues the machine is assumed to have a shared primary memory. On current hardware a shared memory limits the number of processing agents to around 100 because of the contention that arises in the memory.

The machine is assumed to be a multiple-instruction multiple-data, or MIMD, machine. Hence each of the processing agents is capable of performing different operations on different data. The machine performs super-combinator graph reduction [53]. That is, the machine evaluates a functional program by evaluating a sequence of simple functions, or combinators. The evaluation strategy used in the machine is lazy except where eagerness is introduced by the primitives described in this Chapter. To preserve laziness the combinators and their arguments are expressed as a graph. In a machine cycle an agent may either

- Perform a super-combinator reduction, or
- Perform a delta-reduction, i.e. evaluate a primitive such as 'plus', or
- Perform a house-keeping activity such as sparking a new task or following an indirection.

The work to be performed by the program is broken into tasks. Each task reduces a subgraph of the program graph. Initially only one task exists. New tasks are sparked by the eager primitives described later. Task synchronisation occurs as follows. A task marks the nodes it is processing as busy. A task encountering a busy node is marked as *blocked.* As soon as the required node is no longer busy the blocked task resumes. A task that is not blocked is termed *active.* The scheduling strategy used in the hypothetical machine is both simple and fair. Every active task is assigned to a processing agent and in a machine cycle the next redex in each active task is reduced.

In a real machine there are a limited number of processing agents. Once all of the agents are being utilised no further concurrency is necessary and a throttling strategy is often employed to reduce the number of new tasks sparked. Throttling is not applied in the hypothetical machine. This is not unrealistic as the number of active tasks in the forthcoming examples is modest.

In many real machines small sub-tasks are not sparked because of the administrative overheads associated with sparking, executing and completing a task. This principle is termed granularity. The grain of parallelism is not controlled in the hypothetical machine :- subtasks are sparked irrespective of their size. This is not unrealistic as most of the tasks sparked in the prototype database represent bulk-data operations such as lookup or update and are of a respectable size.

The hypothetical machine is consistent with existing models of parallelism [31, 70]. It is, however, overly simplistic in several respects. Assuming a uniform machine-cycle is unrealistic as different primitives and different super-combinators take different times to evaluate. Assuming that all active tasks can be reduced in a machine cycle is unrealistic. In a real machine a newly-sparked task must be migrated to an idle processing agent.

5.2.2 Metrics

The hypothetical machine is instrumented to record significant information during the evaluation of a program. The metrics used are the number of super-combinator and delta- reductions, the number of graph nodes allocated, the number of machine cycles and the average number of active processes. The number of super-combinator and delta- reductions is a measure of the time-complexity of the problem. The number of graph nodes allocated is a measure of a program's memory usage. Under the assumption that machine cycles take constant time, the number of machine cycles is a measure of the elapsed-time taken to evaluate a program. The average number of active processes gives the average concurrency available during a program's evaluation.

In addition to the above figures, every 10 machine cycles the average number of active tasks during those cycles is recorded. This information can be used to plot a graph of the average number of active tasks against time, measured in machine cycles. For clarity every point on these graphs is not plotted. Instead a simplified graph giving the significant features of the evaluation is given. To demonstrate the correspondence between the abstract graph and the detailed data, the first two active task graphs also plot the average

number of active tasks every 50 cycles.

5.2.3 Database Architecture

The database used to demonstrate parallelism represents a single class of account entities. Each account entity has an account number, a balance, a class and a credit limit. There are 512 account entities and each entity occupies 14 bytes, giving a total of 7 Kbytes. To make the analysis of concurrency simple the account entities are stored in a binary tree. With the exception of rebalancing, the concurrency possible in such a tree is similar to that obtainable in a B-tree in which each of the nodes contains a small binary tree.

The database resides entirely in primary memory. Whilst this data structure is small it is sufficient to demonstrate the behaviour of the data manager. Furthermore, Section 5.3.4 shows that, while all of the database remains in primary memory, the rate that bulk-data operations can be processed is independent of the size of the database.

5.3 Bulk-Data Operations

In this Section the concurrency between bulk-data operations such as *lookup* and *update* is demonstrated. In contrast to transactions that may manipulate several entities, bulk-data operations manipulate a single entity.

5.3.1 Potential

A purely lazy, or demand-driven, evaluation of bulk-data operations does not lead to any concurrency. The operations are performed serially because there is only a single source of demand, or task. It is reasonable to assume that the result of all of the operations will be demanded. Thus a task can be sparked to evaluate a subsequent operation before the current operation

has completed. This effect can be achieved using an eager constructor. An eager list constructor sparks tasks to evaluate both the head and the tail of the list concurrently. Consider the manager from Section 4.4.2:

$$\begin{aligned} manager\ d\ (f : fs)\ &=\ out : manager\ d'\ fs \\ &\quad \textbf{where} \\ &\quad\quad (out, d') = f\ d \end{aligned}$$

To introduce concurrency an eager list constructor is used to create the output list. As a result a task is sparked to evaluate the output of the current operation and another is sparked to evaluate the manager applied to the subsequent transactions.

Some form of exclusion is necessary to prevent concurrent transactions from interfering, as outlined in Section 3.2.1. For simplicity exclusion between bulk-data operations is described. The principle remains the same for transactions. Suppose an update is creating a new version of the database, and a lookup is directed at the entity being updated. The lookup cannot be allowed to read the new data until the update has finished with it. A common imperative solution is for a transaction to lock all of the nodes it may change, so denying access to other transactions until the original transaction is complete.

Exclusion occurs within the manager as a result of *data dependency*. Recall that update constructs a new version of the database. Until a node in the new version exists no other function can read its contents. Any task demanding, or depending on, a node that is being constructed is blocked until the node becomes available. Once the required data is available the demanding process is reactivated. Both blocking and reactivation occur automatically within the parallel evaluator. Clearly the lookup can consume each node of the new version of the database as it is produced by the update. A discussion of the properties of data dependency as an exclusion mechanism is given in Section 5.3.5.

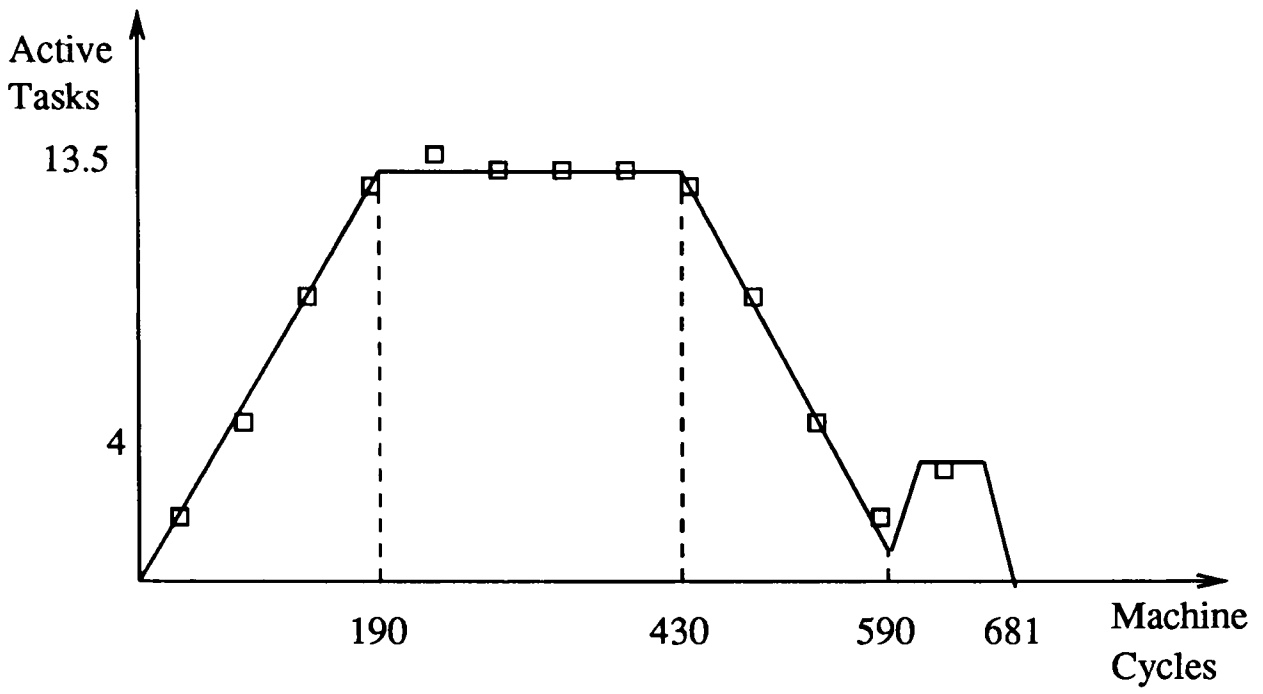
5.3.2 Lookups

Consider the manager processing a sequence of lookups directed to entities throughout the structure. There is no data-dependency between the lookups and hence one lookup never excludes another. The eager output-list constructor in the manager sparks tasks to perform the first lookup and to apply the manager to the remaining lookups. Because the first lookup does not change the database it is immediately available for processing the second lookup. On encountering each of the remaining lookups the manager will spark a task to perform the lookup and another to evaluate the manager against the remaining input stream.

There is no limit to the number of tasks that can be sparked in this way. Hence a fair scheduler is required to ensure that, not only are new lookups sparked, but those already in the database make progress. The scheduling strategy employed in the hypothetical machine is fair because it attempts to perform a reduction in every active task in each machine cycle. As a consequence the first lookup will complete at some point. From this point onwards earlier lookup-tasks will complete at the same rate as the manager sparks new ones. The manager has reached a state of dynamic equilibrium. If the input stream is finite, then, once the last lookup has been sparked, the number of active processes will decline as the earlier lookup-tasks complete.

This behaviour is exhibited by the manager processing a sequence of 30 lookups directed to different entities. Appendix A contains the LML programs for each of the examples in this Chapter. The bulk data manager is in Appendix A.1.1 and the program invoking 30 lookups is in Appendix A.2.1. The active task graph for the eager evaluation of this program is as follows.

Figure 5.1 30 Lookups



The first lookup completes after approximately 190 cycles and the maximum concurrency is reached at this point. The average of 13.5 active tasks represent the manager and 12.5 lookup-tasks. After the last lookup-task is sparked at approximately 430 cycles the concurrency declines steadily until the output phase is entered at cycle 590. The manager function requires 430 machine cycles to spark all 30 tasks, indicating that it requires approximately 14 cycles to spark a new task. Note that the number of active tasks depends on this delay and the number of machine cycles each task takes to complete. The small peak between cycles 590 and 681 represents an output phase. The printing of the value of the first lookup sparks the evaluation of all of the lookups, and only when there are no more tasks does printing proceed. The metrics obtained during the evaluation of the program are as follows.

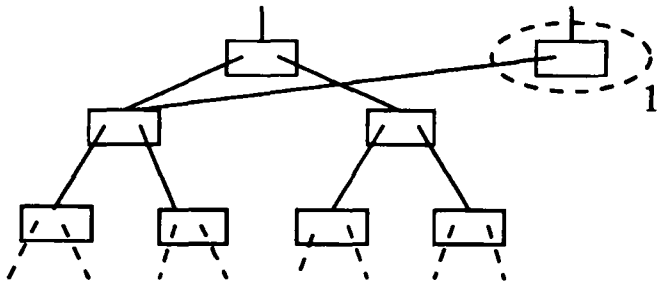
30 LOOKUPS

Metric	Lazy	Eager
Number of super-combinator reductions	1923	1923
Number of delta-reductions	1573	1573
Number of graph nodes allocated	32366	32366
Number of machine cycles	5734	681
Average number of active tasks	1.04	8.82

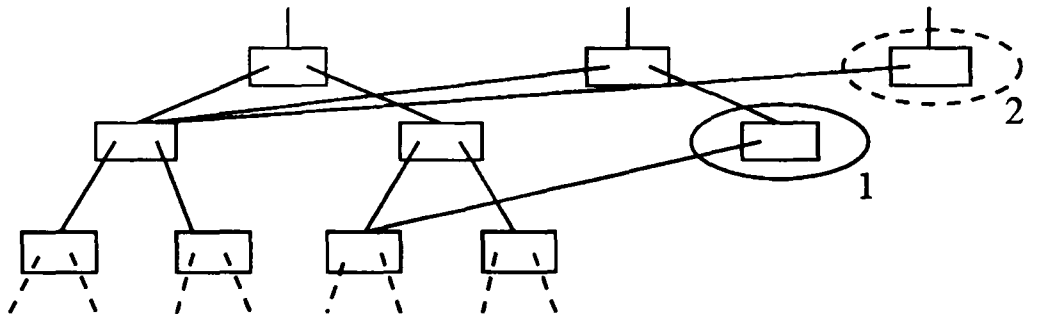
Note that the average concurrency during the lazy evaluation of the program is not 1.00. This is because a strict primitive in the lazy program will spark a sub-task and for a brief period both parent and child tasks are active before the parent task discovers that it is blocked. To compensate for this calibration error the average concurrency in the eager evaluation can be divided by the lazy average concurrency to give an *adjusted average concurrency* of 8.48 active tasks. Note that the elapsed-time to evaluate the eager program has been reduced by a factor of 8.42. The *elapsed time reduction factor* is reassuringly close to the adjusted average concurrency, indicating that the additional tasks are reducing the elapsed time by performing useful work.

5.3.3 Updates

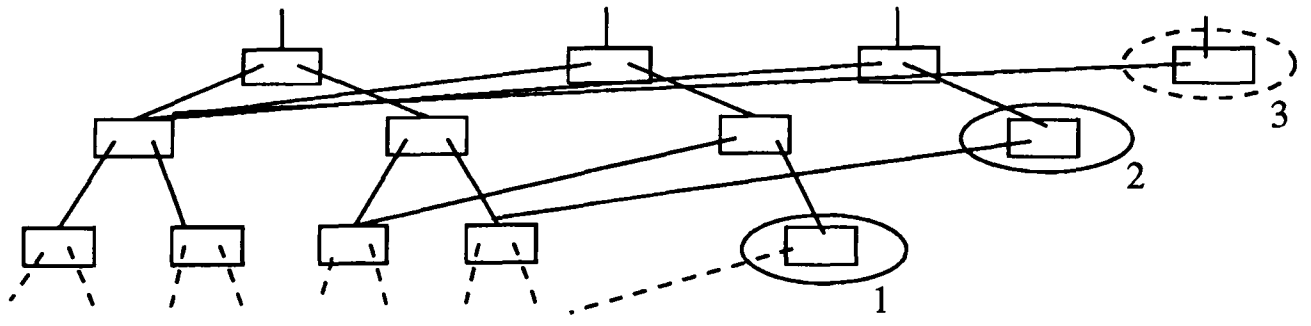
Not all of the parallelism is unbounded. Consider a stream of updates directed at the same entity. Each update must wait until the path in the tree being created by the preceding update exists. As a result the parallelism is bounded by the depth of the tree. This is illustrated by the following diagrams. Initially the first update-task has control of the original root and is constructing a new one.



Once the first update has constructed a new root the second update is activated.



Similarly, once the second update has constructed a new root, the third update is reactivated.



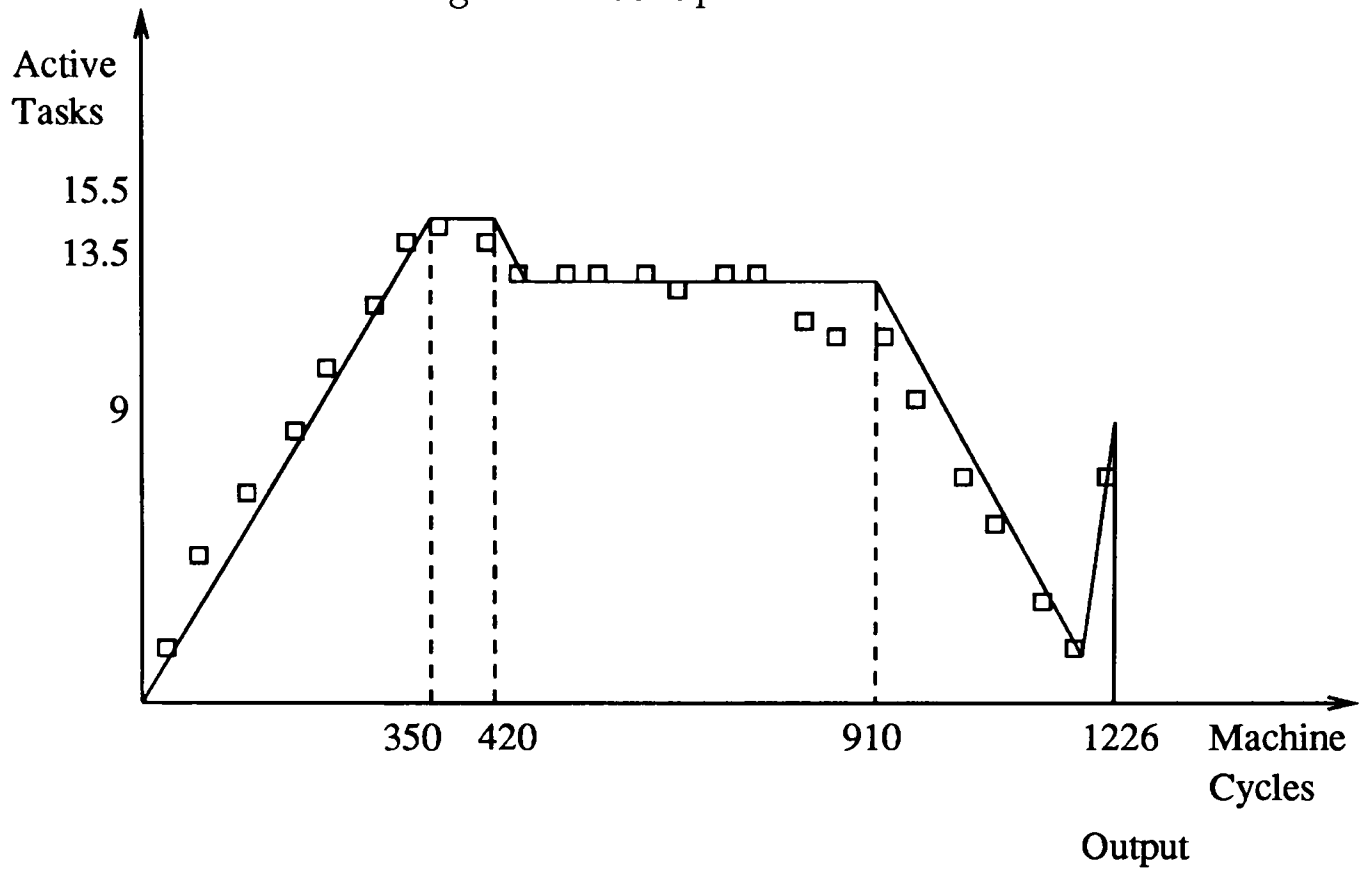
Note how a tree-node just created by an update is immediately consumed

by the following update. This is pipelined behaviour. In fact the pipeline is slightly more complex than figure 5.1 indicates because the tree constructor releases a node once it is in weak head normal form, i.e. before the sub-trees are complete. As a consequence two tasks can perform useful work at each level in the tree :- one selecting a subtree to update and the other constructing a new sub-tree. Because these two activities take different times, there will not always be two active tasks at each level in the tree. For example, as the account tree has 10 levels, there can be at most 20 active tasks in it. The manager will also be active, giving a maximum of 21 active tasks. In the following example the maximum number of active tasks recorded is 19.

It is undesirable to allow closures that may be erroneous to persist, as outlined in Section 3.2.3. The eager output-list constructor forces most of the work of an update to be performed, as the output required is only obtained once the update reaches the entity at the leaf of the tree. Updates can be made slightly more eager using a tree-constructor that eagerly constructs both of its sub-trees. This forces the update to be performed on the entity found at the leaf of the tree.

The following statistics are obtained when 30 updates are directed to the same entity. The LML program is in Appendix A.2.2

Figure 5.2 30 Updates



Maximum concurrency is reached after approximately 350 cycles, when the first update completes. At this point the pipeline described above is full, the manager is active and there may be an update that has been sparked but has not yet demanded the root of the tree, i.e. entered the pipeline. Recall that the manager requires approximately 14 cycles to spark a new operation. By cycle 420 the manager has sparked all 30 of the updates and the number of active processes drops at this point. The pipeline remains full until cycle 910. The later update-tasks, although sparked, are blocked waiting for the preceding updates to create a new root. No more tasks are sparked to replace the completing tasks once the last update gains control of the root at cycle 910 and the parallelism declines as before. Finally there is a brief output phase.

The fact that the 30th update does not gain control of the root until cycle 910 shows that constructing a new root requires approximately 30 machine cycles. For updates the construction time is the bottleneck, rather than the

time the manager requires to spark a new task. The effect of this bottleneck on a more realistic mix of operations is investigated in Section 5.3.6. Note that while inspection has revealed the small peak between cycles 350 and 420, a dip of similar size between cycles 820 and 910 has not been detected.

30 UPDATES

Metric	Lazy	Eager
Number of super-combinator reductions	2973	2973
Number of delta-reductions	2783	2792
Number of graph nodes allocated	46515	46524
Number of machine cycles	10373	1226
Average number of active tasks	1.15	10.34

Elapsed-time reduction factor 8.46.
Adjusted average concurrency 8.99 tasks.

The updates require more work in total than the lookups because each one is constructing a new version of the database. The eager manager constructs entities that are not demanded and hence does slightly more work and uses slightly more memory than the lazy manager. Under the assumption that all of the database will ultimately be realised this work is not wasted. Again the adjusted average concurrency and elapsed-time reduction factors are reassuringly close. The discrepancy between these two statistics is consistent for all programs: the elapsed-time reduction factor is always less than the adjusted average concurrency.

5.3.4 Effect of Database Size

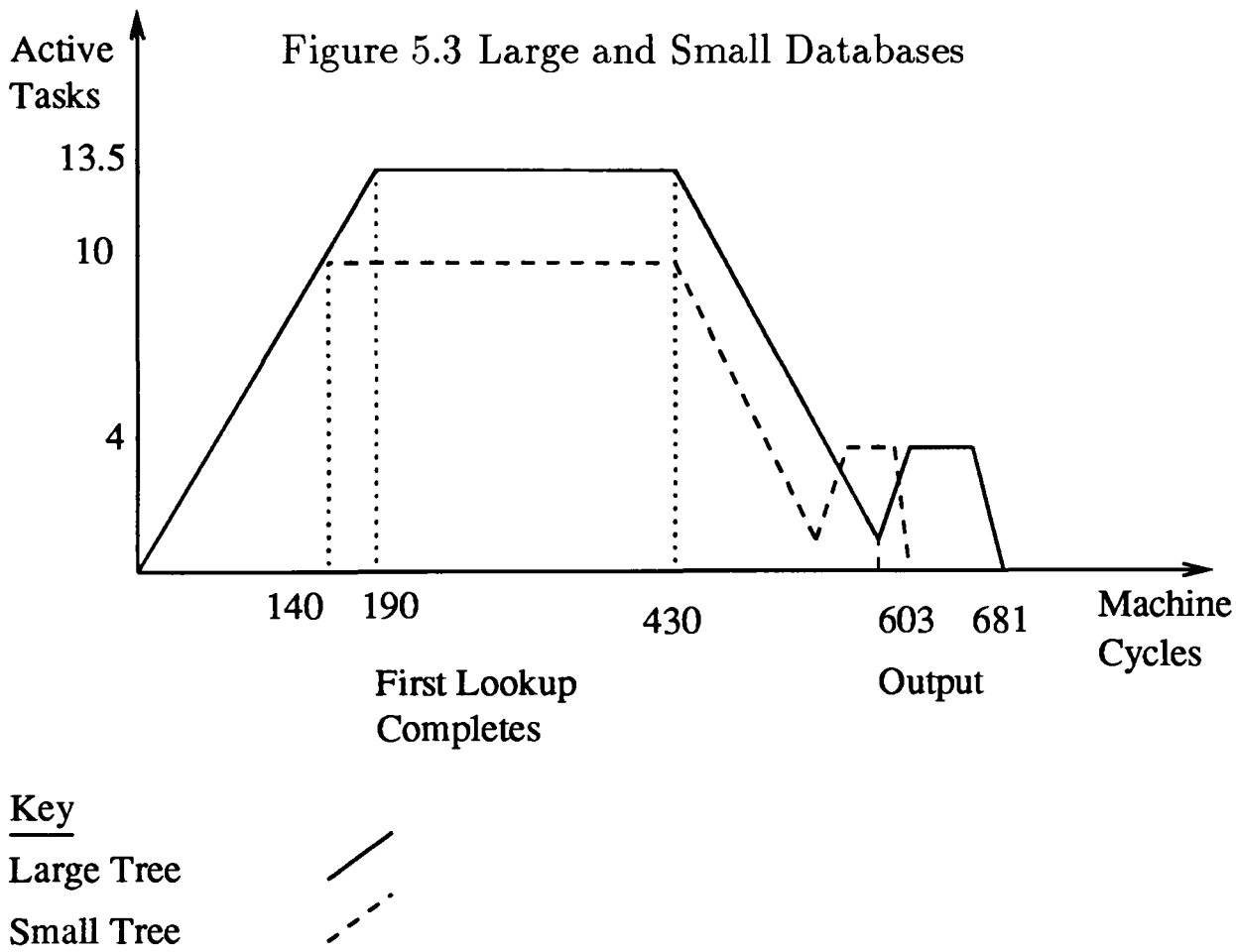
Primary Memory

If all of the database resides in primary memory the time to complete an operation is the log of the database size. Hence an operation on a larger database takes longer to complete. As a result more tasks are able to pass the bottleneck at the root before the first completes and greater concurrency is possible. As in a smaller database, once the first operation completes,

earlier operations will complete at the same rate as they pass the bottleneck. In consequence the manager delivers the results of operations on the larger database at the same rate as on a smaller database.

For example, consider a database only an eighth of the size of the account database. Such a database-tree has only 7 levels and contains only 64 entities. The following graph plots the active processes when the stream of 30 lookups from Section 5.3.2 is consumed by two instances of the manager. One instance processes the lookups against the large database and the other against the small database.

Figure 5.3 Large and Small Databases



Note that the first lookup in the larger database takes longer to complete and hence the time to reach maximum concurrency is greater. This larger set-up time results in a small 13% increase in elapsed time. Also note that the last lookup starts at the same cycle in both instances. This fact and the small increase in elapsed-time confirm the expectation that operations on a

large database occur at the same rate as in a small database.

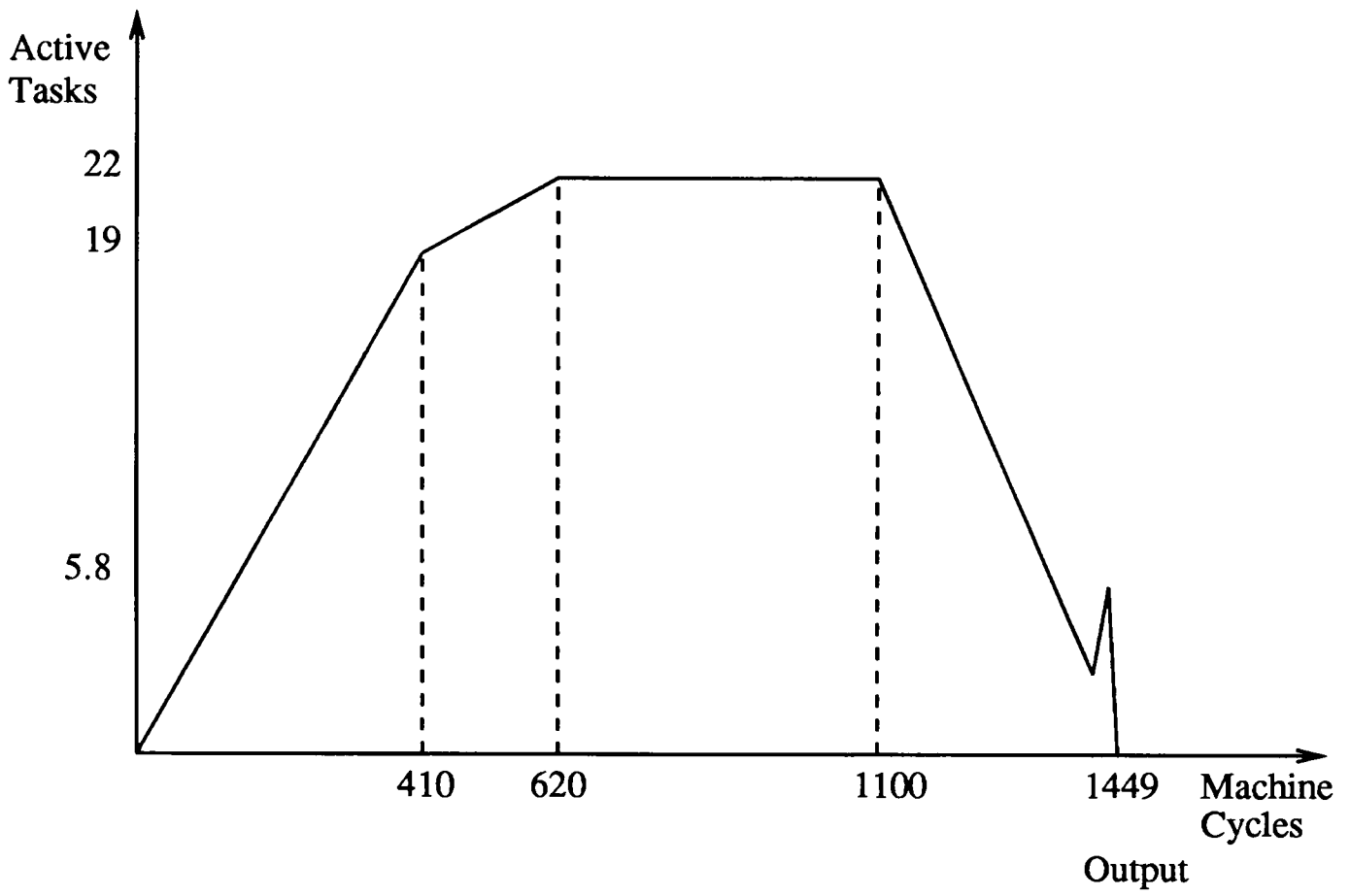
Secondary Storage

If the lower levels of the database-tree are in secondary storage then the time to perform an operation is greatly increased because of the disk access delay. The time to perform a disk access is typically three orders of magnitude greater than the time required to perform a machine instruction. As a result on the order of hundreds of tasks can pass the bottleneck at the root in the time taken for the first operation to complete.

If the operations depend on each other it is reasonable to assume that the path to any shared entity is preserved in cache. In this case the first operation retrieves the path into cache and subsequent operations occur in relatively high-speed primary memory. If the operations are independent then multiple disk-accesses can occur concurrently.

The effect of disk accesses is difficult to demonstrate in the prototype manager because all of the database-tree resides in memory. To simulate the effect of a disk access to retrieve the leaves of the tree a delay function has been added to the lookup and update operations so that they wait for approximately 750 cycles on demanding a leaf. The update operations are made slightly more eager to force them to perform the delay-function even when the value of the updated entity is not demanded. The additional concurrency this introduces is apparent in the average concurrency for the 'lazy' version of the program. When only 15 updates with a disk-delay are directed to different entities in the database the following information is recorded. The LML program for the updates is in Appendix A.2.3, and the bulk data manager with a simulated disk delay is in Appendix A.1.2.

Figure 5.4 15 Updates with Disk Delay



15 UPDATES WITH DISK DELAY

Metric	Lazy	Eager
Number of super-combinator reductions	6123	6123
Number of delta-reductions	5917	6167
Number of graph nodes allocated	70294	70453
Number of machine cycles	15942	1449
Average number of active tasks	1.40	15.89

Elapsed-time reduction factor 11.00.
Adjusted average concurrency 11.35.

The 22 active tasks represent 15 'disk accesses', the output of each update being eagerly constructed and some calibration error.

5.3.5 Data Dependent Exclusion

Data dependency has several desirable properties as an exclusion mechanism. A fuller investigation of data dependent exclusion would be worthwhile and might include a comparison with the conventional methods of locking, optimistic concurrency control and time-stamping. Recall from Subsection 5.3.1 that a process depending on data written by another is suspended until the data it requires exists. In contrast to locking, optimistic concurrency control and time-stamping, the synchronisation occurs within the parallel evaluator and no additional mechanism is required.

Some general properties of data dependent exclusion are as follows. Deadlock is avoided because the manager function allows only one process to have control of the database at a time. That is, one function is applied to a version of the database at a time. Data dependent exclusion has variable granularity. Any part of the database can be 'locked', the entire database, a class of data, an entity, or a field within an entity. The part 'locked' is just that part being constructed by the current transaction function. If the part being constructed is itself a structure, then pipelining occurs automatically between the constructor and a consumer, just as tree nodes can be consumed as they are produced. This property could be useful to support Unix-like byte-stream files.

Data dependent exclusion allows an unusual degree of concurrency between read- and write-transactions. This concurrency is facilitated by the multiple database versions generated under a non-destructive update regime. Consider two transactions that appear adjacent in the manager's input stream. The second transaction is said to *overtake* if, although it is applied to the database after the first, it may complete earlier in real time.

All transactions depend on the preceding write-transaction for at least the root node, and possibly other internal nodes. However, because the entities at the leaves are most likely to be in secondary storage, the dependence between the entities accessed by two transactions is significant. Clearly a transaction that does not read or write any entity read or written by a preceding transaction does not depend on the entities of the earlier transaction and can overtake. Let us therefore only consider read- and write-transactions that

read and write the *same* entity. For simplicity the transactions are taken to be a single lookup and a single update. To emphasise the effect of exclusion, lookup and update operations with a 'disk delay' are used and the effects of caching are ignored. The LML programs for the following four pairs of transactions can be found in Appendix A.2.4.

Examples using larger transactions can be found in Section 7.3.1. It is important to note that a write-transaction only writes entities, it does not read them beforehand. This sort of write-transaction is useful if the new value of the entity does not depend on the existing value. For example, such write-only transactions might be used to maintain a class of personal identity numbers (pins).

Write-Transaction followed by a Read-transaction

In Section 5.3.1 data dependency was shown to prevent a read-transaction from overtaking a write-transaction. Consider an update of an entity followed by a lookup of the same entity. The eager manager introduces some concurrency because the lookup can consume the new path in the tree that the update is creating. However, once the leaf of the tree is reached, the lookup becomes blocked until the update has performed the 'disk access' to create the new entity. Only once the entity exists is the lookup reactivated to perform its 750 cycle 'disk access'. As a result the update completes after 1070 machine cycles and the lookup after 1850 cycles. Recall that the lazy version is made slightly eager to force the disk-access delay to occur. The full statistics are as follows.

Metric	Lazy	Eager
Number of super-combinator reductions	788	788
Number of delta-reductions	767	777
Number of graph nodes allocated	20643	20655
Number of machine cycles	2044	1850
Average number of active tasks	1.39	1.55

Elapsed-time reduction factor 1.10.
Adjusted average concurrency 1.12.

Read-Transaction followed by a Write-transaction

In contrast, a write-transaction can overtake another read-transaction because there is no data dependency between them. As the database is unchanged by the read-transaction it is immediately available for processing by the write-transaction. The write-transaction can construct a new version of the database without disrupting the preceding read-transaction which is proceeding on its own version of the database. Although in real time the read transaction may complete after the write-transaction, logically it occurred first, i.e. on an earlier version of the database.

When the manager processes a lookup followed by an update the following statistics are obtained. The reduction in elapsed time and degree of concurrency indicate that the lookup and update occur concurrently.

Metric	Lazy	Eager
Number of super-combinator reductions	788	788
Number of delta-reductions	750	750
Number of graph nodes allocated	20637	20637
Number of machine cycles	2016	1070
Average number of active tasks	1.39	2.62

Elapsed-time reduction factor 1.88.
Adjusted average concurrency 1.88 tasks.

Read-Transaction followed by a Read-transaction

It is not surprising that a read-transaction can overtake a preceding read transaction. As before, the database is immediately available for processing by the following read-transaction. When the manager processes two lookups the following statistics are obtained. Again the reduction in elapsed time and degree of concurrency indicate that the lookups occur concurrently.

Metric	Lazy	Eager
Number of super-combinator reductions	757	757
Number of delta-reductions	723	723
Number of graph nodes allocated	20220	20220
Number of machine cycles	1913	967
Average number of active tasks	1.38	2.73

Elapsed-time reduction factor 1.98.
Adjusted average concurrency 1.98 tasks.

Write-Transaction followed by a Write-transaction

Most unusually a write-transaction can overtake another write-transaction. Recall that the write-transactions only write entities, they do not both read and write. As described in Section 5.3.3, an update following an earlier update to the same entity must wait until the new path in the database being created by the preceding update exists. However, once the entity at the leaf has been located, both updates can independently construct a new version. Overtaking at the leaves is significant because they are the part of the database most likely to be on secondary storage. The entity written by the first update will not be visible to transactions after the second update. It will, however be visible to any lookups between the first and second update.

When the manager processes two updates directed to the same entity the following results are obtained. As before, the reduction in elapsed time and degree of concurrency indicate that the updates occur concurrently.

Metric	Lazy	Eager
Number of super-combinator reductions	819	819
Number of delta-reductions	794	804
Number of graph nodes allocated	21062	21072
Number of machine cycles	2147	1129
Average number of active tasks	1.39	2.69

Elapsed-time reduction factor 1.90.
Adjusted average concurrency 1.94 tasks.

Summary

In summary, let us compare the concurrency between read- and write-transactions permitted by data-dependency with that permitted by conventional locking schemes. The following table gives the concurrency permitted by each scheme

Concurrency Permitted	Locking	Data-Dependency
Read followed by Read	Y	Y
Read followed by Write	N	Y
Write followed by Read	N	N
Write followed by Write	N	Y

However a conventional write-lock permits the process in possession of the lock to both read and write the entity locked. A read immediately followed by a write is a common sequence. This effect is achieved in the data manager by constructing a transaction-function that performs both a lookup and an update. For example the bank *deposit* transaction has this form. This entails following the path in the database to the entity twice. A more efficient solution is to introduce a new bulk-data operation that follows the path to the entity only once, and then *replaces* the entity with a function of itself.

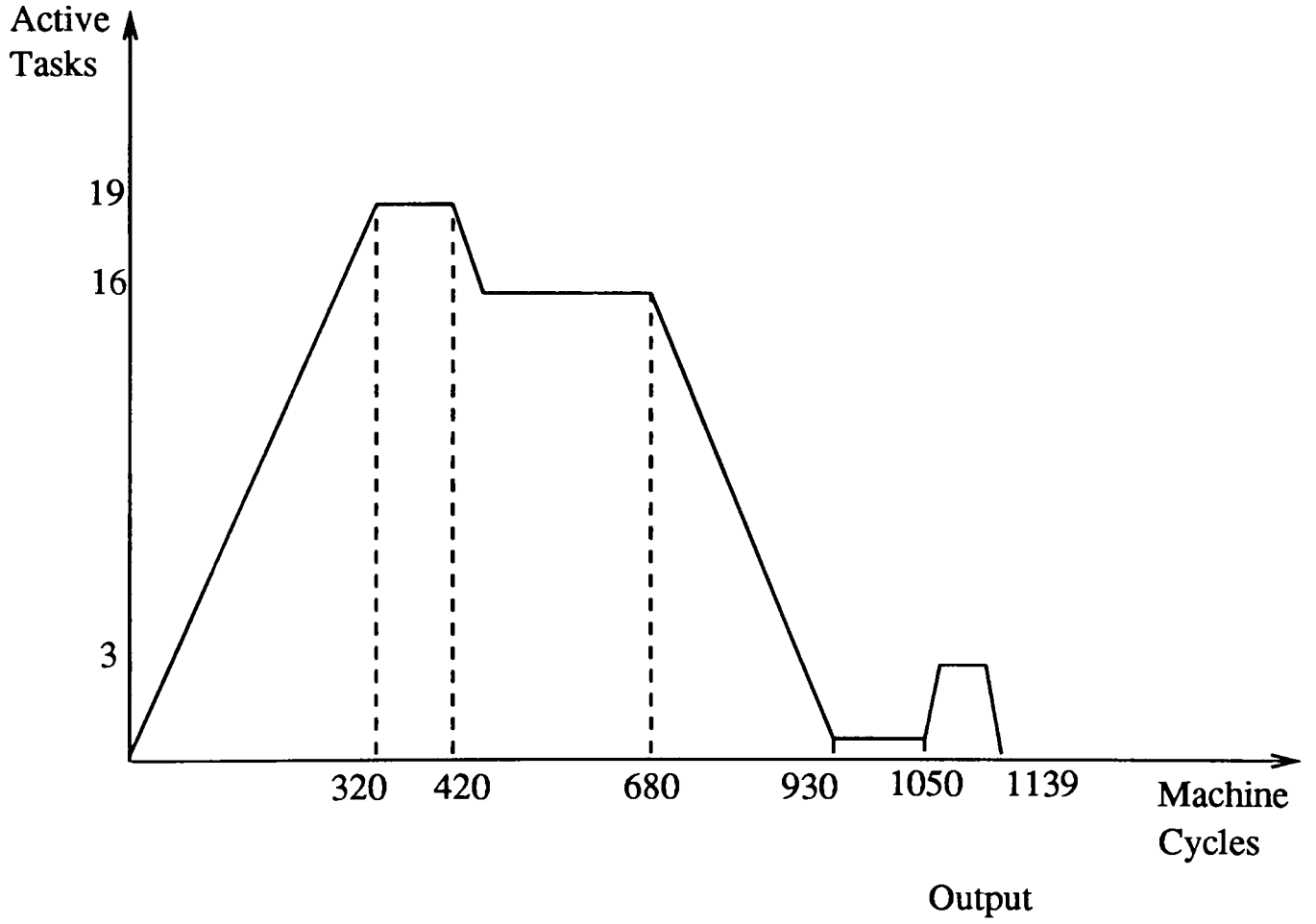
Data dependency permits greater concurrency between lookups and replace operations than a locking scheme does. This is because a replace operation can inspect the original version of the entity and construct a new version of it without disturbing a lookup that is proceeding on the original version. An important use of this concurrency is given in Section 7.3.1. The concurrency between replace and lookup operations permitted by locking and data-dependency is summarised in the table below.

Concurrency Permitted	Locking	Data-Dependency
Read followed by Read	Y	Y
Read followed by Replace	N	Y
Replace followed by Read	N	N
Replace followed by Replace	N	N

5.3.6 Typical Mix

The preceding Sections analyse the behaviour of sequences of lookups, sequences of updates and the interaction between lookups and updates. In practice a bulk-data manager will process a mixture of operations such as inserts, deletes, lookups and updates. The exact composition of the mixture will depend on the nature of the applications the manager is supporting. The statistics below are collected when the manager processes a mix of 30 inserts, deletes, lookups and updates directed to different entities throughout the database. The mixture, given in Appendix A.2.5, contains 11 lookups, 10 updates, 5 inserts and 4 deletes.

Figure 5.5 Typical Mix



MIXTURE OF 30 OPERATIONS

Metric	Lazy	Eager
Number of super-combinator reductions	2814	2814
Number of delta-reductions	2397	2564
Number of graph nodes allocated	44363	44470
Number of machine cycles	8951	1139
Average number of active tasks	1.14	9.74

Elapsed-time reduction factor 7.86.
Adjusted average concurrency 8.54 tasks.

Let us use this mix to make a tentative estimate of the effect of the bottleneck in a real concurrent machine. Let us assume that all of the database resides in primary memory. In the above example the 30th operation gains control of the root after 680 machine cycles and hence the average time for each operation to clear the bottleneck is approximately 23 machine cycles. If each transaction comprises 10 operations this gives a total delay of 230 cycles per transaction.

A machine cycle takes the same length of time as a super-combinator reduction because an agent in the hypothetical machine may perform a super-combinator reduction in a cycle. The rate that super-combinators can be reduced depends primarily on the underlying hardware, the compiler technology and the size of the combinator. With good compiler technology, on Motorola 68030 hardware (e.g. a Sun 3/50), approximately 50000 super-combinators can be reduced per second. If each processing agent in the hypothetical machine has this processing speed then the manager will process in the order of 10^2 transactions per second. This rate compares favourably with the transaction processing rates reported for existing machines with imperative storage summarised in Section 3.2.3 [75].

5.4 Transaction Processing

5.4.1 Drawbacks

There are some problems with the bulk data management methods proposed in Chapter 4 that make the parallelism obtainable using just eager constructors inadequate. Maintaining balance in the database-tree restricts concurrency. As demonstrated in Subsection 4.3.1, insertion may require rotations anywhere along the path to the inserted record. Usually such rotations occur deep in the tree, near the point of insertion. Occasionally, however, the root of the tree is rotated. It is only possible to ascertain whether or not the root need rotation after all the other rotations have taken place. As a result, an insert operation must retain control of the root throughout its execution, preventing any other concurrent operations.

A similar problem arises with total transactions, such as *deposit* from Subsection 4.4.1. A total transaction retains control of the database throughout its execution. This is because neither the original nor the replacement database can be returned until the decision whether to abort or not has been taken. Therefore no other transaction may access any other part of the database until this decision has been made.

To illustrate the occurrence of these problems, consider the manager consuming five transactions: three balance enquiries and two deposits. Some parallelism is possible because the balance enquiries, or lookups, can be overtaken. However, the total deposit transactions prevent subsequent transactions from starting until they have committed or aborted. The resulting serialisation of the operations is demonstrated by comparing the evaluation of the transactions with the evaluation of a sequence of lookups and updates that perform the same operations as the transactions, but without being packaged into transactions with a commit/abort predicate. The LML programs are given in Appendix A.3.1.

FIVE BANK TRANSACTIONS

Metric	Individual Operations	(Eager) Transactions
Number of super-combinator reductions	521	559
Number of delta-reductions	478	504
Number of graph nodes allocated	18807	19243
Number of machine cycles	428	1216
Average number of active tasks	4.43	1.67

The total transactions reduce the concurrency by a factor of 2.65. The following Sections outline and demonstrate methods of overcoming the concurrency restrictions.

5.4.2 Optimistic If

Description

Often total transaction functions have the form

if predicate db *then* transform db *else* db.

In many applications the predicate will usually be true, allowing the transaction to commit. In rarer cases the predicate will be false and the transaction will abort. Normally the predicate is evaluated and only when its value is known is one of the branches selected for evaluation. Advantage can be taken of the supposition that the *then* branch is most likely to be chosen by starting evaluation of the *then* branch immediately.

A new speculative parallelism primitive called optimistic if or *optif* is proposed. When *optif* is evaluated both the predicate and the *then* branch are evaluated. When, as in most cases, the predicate eventually evaluates to true, the evaluation of the *then* branch is well on the way to completion. If the predicate is false, evaluation of the *else* branch is started as usual and evaluation of the *then* branch is arrested.

The only difference between *optif* and *if* is operational; they have identical denotational semantics. The advantage gained from using *optif* is that in most cases concurrency has been increased and the elapsed time to evaluate a total transaction has been reduced. The disadvantages are common to speculative primitives. When the *else* branch is selected, it may be difficult to kill processes in the *then* branch. Unnecessary computation may also be performed in some cases, but this is not serious as these cases are assumed to be rare. Also, an implementation may give speculative processes a low priority, to be performed if there are spare processors but not otherwise.

It may appear desirable to evaluate both branches of a conditional. This is not recommended as the number of speculative processes would be exponential in the depth of nesting of optimistic ifs. Because *optif* only evaluates one branch, the likely one, the number of speculative processes is linear in the depth of nesting. As a result the machine is less likely to be swamped.

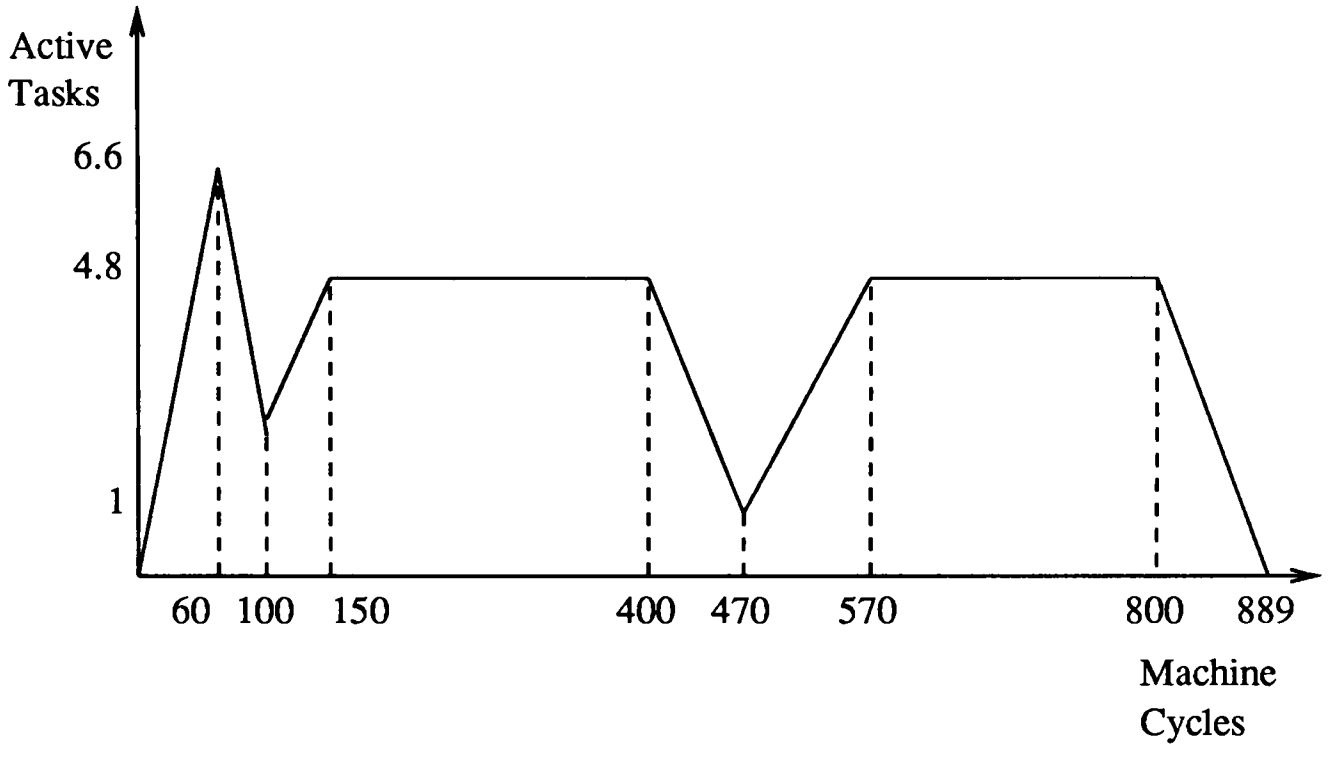
The *deposit* transaction from Section 4.4.1 is too small to provide a good example of the immediate benefits of *optif*. Consider instead a 'long' transaction that performs four updates, and commits only if all of the updates succeed. The transaction optimistically performs all four updates concurrently. Although the operations of a single transaction can be evaluated concurrently, a subsequent transaction is blocked until the preceding transaction has completed. The statistics gathered when the manager processes the two 'long' transactions from Appendix A.3.2 are given below. In the active task graph overleaf the two plateaus represent the updates of the two transactions being performed in parallel. Also note that the optimistic version has performed slightly more reductions than the eager version.

TWO OPTIMISTIC TRANSACTIONS

Metric	Eager	Optimistic
Number of super-combinator reductions	825	889
Number of delta-reductions	734	815
Number of graph nodes allocated	22453	27871
Number of machine cycles	2913	894
Average number of active tasks	1.14	3.97

Elapsed-time reduction factor 3.26.
Concurrency increase factor 3.48.

Figure 5.6 Two Optimistic Transactions



Note that optimistic if is not the only means of introducing concurrency within a transaction. A strictness analyser could ascertain that the commit predicate requires the result of all of the updates, and hence that they can be safely evaluated in parallel.

For larger transactions greater concurrency can be obtained. For example, an optimistic transaction with 10 updates increases concurrency by a factor of 5.60 and reduces the elapsed time by a factor of 5.18. However, because concurrency is only possible within a transaction, the amount of concurrency is bounded by the number of operations in the transactions processed.

Run-time Transformation

Concurrency can be increased further by allowing subsequent transactions to start on the assumption that the present transaction will commit. Loosely speaking a subsequent transaction is applied to the database resulting from the present transaction. If the present transaction is total, and the *then* and *else* keywords are omitted, this can be sketched

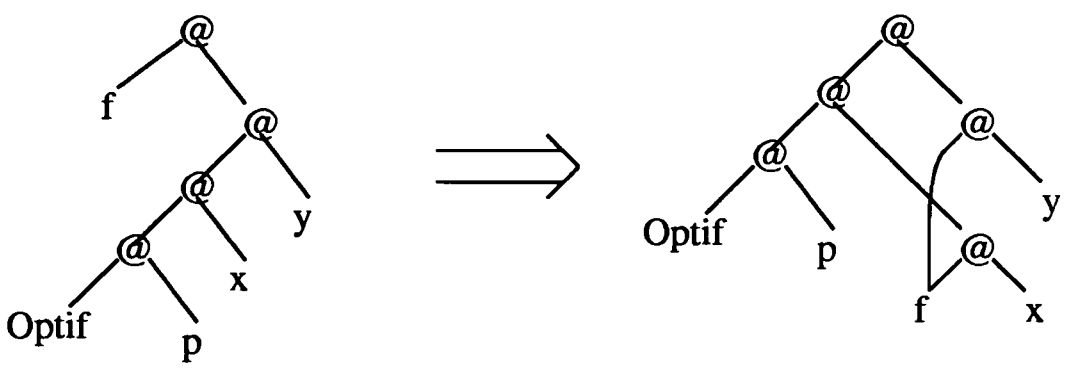
$$f\ (optif\ \ p\ \ x\ \ y).$$

It is reasonable to assume that a transaction function will examine the database, i.e. that f is strict. Recalling that *optif* has identical semantics to *if*, the following distribution law can be used. If f is strict or p terminates, then

$$f\ (optif\ \ p\ \ x\ \ y)\ \ =\ \ optif\ \ p\ \ (f\ \ x)\ \ (f\ \ y).$$

The evaluation of the subsequent transaction, is initiated on the tentative result of the current transaction, $(f\ \ x)$. The required transformation of a reduction graph is shown in figure 5.7. The transformation is performed whenever a function demands the value of an optimistic if. Note that no results can be returned from the subsequent transactions until the predicate is evaluated. If the predicate is false, then the evaluation of the *else* branch will restart subsequent transactions.

Figure 5.7 Optimistic Transformation

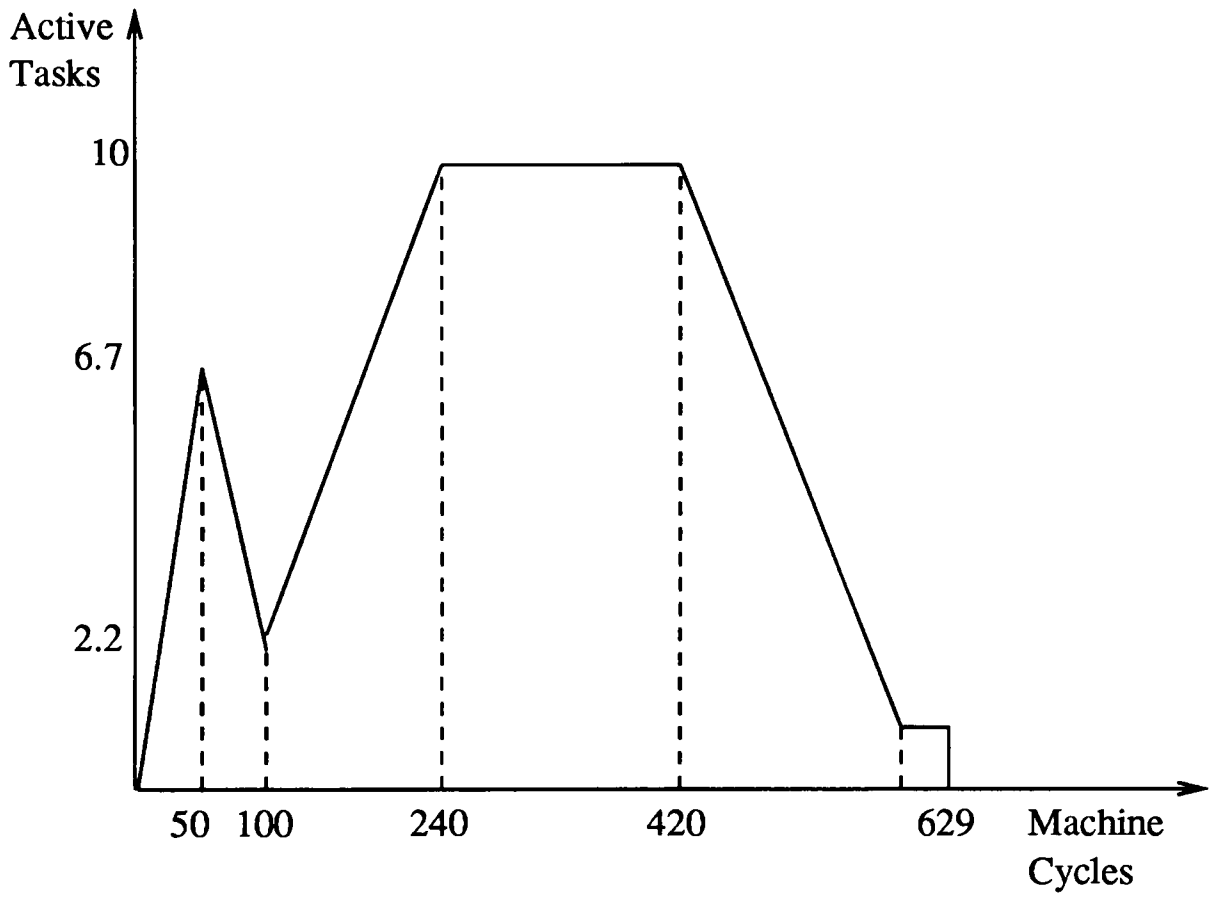


If the two 'long' transactions are evaluated using the transformation the second transaction is no longer blocked awaiting the completion of the first. This is reflected in the results below, where the graph no longer has two separate plateaus representing the two transactions. Instead there is a single plateau representing both transactions occurring concurrently. The metrics record the increase in concurrency from an average of 3.97 tasks in the optimistic case to 6.13 tasks with the transformation. Similarly the elapsed time to evaluate the transactions has dropped from 894 machine cycles in the optimistic case to 627 cycles with the transformation. Note that the elapsed-time reduction factor is significantly lower (15%) than the concurrency increase factor because the evaluation using optimistic if and the transformation has performed more reductions (11%) than the eager version.

Metric	Eager	Optimistic	Opt. + Transf.
No. of super-combinator reductions	825	889	889
No. of delta-reductions	734	815	847
No. of graph nodes allocated	22453	27871	60290
No. of machine cycles	2913	894	627
Average n.o. of active tasks	1.14	3.97	6.13

Elapsed-time reduction factor, over eager evaluation, 4.65.
Concurrency increase factor, over eager evaluation, 5.38.

Figure 5.8 Two Optimistic Transactions with Transformation



Space Consumption

In the above example a total of 123 *optif* transformations are performed. The most significant feature of the results is that the number of nodes allocated has doubled. The additional space is allocated when ~~a~~ an optimistic if is shared. Consider the reduction of the following application.

$$(\lambda a.a + a)\ (optif\ \ p\ \ x\ \ y)$$
$$\Rightarrow \{\beta \text{ reduction}\}$$

$$
\begin{array}{ll}
 & (optif\ \ p\ \ x\ \ y) + (optif\ \ p\ \ x\ \ y) \\
\Rightarrow & \{\text{Transformation}\} \\
 & (optif\ \ p\ \ (+x)(+y))\ \ (optif\ \ p\ \ x\ \ y) \\
\Rightarrow & \{\text{Suppose } p \text{ evaluates to true}\} \\
 & (+x)(optif\ \ True\ \ x\ \ y) \\
\Rightarrow & \{\text{Transformation}\} \\
 & optif\ \ True\ \ (x + x)\ \ (y + y) \\
\Rightarrow & \{\text{Optif}\} \\
 & x + x
\end{array}
$$

The additional nodes are required to construct a new segment of graph that represents the expression $optif\ \ p\ \ (+x)\ \ (+y)$. The optimistic if is evaluated by each expression that shares it and this corresponds to a call-by-name strategy. However, most of the laziness is retained as p, x and y are all shared.

Much of the additional allocation cost can be averted by using a strictness analyser. At present the transformation is performed whenever a strict primitive demands the value of an optimistic conditional. In a program that has been annotated by a strictness analyser the transformation can be applied whenever a strict function is encountered. As a result the optimistic conditionals are distributed through larger functions, and hence fewer transformations are required. This can also be viewed as preserving the sharing structure of strict functions. Recall that the transactions are in fact strict as they inspect the database. For example, if ! indicates a strict function, the above application is reduced as follows and only one transformation is required.

$$
\begin{array}{ll}
 & (\lambda a.a + a)!\ \ (optif\ \ p\ \ x\ \ y) \\
\Rightarrow & \{\text{Transformation}\} \\
 & (optif\ \ p\ \ ((\lambda a.a + a)!\ \ x)\ \ ((\lambda a.a + a)!\ \ y) \\
\Rightarrow & \{\text{Suppose } p \text{ evaluates to true}\} \\
 & (\lambda a.a + a)!\ \ x \\
\Rightarrow & \{\beta \text{ reduction}\} \\
 & x + x
\end{array}
$$

Further Examples

Optimistic if can also improve the sequence of deposits and balance enquiries, as the following results show. The previous results are duplicated for comparison. Note that optimistic evaluation regains 70% of the concurrency available when the operations are performed individually.

FIVE BANK TRANSACTIONS

Metric	Individual Operations	Eager Transactions	Optimistic Transactions
No. of super-combinator reductions	521	559	600
No. of delta-reductions	478	504	688
No. of graph nodes allocated	18807	19243	34000
No. of machine cycles	428	1216	838
Average no. of active tasks	4.43	1.67	3.08

Elapsed-time reduction factor, over eager evaluation, 1.45.
Concurrency increase factor, over eager evaluation, 1.84.

If the predicate in a conditional fails, optimistic evaluation will perform some unnecessary work. However, in all but the most pernicious instances, the concurrency made possible by *optif* will still reduce the time required to evaluate transactions. Consider the case of two 'long' transactions where the second update of the second transaction fails. The LML program can be found in Appendix A.3.3. The following statistics show that, although optimistic evaluation performs 32% more super-combinator reductions than eager evaluation, it reduces the time to evaluate the transactions by a factor of 2.35. The elapsed-time reduction factor differs significantly from the concurrency increase factor because many of the additional tasks made possible by optimistic evaluation have performed unnecessary reductions.

OPTIF FAILING

Metric	Eager	Optimistic
Number of super-combinator reductions	675	889
Number of delta-reductions	571	783
Number of graph nodes allocated	20974	27314
Number of machine cycles	2317	986
Average number of active tasks	1.14	3.62

Elapsed-time reduction factor, over eager evaluation, 2.35.
Concurrency increase factor, over eager evaluation, 3.18.

5.4.3 Friedman and Wise If

As an alternative to optimistic if, Friedman and Wise if also offers a solution to the problem of total transactions. As described before, total transactions retain control of the root of the database until after the decision to commit or abort has been made. In most cases the bulk of the database will be the same whether or not the transaction commits. This common, or unchanged, part of the database will be returned whatever the result of the commit decision. If there were some way of returning the common part early then concurrency would be greatly increased. Transactions that only depend on unchanged data can begin and possibly even complete without waiting for the total transaction to commit or abort.

The common parts of the database can be returned early using *fwif*, a variant of the conditional statement proposed by Friedman and Wise [36]. To define the semantics of *fwif* let us view every value as a constructor and a sequence of constructed values. Every member of a an unstructured type is a zero-arity constructor — the sequence of constructed values is empty. Using C to denote a constructor, the semantics can be given by the following reduction rules. The reduction rules are in a form that differs from Friedman and Wise's, but makes a later definition clear.

$$
\begin{array}{l}
\mathit{fwif}\ \mathit{True}\ x\ y \Rightarrow x \\
\mathit{fwif}\ \mathit{False}\ x\ y \Rightarrow y \\
\mathit{fwif}\ p\ (C\ x_0 \ldots x_n)\ (C\ y_0 \ldots y_n) \Rightarrow C\ (\mathit{fwif}\ p\ x_0\ y_0) \ldots (\mathit{fwif}\ p\ x_n\ y_n)
\end{array}
$$

The third rule of *fwif* represents a family of rules, one for each constructor in the language. The third rule distributes *fwif* inside identical constructors, allowing the identical *then* and *else* constructor, C, to be returned before the predicate is evaluated. As an example, consider a conditional that selects between two lists that have 1 as the first element.

$$
\begin{array}{ll}
 & \mathit{fwif}\ p\ (\mathit{cons}\ 1\ xs)\ (\mathit{cons}\ 1\ ys) \\
= & \{\mathit{fwif}\ 3\} \\
 & \mathit{cons}\ (\mathit{fwif}\ p\ 1\ 1)\ (\mathit{fwif}\ p\ xs\ ys) \\
= & \{\mathit{fwif}\ 3\} \\
 & \mathit{cons}\ 1\ (\mathit{fwif}\ p\ xs\ ys)
\end{array}
$$

Note how the first element of the list is now available.

Semantically *fwif* is identical to the standard *if* except when the predicate fails to terminate, i.e. has value $\perp$. Normally a conditional does not terminate if its predicate fails to terminate, i.e. *if* is strict in its first argument. In contrast *fwif* returns the common parts of its *then* and *else* branches, even if the predicate does not terminate. If the two branches have identical values, then this will be the value of the conditional irrespective of the termination of the predicate. The meaning of *fwif* can be described in domain theory as

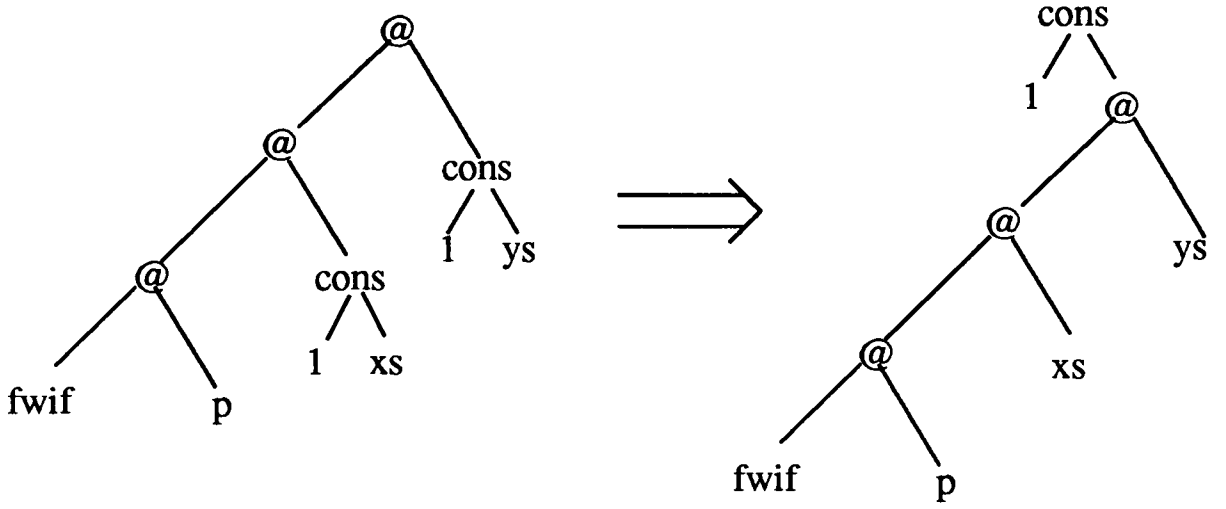
$$\mathit{fwif}\ p\ x\ y\ =\ (\mathit{if}\ p\ x\ y)\ \sqcup\ (x \sqcap y).$$

To implement *fwif* the predicate and the two conditional branches are evaluated concurrently. The values of the conditional branches are compared and

common parts are returned. When a part is found not to be common to both branches the evaluation of those branches ceases. Once the predicate is evaluated, the chosen branch is returned and the evaluation of the other is cancelled. This strategy amounts to speculative parallelism, the conditional branches being evaluated in the hope that parts of them will be identical.

Note that the third reduction is not as inefficient as it at first appears. When '*fwif p*' is distributed over the constructed values every occurrence of *p* shares the same instance of the predicate. Lazy evaluation ensures that the predicate is evaluated only once and its value is used everywhere. The reduction of a fragment of graph representing the foregoing example is illustrated in figure 5.9.

Figure 5.9 *fwif* Reduction



Friedman and Wise if is a known primitive and its application within the manager appears straightforward. It is not implemented in the prototype database.

5.4.4 FWOF

Both *optif* and *fwif* are employed to obtain concurrency in the presence of total transactions. A natural extension is to combine them to produce a primitive, *fwof*, that has a combination of their properties. A total transaction constructed using *fwof* not only returns the common parts of the database early but also optimistically produces tentative results and allows other transactions to proceed on the tentative results.

The semantics of *fwof* can be defined by adding a family of reduction rules to the definition of *fwif*. Like the third reduction rule, the fourth rule represents a family of rules, one for each pair, C_0 and C_1, of different constructors of the same type.

$$
\begin{aligned}
&\mathit{fwof}\ \mathit{True}\ (x, y) \Rightarrow x \\
&\mathit{fwof}\ \mathit{False}\ (x, y) \Rightarrow y \\
&\mathit{fwof}\ p\ (C\ x_0 \ldots x_n)\ (C\ y_0 \ldots y_n) \Rightarrow \\
&\qquad C\ (\mathit{fwof}\ p\ x_0\ y_0) \ldots (\mathit{fwof}\ p\ x_n\ y_n) \\
&\mathit{fwof}\ p\ (C_0\ x_0 \ldots x_n)\ (C_1\ y_0 \ldots y_n) \Rightarrow \\
&\qquad \mathit{optif}\ p\ (C_0\ x_0 \ldots x_n)\ (C_1\ y_0 \ldots y_n), \textbf{ if } C_0 \neq C_1
\end{aligned}
$$

Because *optif* has the same semantics as *if* it is not surprising that *fwof* has identical semantics to *fwif*. This is easy to prove by case analysis on the value of the predicate. The new reduction rule introduces an operational distinction between *fwif* and *fwof*. The new rule is applied when the *then* and *else* branches have different values. At this point in *fwif*, the evaluation of the different branches ceases. In *fwof*, however, an optimistic assumption is made that the *then* branch will be selected, and its evaluation is continued. Incoming transactions may also be distributed into the conditional, permitting them to start on the tentative results of the original transaction.

5.4.5 Balancing

To guarantee access time the tree must be balanced. Maintaining balance reduces concurrency, as described above. One method of alleviating this

problem would be to leave the tree unbalanced after an insertion, and periodically schedule a transaction to rebalance the whole tree. Unfortunately rebalancing the tree requires exclusive access to the database for some time. To minimise this problem conventional databases maintain the balance incrementally, as part of performing transactions.

To allow incremental balancing a new non-deterministic primitive called *ButIfYouHaveTime* was proposed in [5]. The expression

$$(a \ \mathit{ButIfYouHaveTime} \ b)$$

returns the value b, given sufficient time to compute it, but returns a if an answer is required before b has completed. An insert function that preserves balance if given the time can be written

$$\begin{aligned} &\mathit{insert}\ e'\ (\mathit{node}\ lt\ k\ rt) = n\ \mathit{ButIfYouHaveTime}\ (\mathit{rebalance}\ n), \textbf{if}\ \mathit{key}\ e' < k \\ &\quad \textbf{where} \\ &\qquad n = \mathit{node}\ (\mathit{insert}\ e'\ lt)\ k\ rt \\ &\qquad\quad = \ldots \end{aligned}$$

If the newly inserted part of the database is accessed while rebalancing is in progress the rebalancing is cancelled and the unbalanced tree is made available immediately. If, however, the newly inserted part is not accessed the balancing will complete. The behaviour of this version of *insert* mimicks the conventional approach of treating local rebalancing as a 'spare-time' task.

Optimistic if can also be used to permit concurrency while maintaining balance. Each node may be assumed to be balanced, allowing subsequent transactions to proceed. This optimistic assumption will be true for most nodes in a typical B-tree of large order. The insertion function can be written as

$$\begin{aligned} &\mathit{insert}\ e'\ (\mathit{node}\ lt\ k\ rt) = \mathit{optif}\ (\mathit{balanced}\ n)\ n\ (\mathit{rebalance}\ n), \textbf{if}\ \mathit{key}\ e' < k \\ &\quad \textbf{where} \\ &\qquad n = \mathit{node}\ (\mathit{insert}\ e'\ lt)\ k\ rt \\ &\qquad\quad = \ldots \end{aligned}$$

This solution has several advantages over *ButIfYouHaveTime*. The tree is always balanced, and this guarantees response time. Also, neither a new primitive nor non-determinism is introduced. When rebalancing is required and a subsequent transaction demands the rebalanced node *ButIfYouHaveTime* has a speed advantage because it abandons the rebalancing whereas *optif* completes the balancing.

Chapter 6

Database Support

This Chapter covers the design of a more realistic functional database. The facilities illustrated are access to multiple classes, views, security, alternative data-structures and support for data models. A class of data structure that cannot be maintained under a non-destructive update regime is also encountered.

6.1 Introduction

Efficient and concurrent access to a single class of data was demonstrated in the preceding Chapters. How to extend the implementation to support multiple classes is described in this Chapter. Different views of the data are constructed and restricted views can be used to maintain the security of the data. The potential use of data structures other than trees is described, including the provision of secondary indices. The implementation is shown to support both the relational and the functional data models. Finally it is shown that 'closely-linked' graph structures cannot be maintained efficiently under a non-destructive update regime. Several of the extensions described above use abstract data types in a conventional, but purely functional manner. This demonstrates that a type system developed for procedural database programming languages can be used in functional database languages.

Some database issues not addressed are resilience, consistency constraints and active database facilities. These are not addressed because of time constraints rather than any intrinsic difficulty. For example, the multiple versions of a database appear to make a checkpointing form of resilience particularly easy to implement.

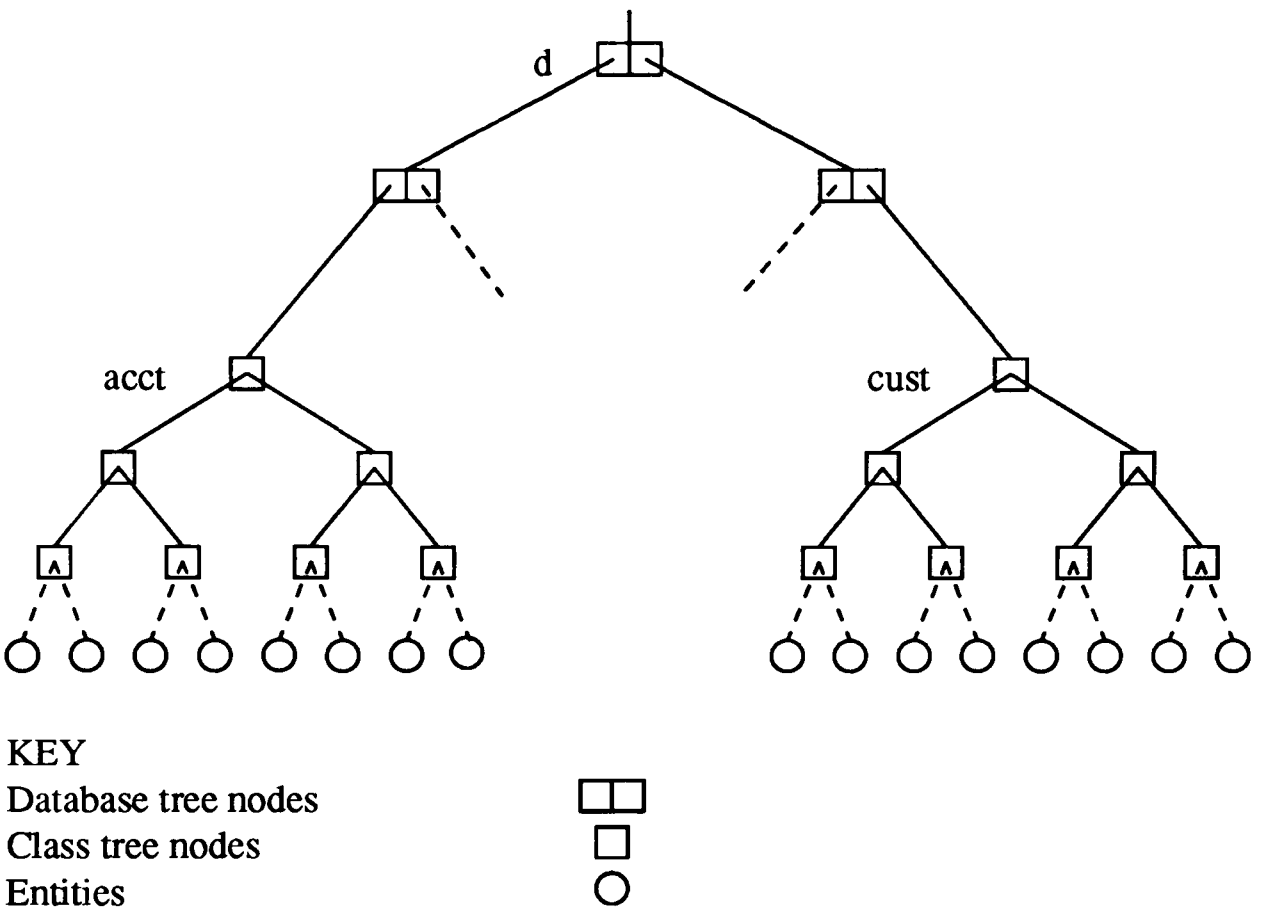
The remainder of this Chapter is structured as follows. Section 6.2 describes how multiple classes of data can be stored. Section 6.3 describes how security and different views of the data can be implemented. Section 6.4 describes how data structures other than simple trees can be supported. Section 6.5 describes how the implementation can support data models.

6.2 Multiple Classes

A database must support classification, i.e. many homogeneous sets of entities must be stored. Different data models view these sets as relations, classes or entity sets. To support multiple classes the manager function remains unchanged but the database becomes a tree of trees, as depicted in Figure 6.1. This approach has a pleasing generality as any operation that can be applied to an entity can be applied to a class. For example, a class can be inserted or updated.

The entities belonging to a class and the operations permitted on the class can be encapsulated as an abstract data type. As described in Subsection 4.2.2, the tree structure is polymorphic, i.e. it is independent of the entities stored. As a consequence, a single generic data manager can be constructed. An instance of the manager can then be constructed for each different class of data simply by parameterising the generic data manager with the key and entity types, a null value and comparison functions.

Figure 6.1: A Multiple-Class Database



An example of such a generic data manager is included in Appendix B. Two instances of the manager are constructed. Bank account entities are stored in the *acct* structure and customer entities are stored in the *cust* structure. The manager is constructed in a purely functional subset of standard ML and uses the functor and signature mechanisms provided [45].

A detailed explanation of the program is not warranted because it is not uncommon to use abstract data types to encapsulate database structure and functionality [2, 21, 66]. The data manager does, however, demonstrate how multiple classes can be supported by a functional database. It also shows that the type mechanisms developed for procedural database programming languages can be used in functional database languages.

Because all of the classes are stored in a tree with a common root node any

transaction that accesses more than one class will see them in a consistent state. That is, logically each transaction is granted exclusive access to the entire database. Means of restricting this access are described in the next Section. Clearly a transaction must select the classes it needs to access. For example, if *lookup'* is the original retrieval function then the function to retrieve an account entity must be written

$$lookup\ k'\ d\ = lookup'\ k'\ (lookup'\ \text{`}Acct\text{'}\ d)$$

Although not actually done in the program in Appendix B, the class selection can be incorporated into the parameterised data manager in an obvious manner. Assuming this is done, the deposit function from section 4.4.1 becomes

$$\begin{aligned} deposit\ a\ n\ d &= acct.update\ (dep\ m\ n)\ d, \textbf{if}\ (isok\ m) \\ &= (Error, d), \textbf{otherwise} \\ &\quad \textbf{where} \\ &\quad\quad m = acct.lookup\ a\ d \end{aligned}$$

6.3 Views and Security

Often a user's model of the data, or *view* differs from what is actually stored. The user may be interested in only part of the data, for example just the balances of bank accounts. Conversely the user's view may contain intensional data, i.e. data not actually stored but constructed from the stored data. For example a customer's age might be calculated from his or her date of birth.

A related requirement is security. Not all users should have access to every operation on all of the data. For example, a customer's credit limit might not be visible to a teller. Essentially database security allows a user to perform

- a restricted set of operations on
- a restricted set of attributes of

- a restricted set of entities in
- a restricted set of classes.

In conventional databases, security is often provided by views. The parameterised data manager in Appendix B is extended using ML functors to construct two views of the account class. The *atmv* view is suitable for an automatic telling machine. The *ccv* view is suitable for a credit controller. New views can be added without changing the data manager or existing views. This independence is termed *logical data independence* [91].

An example of intensional data is found in *ccv* where the safety margin on an account is constructed from the current balance and the overdraft limit. An example of a restricted set of operations is found in *atmv* where only *update* and *lookup* are possible. An example of a restricted set of attributes is also found in *atmv* where only the balance and account class are visible. An example of a restricted set of entities is found in *ccv* where the credit controller can only see those accounts with a balance less than £50. An abstract data type encapsulating the entire database could be constructed to restrict the classes visible.

As before the use of abstract data types to provide views is not unusual and a detailed explanation is not warranted. The views constructed do, however, demonstrate that both views and security measures can be easily provided for a functional database. The views are a second example of the use of a type system similar to that used in procedural database programming languages.

6.4 Data Structures

It is desirable for an implementation to support a rich set of data structures to model real-world objects and their relationships. Most conventional file organisations correspond to a persistent data structure. This Section describes some desirable structures and their practicality under a non-destructive update regime.

6.4.1 Conventional Structures

An unsorted list corresponds to a heap file organisation [91]. A sorted list corresponds to a sequential file organisation. The great expense of list update was shown in Chapter 4. A list may, however, be used if it is short enough to make lookup cheap and is seldom updated when a copy must be retained.

Conventional databases use dense and sparse indices. A dense index provides an index on every entity in the class or file. An entry in a sparse index identifies several entities, typically as many as will fit on a disk block. Both dense and sparse indices can be implemented in a functional database. A dense index is a tree with individual entities at its leaves. A sparse index is a tree with a list of entities at the leaves. In fact, the B-tree structure recommended in Chapter 4 is a sparse index with additional properties. As a consequence the costs of accessing dense and sparse indices are similar to the costs for B-trees. Generic abstract data types can be constructed to represent a list, a sparse index and a dense index.

Conventional databases also use hash files to provide fast, i.e. constant time, access to data. Unfortunately there seems to be no cheap means of non-destructively updating hashed structures. Hash-file update seems to be closely related to the aggregate update problem outlined in 4.2.1. In the next Section a closely-linked data structure is encountered that cannot be maintained non-destructively. For a time and complexity penalty, however, even these can be represented.

6.4.2 Graph Structures

The structures found in many databases are more complex than the tree structures described in Chapter 4. A graph structure is common, i.e. one with shared sub-structures. Nikhil has noted that non-destructive update of graph structures is expensive. The example Nikhil uses to illustrate this is that of several *course* entities that share the same *classroom* entity. If the seating capacity of the classroom changes, a new version of the *classroom* entity must be constructed. A new version of the *course* entities that share it must also be constructed to refer to the new *classroom* entity. 'In general

the transaction programmer must explicitly identify and rebuild every path from the root of the database down to the "updated" object' [67]. Let us call this problem *graph modification.*

The cost of graph modification is not as high as it might initially appear. Often there are only a few, well-defined paths from the root of the database to the "updated" entity. Secondary indices are a common database structure with shared sub-structures, i.e. the entities pointed to by both indices. In the next Subsection a secondary index is shown to require a single additional access path to be maintained. As a result a secondary index can be implemented at a reasonable cost in time, space and programming effort.

Further, *update* is the only operation that is seriously affected during graph modification. The update regime is not significant for a *lookup.* The *insert* operation is not affected because a reference to the common sub-structure must be added to all of the entities that share it. It is demonstrated in the next Subsection that all references to a *delete*d entity must be removed, or a substantial complexity and access time penalty is paid.

The graph modification problem can always be avoided by using keys to represent the graphical structure, rather than pointers. For example, if the key of the *classroom* entity was stored in each of the *course* entities then they need not be changed when a new version of the *classroom* entity is created. Representing a graphical structure in this way costs access time because, instead of simply dereferencing a pointer, an index lookup must be performed. More significantly, using keys introduces complexity for the programmer who must explicitly perform the lookup. The *classroom* entity is no longer logically part of the *course* entities. A pernicious instance of the graph modification problem is described in Subsection 6.5.3.

6.4.3 Secondary Indices

It is often desirable to retrieve data on more than one key. For example, in addition to being able to retrieve an account entity by its account number it may also be necessary to retrieve it by its holder's name. The conventional means of providing this access is to have a *secondary index* on the account

class that maps a customer's name to the list of account entities that the customer owns. All accounts owned by a customer could be determined by scanning the account class, requiring n block accesses. An index is much faster, requiring only $O(\log n)$ accesses. The index, however, requires additional space and must be maintained, for example when an entity is inserted.

Secondary indices can be supported in a functional database. The primary and secondary indices appear as separate trees. However, the entities at the leaves of the trees are shared. As a result, a functional multiply-indexed tree has the same space requirements as its procedural counterpart. Often a secondary index will be sparse, i.e. it will contain a list of pointers to entities. Figure 6.2 depicts the *acct* and *owns* indices.

Because of the expense of non-destructive list update these indices work well if the list is short or is seldom updated when a copy is required. The cost is not as high as it might at first appear because the list elements are key, pointer pairs and hence so small that many of them will reside on a single block.

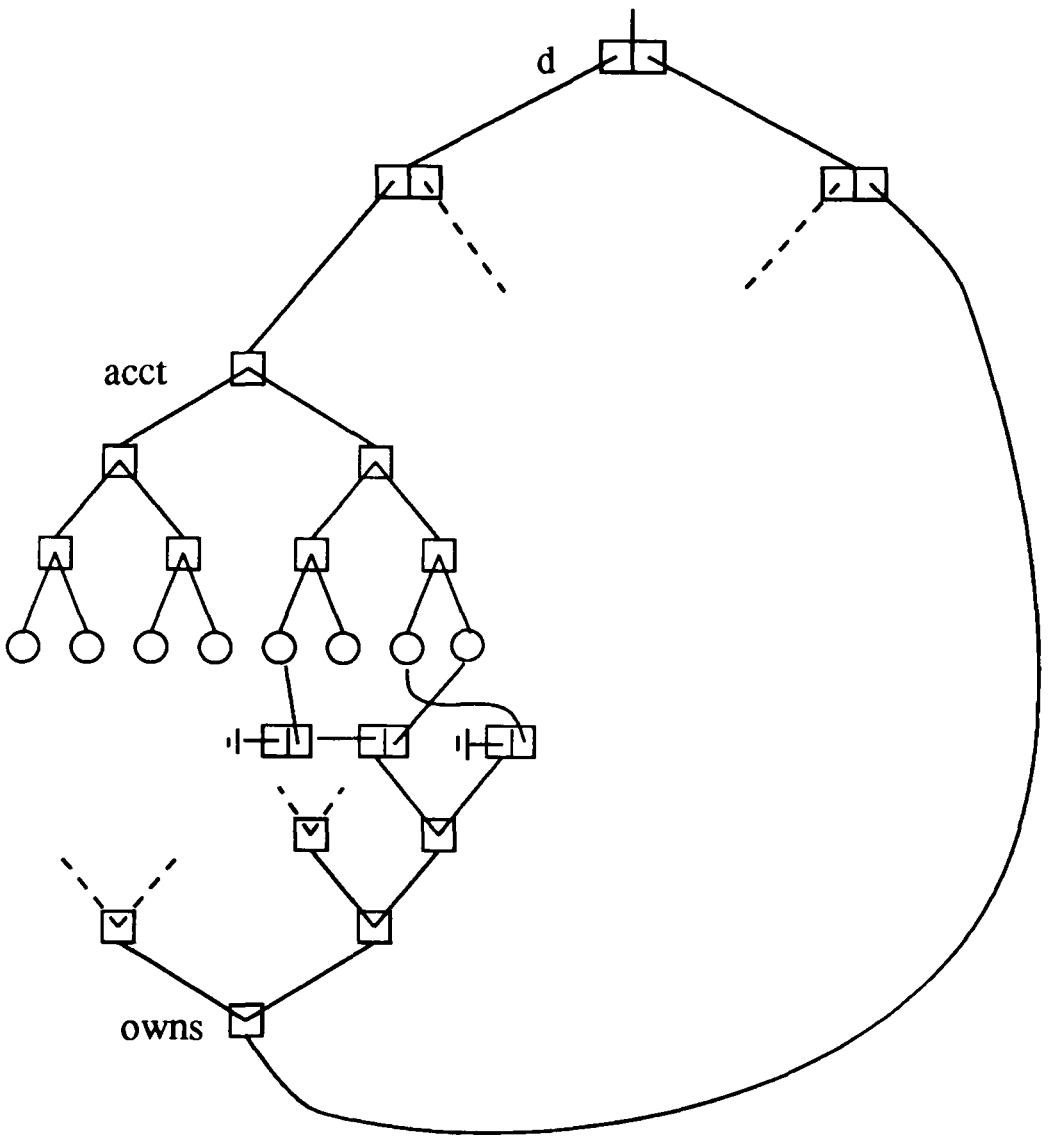
To ensure that the two indices share the same entities the tree manipulating operations must be written with some care. In *insert*, for example, the new entity must be added to both indices. If the secondary index *owns* is an instance of the data manager parameterised to use a customer's name as the key and have a list of accounts as the entity type, then the *insert* function can be written as follows. Both indices point to the same account entity, e, because e is passed to both *acct.insert* and *owns.insert*.

$$
\begin{aligned}
insert\ e\ d &= (d\ , out'),\ \ if\ out' \neq OK \\
&= (d\ , out''),\ \ if\ out'' \neq OK \\
&= (d''\ , out''),\ \ otherwise
\end{aligned}
$$

where

$$
\begin{aligned}
(d'\ , out') &= acct.insert\ e\ d \\
(d''\ , out'') &= owns.insert\ e\ d'
\end{aligned}
$$

Figure 6.2 A Secondary Index



Let us examine the efficiency of operations on a class of size n with m secondary indices. With the reference-counting optimisation, if the old database

version is not to be preserved the operations have the same space costs under a destructive update regime as under a non-destructive regime. The time costs of destructive and non-destructive index operations are summarised in the following table and are amplified below.

TIME COSTS

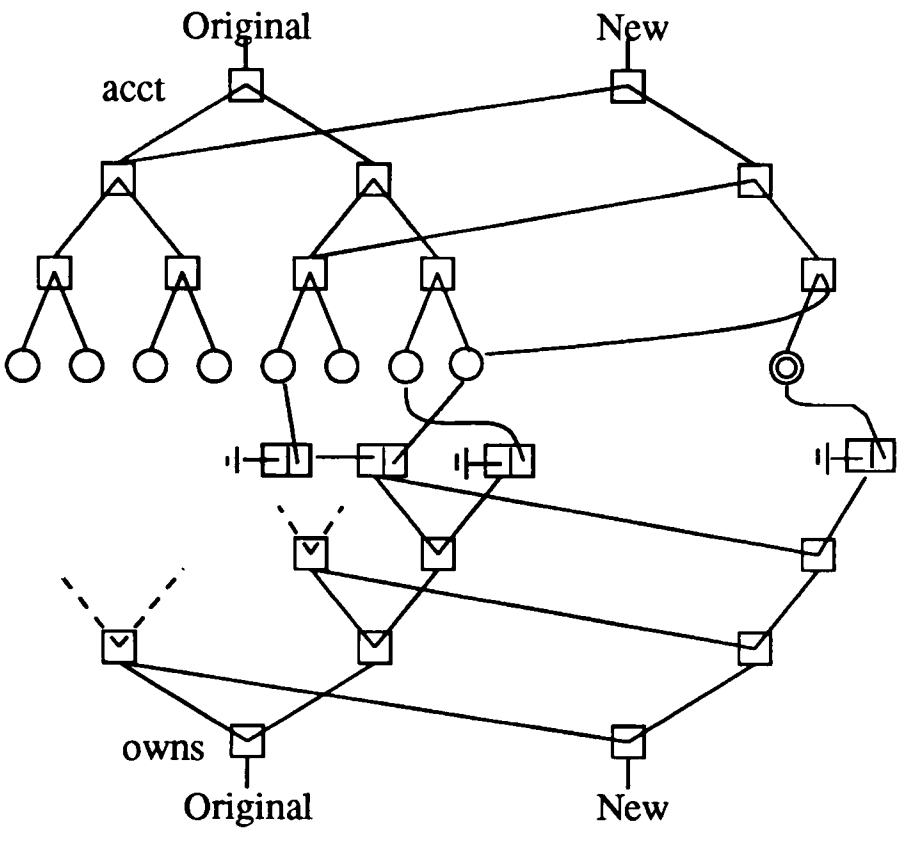
Operation	Destructive Update	Non-destructive Update
lookup	$O(\log n)$	$O(\log n)$
insert	$O(m \log n)$	$O(m \log n)$
update	$O(\log n)$	$O(m \log n)$
delete	$O(\log n)$ OR $O(m \log n)$	$O(m \log n)$

A *lookup* requires the same time under destructive and non-destructive regimes. The path in a single index is followed with a cost of $O(\log n)$. The *insert* operation requires the same time under both regimes because the new entity must be inserted into each of the m indices, giving a total cost of $O(m \log n)$.

Under a non-destructive regime updating an entity in a tree with a secondary index is slower. More space will be required if the original version is to be preserved or there is no reference-counting optimisation. Under a destructive regime any index can be followed and the specified entity modified in place, with a cost of only $O(\log n)$. The destructively modified entity will remain visible from any other index. Under a non-destructive regime a new path must be constructed in every index that references the updated entity giving a total cost of $O(m \log n)$. A new version of the *acct* and *owns* indices is depicted in Figure 6.3. Note how the original version of the index structure can be preserved.

Under a non-destructive regime deleting an entity in a tree with a secondary index may be slower. Under a non-destructive regime the existing path must be removed from every index that references the entity. A destructive deletion can be performed by deleting the entity from every index, with a cost of $O(m \log n)$. Alternately a destructive deletion can be performed by following a single index and marking the entity as deleted, with a cost of just $O(\log n)$. This strategy, however, causes pointers to the deleted entity in other indices to *dangle* — they point to an entity that no longer exists.

Figure 6.3: A New Version of a Secondary Index



Dangling pointers make moving entities and reusing the space freed by deleted entities difficult [32, 91]. If there are several pointers to an entity, then moving the entity will cause pointers to dangle. Ullman's scheme disallows movement altogether. This is a serious restriction because some structures require to move entities — B-tree rotations are a good example. Date describes an indirection mechanism that permits movement but both complicates and slows future operations. If the space originally allocated to a deleted entity is reused, a dangling pointer may now point to the wrong entity. For this reason the space occupied by deleted entities is not available for reuse in

Ullman's scheme. Date again uses indirection to reclaim most of the space at the cost of slower and more complex operations. In summary, the cheaper destructive delete introduces complexity and may slow future operations.

6.5 Data Models

A database management system must support a data model to enable the users to reason about their data at a higher level than that of entities and bulk data structures. The functional database supports both the relational and the functional data models.

6.5.1 Relational Model

The relational data model was defined by Codd [26], and underlies many commercial databases [6, 47, 102]. Homogeneous classes of entities are viewed as mathematical relations, i.e. sets of tuples. In the bank example the account and customer information can be viewed as the following relations.

CUSTOMERS

Name	Address	Phone n.o.
Bloggs, S.	7 Bellevue Rd.	3345632
Bloggs, C.	7 Bellevue Rd.	3345632
Jones, F.	2 George Rd.	5649822
⋮	⋮	⋮

ACCOUNTS

Account n.o.	Balance	Class	Credit Limit
1035	-30	C	200
1040	200	B	50
1045	54	C	100
⋮	⋮	⋮	⋮

A new relation OWNS is introduced to associate customers with the accounts they own. Note that OWNS is many-many, a customer may have many accounts and an account may be owned by many customers.

OWNS

Account n.o.	Names
1035	Bloggs, C.
1035	Bloggs, S.
1040	Bloggs, S.
⋮	⋮

It is easy to provide support for the relational model using the functional database described earlier. A relation is simply a tree. The entities stored in the tree are tuples. The tree can be flattened to obtain the list of tuples currently stored in the relation. This list of tuples can be processed to answer *ad hoc* queries about the database. In Chapter 8 a list-processing notation is presented and proved to be relationally *complete*, i.e. to have at least the expressive power of the relational calculus. Chapter 8 also describes how the uniqueness property imposed by the relational model can be enforced using a list duplicate-removal function.

To implement the scheme outlined above a *flatten* function must be incorporated into the bulk data manager. Suitably parameterised instances of the bulk data manager are also required to store the relations. If these are named *customer*, *account* and *owns*, and *Db* is the type of the multiple-class database, then the following abstract data type provides a relational bank environment. The data type is given in standard ML, with a signature that specifies the interface and a structure that implements the signature.

```
signature BANK =
  sig
    type Customer                    (*Type of a customer tuple*)
    val customers : Db → list Customer

    type Account
    val accounts : Db → list Account
```

```
    type Own
    val owns : Db → list Own
  end;


structure bank : BANK =
  struct
  (*Import structures giving access to the classes of data*)
    structure c = customer
    structure a = account
    structure o = own

    type Customer = c.bkr.Rt (*Type from mgr instance*)
    fun customers d = c.flatten d

    type Account = a.bkr.Rt
    fun accounts d = a.flatten d

    type Own = o.bkr.Rt
    fun owns d = o.flatten d
  end;
```

Note that the relation names, such as *customers*, are not bound to a list of tuples. Instead they are bound to a function that, when applied to the database will return the list of tuples currently stored in the relation. Section 8.3.2 illustrates how this simplifies the task of expressing a query as a transaction function, i.e. a function of the database.

New relations can be constructed and stored for future use. A suitably parameterised instance of the bulk data manager is required. This must be inserted as an, initially empty, element of the multiple class database. The new instance might be called *new*. If $(f\ d)$ is an expression that constructs the list of tuples that is to reside in the new relation, then the following function will construct a new tree containing the elements of $(f\ d)$.

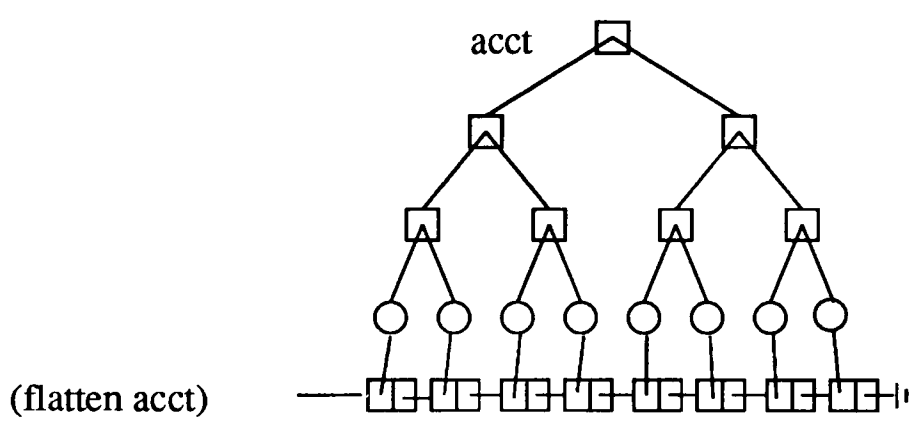
```
makenew d = foldl new.insert d (f d)
```

Let us examine the task of flattening the tree in more detail. The *flatten* function simply traverses the tree and constructs a list that points to the entities at the leaves. The resulting data structure is depicted in Figure 6.4. In Bird and Wadler syntax *flatten* can be written as follows.

$$flatten\ rel\ =\ flat\ []\ rel$$

$$flat\ es\ (Entity\ e)\ =\ e : es$$
$$flat\ es\ (Node\ lt\ k\ rt)\ =\ flat\ (flat\ es\ lt)\ rt$$

Figure 6.4: A Flattened Tree



The *flatten* function is an $O(n)$ operation because it traverses the tree visiting each node and each leaf only once. The list produced by flattening is often processed, for example by a query. The work expended in constructing the list may be avoided by deforestation techniques [96]. These techniques eliminate the intermediate data structures created during list processing.

The flattened list takes up little space, simply n pairs of pointers. As an example, the flattened list for the 5 megabyte tree from Subsection 4.3.2 requires 59 kilobytes, or 1% of the space taken by the tree. Hence retaining a copy of the flattened tree is cheap. If the tree is flattened frequently compared with the frequency of modification, then *memoising* the flatten function will be worthwhile. A memo function is like an ordinary function except that it stores the arguments it is applied to, together with the results computed from them. If a memo function is applied to the same argument again the previously computed value can simply be looked up [64]. Because *flatten*'s argument is a large data structure a variant of memo functions, namely lazy memo functions [54], is appropriate. If *flatten* is memoised and the tree has been flattened previously, a subsequent call to *flatten* need not traverse the tree. Instead the existing flattened version can be used.

A number of memoising strategies are possible, depending on the frequency of modification and the space available. The cheapest alternative is to memoise *flatten* and not *flat*. In this case, if the relation is modified at all, a subsequent invocation of *flatten* must traverse the entire tree again. The most expensive alternative is to memoise both *flatten* and *flat*. In this case every sub-tree in the tree will be memoised. A call to *flatten* on a modified tree will recompute only the changed parts of the tree. Alternative memoising strategies are also conceivable, for example memoising just the second level in the tree

6.5.2 Functional Data Model

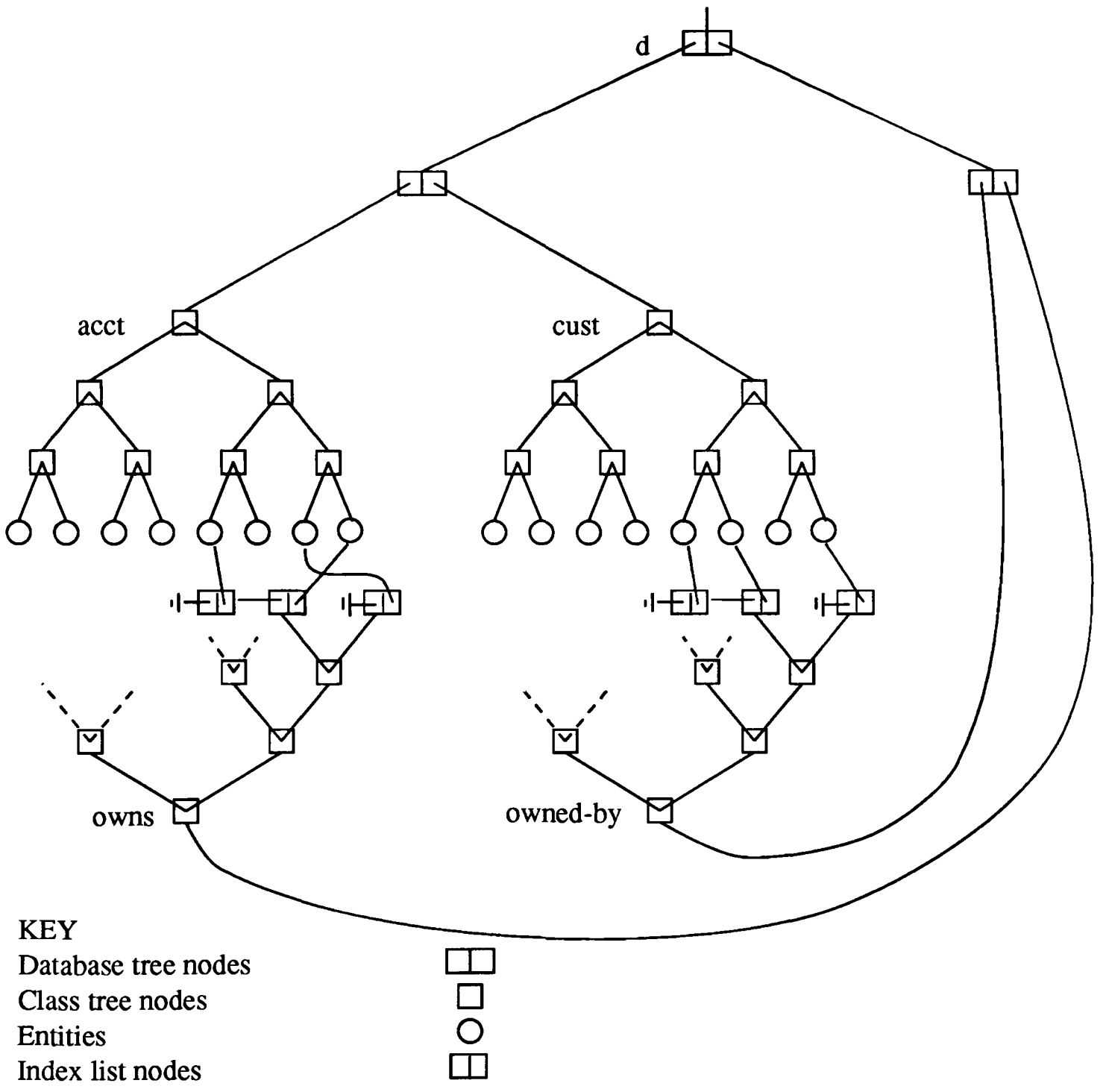
The functional data model was proposed by Shipman in a language called DAPLEX [80]. Related work on expressing the functional data model in functional languages was cited in Sections 3.2 and 3.3. Two concepts are used to model data, *entities* and *functions*. Entities correspond to objects in the real world, for example a customer or a bank account. Entities are collected into classes called *entity sets*. A function maps an entity to a set of target entities. For example *balance* might be a function that maps an account entity to the sum of money in the account. Similarly *owns* might be a function that maps a customer entity to the account entities he or she maintains.

The functional database supports the functional data model naturally. An entity may be represented as a tuple. An entity set is represented as a tree of tuples that can be flattened and queried as described for the relational model. A function such as *balance* that returns an attribute of an entity is simply a tuple selector function. A function that maps an entity to others can be implemented as a secondary index, or by associating a list of keys with the entity.

For example the *owns* function can be represented as a secondary index keyed on the customer's name that stores a list of accounts the customer owns. Alternately a customer entity could contain a list of the keys of the accounts owned. Additional space is required to store the secondary index. The index, however, provides faster access requiring only $O(\log n)$ accesses to locate all of the accounts a customer owns. If a list of keys is stored, each account must be looked up separately, with a cost of $O(\log n)$ for each account. The next Subsection describes why a customer entity cannot refer directly to the account entities.

To represent the many-many relationship between customers and accounts in the FDM, two functions are required. In addition to *owns* described above, a function *owned_by* is needed. Given an account entity *owned_by* returns the list of customers who own the account. Let us use indices to represent both *owns* and *owned_by*. Figure 6.5 is a sketch of the database structure used to implement the *bank* database.

Figure 6.5: The *bank* Data Structure



Assuming that suitable instances of the bulk data manager exist, the following Standard ML abstract data type provides a functional data model of the bank database.

```
signature BANK =
  sig
    type Customer
    val customers : Db → list Customer
    val name : Customer → string
    val address : Customer → list string
    val phone_no : Customer → int

    type Account
    val accounts : Db → list Account
    val account_no : Account → int
    val balance : Account → real
    val class : Account → string
    val credit_limit : Account → real

    val owns : Customer → Db → list Account
    val owned_by : Account → Db → list Customer
  end;


structure bank : BANK =
  struct
  (*Import structures giving access to the classes of data*)
    structure c = customer
    structure a = account
    structure o = owns
    structure ob = owned_by

    type Customer = c.bkr.Rt      (*Type from mgr instance*)
    fun customers d = c.flatten d
    fun name (n, a, p) = n
    fun address (n, a, p) = a
```

```
    fun phone_no (n, a, p) = p

    type Account = a.bkr.Rt
    fun accounts d = a.flatten d
    fun account_no (n, b, cls, crl) = n
    fun balance (n, b, cls, crl) = b
    fun class (n, b, cls, crl) = cls
    fun credit_limit (n, b, cls, crl) = crl

    fun owns c d = o.lookup c d
    fun owned_by a d = ob.lookup a d
end;
```

Nikhil uses an almost identical Student-Course database to illustrate functional data modelling in a functional language. The only difference between the above signature and Nikhil's corresponding environment is that *accounts*, *customers*, *owns* and *owned_by* all take the database as a parameter. For example, the value of *customers d* is the list of customer entities in the database *d*. The explicit reference to the database means that queries using a functional data model can be expressed as transaction functions.

It is possible to argue that the functional database described here provides a more consistently functional data model than that provided by DAPLEX [80], or FDL [72]. In both DAPLEX and FDL updates are achieved by Prolog-like assertion. This is in contrast to the update model outlined in Chapter 4 that allows arbitrary transaction functions over the database.

6.5.3 Closure Problem

The structure required to support a functional model of the bank database illustrates the graph modification problem from Section 6.4.2. The *owns* index is keyed on customer name, just as the *account* entity set is. Rather than having two identical indices we might wish to include the *owns* information in a customer entity. This would take the form of a list of pointers to the accounts a customer owns.

Including this information in the customer entities not only saves the space used by the *owns* index, but makes some accesses fast. Consider the task of listing all of the customers with their accounts, rather like a relational join. This can be achieved in time proportional to the size of the customer entity set, by simply scanning it and following the pointers to the related account entities.

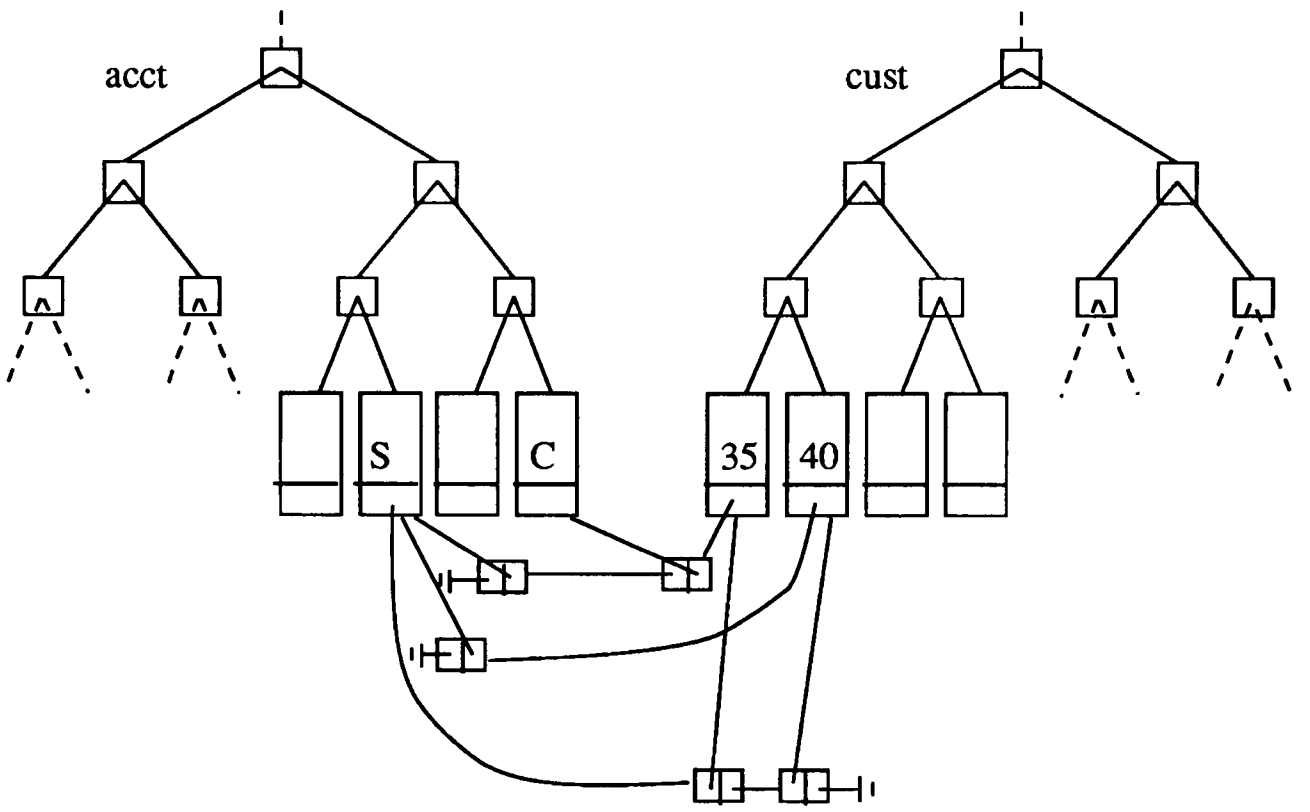
A similar situation exists for the *owned_by* information, and we might wish to incorporate it into the account entities. If both *owns* and *owned_by* information are incorporated, a circular structure is created with customer entities referring to the account entities they own and account entities referring to the customer entities that own them. Figure 6.6 depicts some of the *owns* and *owned_by* information from Subsection 6.5.1. Customers "Bloggs, C." and "Bloggs, S." are denoted "C" and "S" respectively. Similarly accounts 1035 and 1040 are denoted 35 and 40.

Such a closely-linked data structure is a pernicious example of the graph modification problem. Consider the task of allowing a customer *Jones* to use account 1035. A new version of the database must be constructed in which

- *Jones* appears on the *owned_by* list of account 1035, and
- Account 1035 appears on the *owns* list of Jones.

A new customer entity for Jones can easily be constructed. Now, however, a new version of every account owned by Jones must be constructed to refer to the new customer entity. Next, a new version will be required of any customer entity that owns one of the new account entities. The pattern is clear, and in general a new version is required of every entity that is reachable from the updated entity in the transitive closure of the relation represented by *owns* and *owned_by*.

Figure 6.6: Directly-linked *bank* Data Structure



A solution to the problem is to store the key values of the accounts a customer owns in the customer entity and likewise for the *owned_by* information. This breaks the multiple pathways from one entity to another. Jones can be given access to account 1035 as follows. The customer entity set is updated with a new version of Jones's customer entity with the key of account 1035 in the *owns* list. Similarly, the account entity set is updated with a new version of account 1035 that has Jones's name in its *owned_by* list. As noted in Subsection 6.4.2, storing keys in place of pointers increases the access cost and introduces complexity for the programmer.

Part III

Manipulation

Chapter 7

Transactions

This Chapter covers the use of functions as transactions that manipulate the database. How transaction-functions are made atomic is described and the techniques are compared with a conventional logging-and-locking approach. Some issues in the transaction model are also addressed. These include processing long read-only transactions, restarting long transactions, evicting non-terminating transactions and the provision of nested transactions.

7.1 Introduction

A transaction was introduced as a function that consumes the database and constructs a new version of it, i.e. a function of type $bdt \rightarrow (output \times bdt)$. Such functions are a powerful manipulation language. Because a transaction function takes the database as a parameter it can inspect any part of the data, subject to the security mechanism described in Chapter 6. Chapter 8 illustrates queries that interrogate several classes of data and others that are recursive. Because a transaction returns a new version of the database it can modify any part of the data, again subject to the security mechanism described in Chapter 6. A transaction that populates, or builds, an entire class of data was demonstrated in Subsection 6.5.1.

Not only is the notation powerful, it is also referentially transparent and hence amenable to reasoning. The list comprehension queries presented in the next Chapter are essentially the bodies of transaction functions and Chapter 9 demonstrates how these transactions can be transformed to improve efficiency. Referential transparency also facilitates proofs about transactions. For example a transaction might be shown to preserve an invariant in a manner similar to that employed by [79].

The transaction language and the implementation are closely linked. The implementation processes a stream of transactions and expects each transaction to be a function over the database as described above. The transaction language makes use of the cheap multiple versions of the database generated under the non-destructive update regime enforced in the implementation language.

The remainder of this Chapter is structured as follows. Section 7.2 describes how transaction functions are made atomic, and contrasts the techniques used with a conventional logging-and-locking scheme. Section 7.3 describes how some transaction issues can be addressed in the functional database. These are the tasks of processing long read-only transactions, restarting long transactions, evicting transactions that fail to terminate and providing nested transactions. A task not addressed is that of making transactions reliable, i.e. guaranteeing their atomicity in the event of a system failure.

7.2 Atomicity

In Section 3.2 a transaction was defined to be a collection of actions that is atomic. To be atomic a transaction must be both serialisable and total. The actions of a transaction are serialisable if, when a collection of transactions is evaluated concurrently, the result is as if the individual transactions were carried out one at a time in some order. A transaction is total if, providing it returns, it is carried out completely or (apparently) not started.

Some transaction models use a stronger property than totality, namely *reliability* [40]. Totality guarantees that a transaction is executed zero or one

times. Reliability guarantees that a transaction is evaluated exactly once, irrespective of machine failures. Such models are related to the task of making the database resilient in the event of system failures. The transaction functions described in this Thesis are total but not reliable.

Several mechanisms for guaranteeing atomicity exist. The most common is a logging-and-locking strategy. Optimistic concurrency control and time-domain addressing are two other alternatives [59, 74]. For brevity the transaction function mechanism is only compared to a logging-and-locking scheme.

7.2.1 Serialisability

Care is required to ensure serialisability in a conventional database. The most common approach is to employ a two-phase locking protocol. Each transaction must be well-formed, i.e. it must lock every entity it accesses and unlock it after use. A two-phase transaction obtains all of its locks before releasing any. As a simple example a withdrawal transaction must lock the target account entity to exclude other transactions until the withdrawal is complete; it must then release the lock. Locking introduces several difficult issues that are not elaborated here. These include selecting a suitable granularity, avoiding phantoms and the detection and resolution of deadlock.

Essentially serialisability ensures that no two transactions interfere. For example, if a withdrawal transaction does not lock the target account a competing deposit transaction might read the current balance and write a larger balance. The effect of the deposit will be lost when the withdrawal writes its new balance.

A collection of transactions processed by the functional database manager is always serialisable. The transactions occur in the order that they appear in the input stream to the manager. This is because the manager applies the transaction functions to the database in the order that they appear in the input stream. Although the parallelism primitives described in Chapter 5 allow the evaluation of several transactions to be interleaved in time, logically each transaction has exclusive access to the database and subsequent transactions process the database it constructs. Hence a transaction function

can perform many actions on the database without danger of interference.

7.2.2 Totality

The non-destructive update regime enforced by the functional implementation language makes constructing total transactions easy. Indeed the conventional techniques of logging-and-locking, optimistic concurrency control and time-domain addressing all use some form of non-destructive update to ensure totality. In a locking system a log of each transactions actions is maintained and if a transaction aborts all of the changes made are reversed using entries in the log. Under optimistic concurrency control writes occur on temporary copies of the entities. A time-domain addressing scheme is inherently non-destructive. Update in place has even been described as a "poison apple" for reliable data manipulation [41]. In fact the technique used in the functional database that produces a new copy of each node changed in the database-tree is similar to the shadow paging technique found in conventional databases [46].

A transaction function is easily made total because, under the non-destructive update regime, the database prior to evaluating the transaction is preserved. If a transaction needs to abort, the original database is simply reinstated. Preserving the original version of the database is cheap, as described in Section 4.2.3. Further, the transaction programmer's task is simple because the original database is named. For example, the original database is denoted d in the transaction below.

To illustrate totality, consider the transaction *two* that sets the values of two entities to 10. If either of the updates fails the unchanged database d is returned. Only if both updates succeed does the transaction return the database d'' that has been modified by both updates. Note that the d', the database reflecting only one of the updates is never returned by *two*.

$$
\begin{aligned}
two\ a\ b\ d\ = {} & (out',\ d), \textbf{ if } out' \neq OK \\
& (out'',\ d), \textbf{ if } out'' \neq OK \\
& (out'',\ d''), \textbf{ otherwise} \\
& \textbf{where} \\
& \quad (out', d') = update\ a\ 10\ d \\
& \quad (out'', d'') = update\ b\ 10\ d'
\end{aligned}
$$

In a conventional transaction language totality is enforced by a regime that aborts every transaction unless the transaction executes a *commit* command. In the functional transaction language there is no such default behaviour. It is the programmer's responsibility to ensure that the database component of every result returned by a transaction-function reflects either all of the transaction's actions or none of them. As a consequence, non-total transaction functions can be written. Consider the following transaction.

$$
\begin{aligned}
partial\ a\ b\ d\ = {} & (d'', out'') \\
& \textbf{where} \\
& \quad (out', d') = update\ a\ 10\ d \\
& \quad (out'', d'') = update\ b\ 10\ d'
\end{aligned}
$$

If the update of a fails, but the update of b succeeds then only the first of the transaction's updates will have been performed, i.e. *partial* is not total. Worse still, because the update of b succeeded the user is unaware that the transaction has failed. The situation can be alleviated by providing the user with totality-preserving higher order functions or combining forms. One combining form might be a conditional, another might apply a sequence of functions to the database. The sequence combining form can be written as follows.

$$
\begin{aligned}
& sequence\ fs\ d\ = seq\ fs\ d\ d \\
& \\
& seq\ (f : fs)\ d\ d' = seq\ fs\ d\ d'', \textbf{ if } out'\ =\ OK \\
& \qquad\qquad\qquad\ = (out'', d), \textbf{ otherwise} \\
& \qquad\qquad\qquad\qquad \textbf{where} \\
& \qquad\qquad\qquad\qquad\quad (out'', d'')\ =\ f\ d'
\end{aligned}
$$

The *partial* transaction can be made total by using *sequence* as follows.

$$partial\ a\ b\ d\ =\ sequence\ [update\ a\ 10,\ update\ b\ 10]\ d$$

Using combining forms to encourage programmers to write total transactions is a far from perfect solution. A programmer may elect not to use the combining forms and then construct non-total transactions. Any set of combining forms is restrictive and some of the combining forms are clumsy.

7.3 Transaction Issues

This Section outlines how some issues that arise in a transaction-processing model can be addressed in the functional database.

7.3.1 Long Read-Only Transactions

Many applications require long transactions that only read entities. A query that examines every account entity is an example of a long read-only transaction. A query that performs significant computation, a relational join for example, will be even longer. In a conventional database, to guarantee that the query is evaluated in a consistent state, every account entity must be locked and only unlocked once it has been processed. If the query is competing with update transactions it may be difficult for it to obtain a read-lock on every account entity. Furthermore, once the query has locked every account entity it will exclude many updates for a long period. For these reasons long read-only transactions are usually evaluated when few updates are occurring.

In the functional database both reads and writes can overtake a read, as described in Section 5.3.5. Thus a long read-only transaction can proceed on its own private version of the database while subsequent transactions are free to create new versions and inspect them. The space cost of preserving the private version of the database used by the read-only transaction is low,

as described in Section 4.2.3. The cost does, however, go up for each update performed during the read-only transaction. The result of the read-only transaction will reflect the state of the database at the time the transaction started, and this may differ significantly from the state at the time the transaction completes.

Consider a pair of transactions. The first looks up four entities and the second updates the same four entities. When these transactions are processed by the prototype data manager the following results are obtained. The LML program is in Appendix A.3.4.

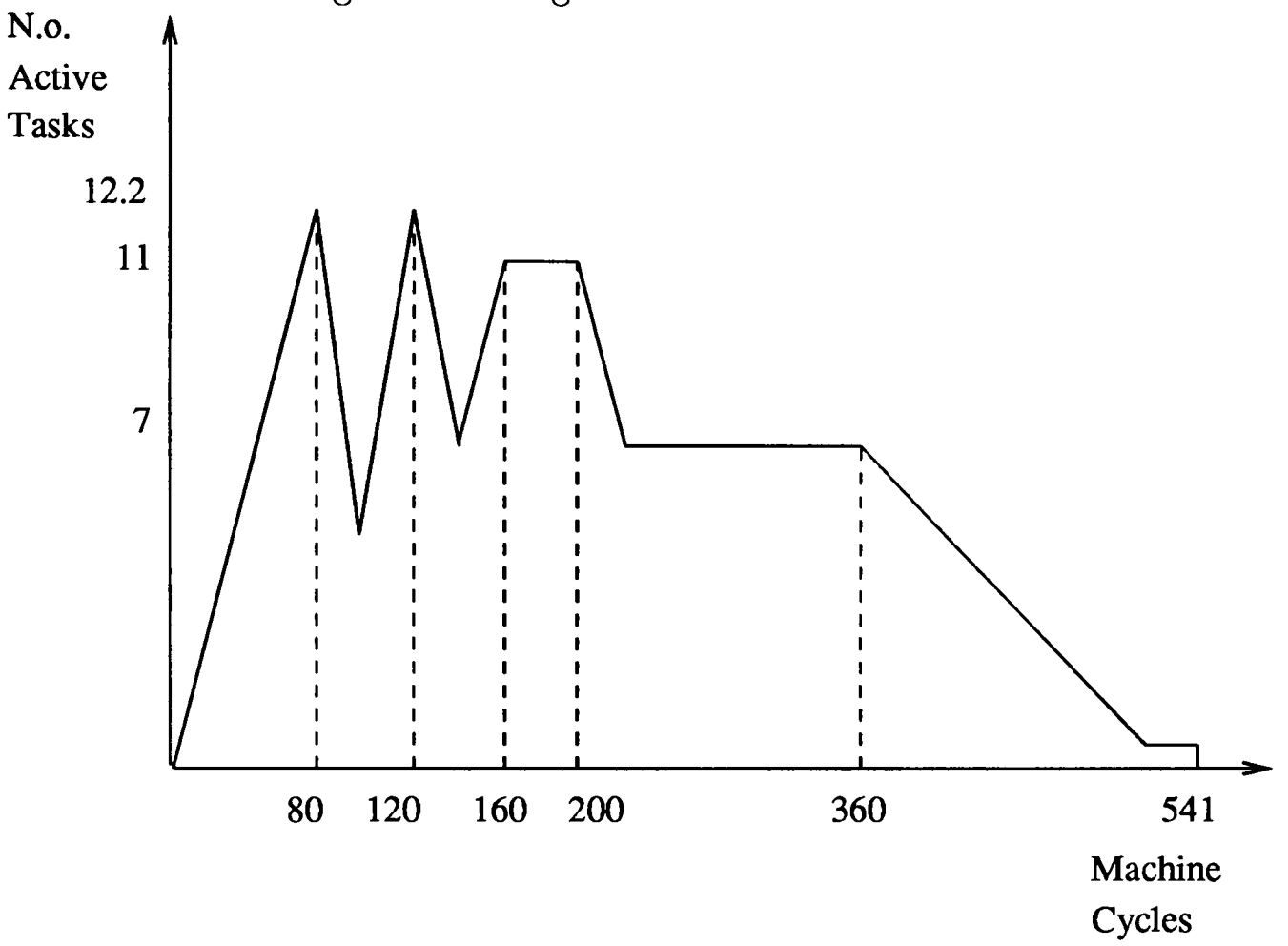
As each transaction has only 4 operations, the maximum concurrency possible if the transactions are processed serially is approximately 5 tasks, including a task for the manager. At least 7 tasks are active during most of the evaluation indicating that the update transaction is occurring in parallel with the read-transaction. Further evidence is provided by the elapsed-time reduction factor of 3.63. Because of the 'set-up' and 'wind-down' times, a speed-up of this size is too great to be obtained by evaluating the two four-operation transactions serially.

LONG READ-TRANSACTION FOLLOWED BY A WRITE-TRANSACTION

Metric	Eager	Opt. + Transf.
No. of super-combinator reductions	685	800
No. of delta-reductions	594	685
No. of graph nodes allocated	20764	52458
No. of machine cycles	1966	541
Average no. of active tasks	1.30	6.02

Elapsed-time reduction factor, over eager evaluation, 3.63.
Concurrency increase factor, over eager evaluation, 4.63.

Figure 7.1 Long Read Transaction



A less spectacular, but more realistic example is obtained if the same read-only transaction is followed by two deposit transactions that both access an entity that is being looked up. The LML program is in Appendix A.3.5. If the transactions are evaluated optimistically, but without the transformation, the deposits are blocked until the read-transaction completes and the sequence of transactions require 1111 machine cycles to complete. Introducing the transformation enables the deposits to occur concurrently with the read-transaction and the transactions only require 904 machine cycles to complete. The full statistics are as follows.

LONG READ-TRANSACTION FOLLOWED BY TWO DEPOSITS

Metric	Opt., no Transf.	Opt. + Transf.
No.of super-combinator reductions	638	696
No.of delta-reductions	503	819
No.of graph nodes allocated	23424	40383
No.of machine cycles	1111	904
Average no. of active tasks	2.07	3.34

Elapsed-time reduction factor, over optimistic evaluation, 1.23.
Concurrency increase factor, over optimistic evaluation, 1.56.

7.3.2 Long Transaction Restart

Many applications require transactions that perform large amounts of work [41]. A great deal of work is lost if a long transaction is aborted. For example, a transaction may abort because an entity it depends on has an erroneous value. Often much of the work the transaction has performed does not depend on the erroneous value and is simply repeated when the transaction is later retried. In the situation described above the cost of aborting a long transaction can be dramatically reduced by memoising the sub-functions of the transaction. Consider the following transaction that depends on 2 entities p and q. Note that the sub-functions f_1 and f_2 have been memoised.

```
long d = f3 a b
         where
            a = memo f1 (lookup 'p' d)
            b = memo f2 (lookup 'q' d)
```

Suppose *long* aborts with the given value of p. The transaction may be retried with a new value of p. Because f_2 (*lookup* 'q' d) does not depend on p and f_2 is memoised the value of the application can simply be retrieved from f_2's memo-table, saving recomputation. The value of any part of the transaction

that depends upon the changed value will automatically be recomputed. In the example the parts that will be recomputed are f_1 (*lookup* '*p*' *d*) and f_3 *a b*.

7.3.3 Non-Terminating Transactions

In a conventional database a transaction may acquire some locks and then fail to terminate. In this case part of the database is unavailable for an indefinite period. A common solution to this problem is to use a timer and evict any transaction that does not complete before its deadline.

A similar situation arises in the functional database. In fact, the situation may be even worse. Recall that the database manager applies the transaction-functions to the database one after the other, a following transaction consuming the database produced by its predecessor. As a consequence, should a transaction not only fail to terminate but also fail to produce any part of the database the following transaction will be unable to start. The failing transaction excludes every other transaction from the database. A more common type of failure will occur where the transaction produces most of the result database before failing. If the failing transaction is constructed using *fwif* or *fwof* then the unchanged parts of the database will remain visible. This situation is similar to that arising under a conventional locking scheme with only small parts of the database unavailable for an indefinite period.

A time-out solution can be employed in the functional database if non-determinism is introduced into the implementation language. The database manager chooses non-deterministically between the database produced by the transaction and an unchanged database. At a machine level the factor guiding the choice is the time taken by the transaction function. Note that the semantics of the transaction language are unchanged. The transactions remain total, i.e. they are either performed completely or not at all. However the laws the database satisfies have changed because it may choose not to apply a transaction-function to the database.

7.3.4 Nested Transactions

It is desirable that existing transactions can be used to construct new transactions; this is called *nesting*. Several models of nested transactions exist [62, 65, 74]. A common model is as follows. The changes made by a sub-transaction are only visible once its parent transaction has completed, i.e. committed. If a parent transaction fails then any changes made by its sub-transactions must be reversed. In contrast, the sub-transactions of a parent transaction fail independently of their parent transaction and of one another. This enables an alternative sub-transaction to be invoked in place of a failed sub-transaction to accomplish a similar task.

Care is required to support nested transactions in a conventional database because the commitment of a sub-transaction is contingent on the commitment of its parent transaction. Nested transactions also complicate the rules governing locking, for example a sub-transaction can lock entities locked by its parent.

In the functional database transactions are simply functions of a given type and hence existing transaction-functions can be freely composed to construct new transactions. A sub-transaction can itself have sub-transactions. Reversing a parent or sub-transaction is easy because the database prior to the application of the transaction function can be cheaply preserved. As a simple example consider *transfer*, a transaction constructed from *deposit* and *withdraw* transactions to transfer a sum of money n from account a to account b. In *transfer* if either sub-transaction fails the entire transaction fails. In the general case, however, the parent transaction is free to inspect the output returned by the sub-transaction and perform some alternate action.

$$
\begin{aligned}
transfer\ a\ b\ n\ d\ = {} & (out', d), \textbf{ if } out' \neq OK \\
& (out'', d), \textbf{ if } out'' \neq OK \\
& (out'', d''), \textbf{ otherwise} \\
& \textbf{where} \\
& \quad (out', d') = withdraw\ a\ n\ d \\
& \quad (out'', d'') = deposit\ b\ n\ d'
\end{aligned}
$$

Part IV

Interrogation

Chapter 8

Queries

This Chapter covers theoretical work on the expression of queries in programming languages. Queries are written as list comprehensions, a feature of some programming languages. Relational queries are demonstrated, as are queries requiring greater power than the relational model provides. It is argued that comprehensions are clear because of their close resemblence to the relational calculus. The power, or relational *completeness*, of list comprehensions is proved. Database and programming language theories are further integrated by describing the relational calculus in a programming language semantics.

8.1 Introduction

There is a great deal of interest in unifying databases with programming languages [1, 2, 7, 12]. Any database programming language must incorporate a query notation. Any well-integrated database programming language must have a consistent theoretical basis. The work described in this Chapter contributes towards unifying database and programming language theory. The database theory used is the relational calculus. The programming language construct recommended for expressing database queries is the list comprehension.

In Chapter 3 a query notation was required to be clear, powerful, concise, mathematically sound and well integrated with the manipulation language. Other work illustrating that list comprehensions are clear, powerful, concise and well-integrated was referenced. In this Chapter it is argued that comprehensions are clear because of their close resemblence to the relational calculus. The power of comprehension notation is proved. Database and programming language theories are further integrated by describing the relational calculus in a programming language semantics.

The power of list comprehension notation is proved by showing that it is relationally *complete*. A notation is complete if it has at least the expressive power of the relational calculus. The completeness of comprehension notation is proved by giving a translation of relational calculus queries into list comprehensions. List comprehensions are shown to be strictly more powerful than the relational calculus by demonstrating that they can express queries entailing computation and recursion, neither of which can be expressed in the calculus.

There is a gap between database theory and programming language theory. The rift arises because database theory is based on relations whereas programming language semantics are not. A semantics is given that bridges this gap. It does so by describing the relational calculus in a well-understood programming language formalism, denotational semantics [39, 82]. This provides the opportunity to apply techniques from the denotational world to relational implementations. These might include proving that implementations match their specification and that optimisations preserve correctness.

It is extremely important to emphasise that the query language work presented in this Chapter is independent of the implementation described in the foregoing Chapters. List comprehensions are not restricted to functional languages. They can be implemented in any language that supports easy heap allocation. A pilot implementation in PS-Algol is under development. Comprehensions may also be used to process lists of data extracted from conventional databases. This is, in fact the approach taken in a commercial database product based on FQL, a lazy list-processing functional language [21].

List comprehension queries can be cleanly integrated with the implementa-

tion described earlier. A comprehension simply forms the body of a transaction function that takes the database as an argument and returns the list of tuples in the database that satisfy the query. Even if the evaluation of a query is time-consuming, it need not delay subsequent transactions because it is read-only and can cheaply retain a private version of the database.

The remainder of this Chapter is structured as follows. Section 8.2 introduces the relational calculus. Section 8.3 introduces list comprehensions and demonstrates their use to express both relational and extra-relational queries. Section 8.4 describes the syntactic correspondence between comprehensions and the relational calculus. Section 8.5 describes the translation process and includes an example. Section 8.6 describes the semantics and its potential uses.

8.2 Relational Calculus

The formulation of the relational calculus used in this Chapter is described in this Section. There are only two minor distinctions between this formulation and Ullman's [91]. Readers familiar with the calculus may wish to omit this Section.

The relational calculus is a formalism for expressing database queries. It was defined by Codd [27] and underlies a number of query languages [47, 60, 102]. A query is written as a Zermelo-Fraenkel set comprehension with a predicate or formula specifying the conditions the tuples must satisfy. This represents a declarative specification of the query as, unlike the relational algebra, no order needs to be given for the computation of the tuples satisfying the formula.

Much of the following description is paraphrased from Ullman *op. cit.* There are two flavours of the calculus, tuple relational and domain relational. The flavours are equivalent as any query expressed in the tuple relational calculus can be converted to a query in the domain relational calculus, and vice versa. Tuple relational queries can be expressed as comprehensions or given a meaning by first converting them to a domain relational form, and then

using the translation or semantics.

Queries in the calculus are of the form $\{(i_1, .. \ i_n)|F(i_1, .. \ i_n)\}$, where $i_1, .. \ i_n$ are *domain variables*, and F is a formula built from atoms and operators. Atoms are of two types.

- $a_i \ \theta \ a_j$ where a_i and a_j constants or domain variables and θ is a comparison operator ($<, =$, etc). This asserts that a_i stands in relation θ to a_j. For example, $x < 3$.
- $(i_1, .. \ i_n) \in R$ where R is a relation and $i_1, .. \ i_n$ are *unbound* domain variables. In a slight departure from Ullman's formulation, membership with constants or bound identifiers $(a_1, .. \ a_n) \in R$ is represented by $(i_1, .. \ i_n) \in R \wedge i_1 = a_1 \wedge .. \ i_n = a_n$.

If F and F' are formulae, then a formula may be one of the following.

- An atom.
- Formulae combined using logical operators, $F \wedge F'$, $F \vee F'$ or $\neg F$.
- A quantified expression, $\exists(i_1, .. \ i_n) : R \ . \ F$ or $\forall(i_1, .. \ i_n) : R \ . \ F$, where $(i_1, .. \ i_n)$ are unbound domain variables. The explicit mention of the relation over which the domain variables range is the second difference between this formulation and Ullman's. It is, however, found in Date's formulation [32].

Parentheses may be used as needed.

Examples

Cartesian Product:

$$\{(u_1 .. \ u_r, v_1, .. \ v_s)|(u_1, .. \ u_r) \in R \wedge (v_1 .. \ v_s \in S\}$$

Difference:

$$\{(i)|(i) \in R \wedge \neg((i') \in S \wedge i = i')\}$$

The projection $\Pi_{i_1,i_2,..\,i_k} R$ is expressed by:

$$\{(u_{i_1}, u_{i_2}, ..\ u_{i_k})|(u_1, ..\ u_r) \in R\}$$

The selection of tuples of a relation R satisfying a predicate F, or $\sigma_F R$, is expressed by

$$\{(u_1, ..\ u_r)|(u_1, ..\ u_r) \in R \wedge F'\}$$

where F' is the formula F with each operand i denoting the ith component of R, replaced by u_i.

Safety

In order to disallow queries with infinite answers such as

$$\{(u)|\neg((u) \in R)\},$$

safety conditions are defined. Informally a query, $\{(u_1, ..\ u_r)|F(u_1, ..\ u_r)\}$, is safe if it can be demonstrated that each component of any $(u_1, ..\ u_r)$ that satisfies F is a member of $Dom(F)$, which is defined as the set of symbols that either occur explicitly in F, or are components of some tuple in some relation R mentioned in F. For example if $F(u_1, u_2)$ is $u_1 =$'a'$\vee(u_1, u_2) \in R$, where R is a binary relation then $Dom(F) = \{$'a'$\} \cup \Pi_1 R \cup \Pi_2 R$. A formal definition can be found in [91].

8.3 List Comprehensions

8.3.1 Introduction

This Subsection briefly introduces list comprehensions or ZF-expressions. Readers familiar with comprehensions may wish to omit it. Full descriptions of comprehensions can be found in many functional programming texts, including [17, 70].

List comprehensions are a construct based on Zermelo-Fraenkel set comprehensions and are found in some functional languages [52, 88, 89, 93]. A set comprehension specifying the set of squares of all the odd numbers in a set A can be written

$$\{square\ x \mid x \in A \wedge odd\ x\}$$

and has a corresponding list comprehension

$$[square\ x \mid x \leftarrow A;\ odd\ x].$$

This can be read as 'the list of squares of x such that x is drawn from A and x is odd'. The syntax of comprehensions can be sketched as

$$\begin{aligned} comp &::= [e \mid q] \\ q &::= q; q \mid f \mid p \leftarrow e \end{aligned}$$

Here e is an expression in the language. The expressions to the right of the vertical bar are called qualifiers. Qualifiers are either filters or generators. A filter, f, is a boolean-valued expression that specifies a condition that potential list elements must satisfy to be included in the result, e.g. $odd\ x$. A generator has the form $p \leftarrow e$, where p is a pattern that introduces one or more new variables. The expression e is list-valued and denotes the sequence of values that the pattern is to be successively matched against. The generator in the above example is $x \leftarrow A$.

8.3.2 Relational Queries

Relational database queries are easily expressed using list comprehensions. The following queries are used by Ullman to compare different query languages. The queries are posed against a small database, called the Happy Valley Farmers Coop, or HVFC. The HVFC database comprises the following relations.

MEMBERS(NAME,ADDRESS,BALANCE)
ORDERS(ORDER_NO,NAME,ITEM,QUANTITY)
SUPPLIERS(SNAME,SADDRESS,ITEM,PRICE)

Let us assume that the underlying implementation provides the following support for relations. The names of the relations are bound to the current list of tuples in the database. For each attribute of every relation there is a selector function that maps from a tuple in the relation to that attribute. For example, *balance* will select the BALANCE attribute of a MEMBERS tuple.

Given this support, the query "Print the names of the members with negative balances" can be written as the comprehension

$$[name\ \ m \mid m \leftarrow members;\ balance\ \ m < 0].$$

The query works in a straight-forward manner. Each tuple in MEMBERS is retrieved by $m \leftarrow members$. If the BALANCE attribute is less than zero, $balance\ m < 0$, then the NAME attribute is included in the result by $name\ m$.

"Print the supplier names, items and prices of all the suppliers that supply at least one item ordered by Brooks" can be written

$$[(sname\ \ s, sitem\ \ s, sprice\ \ s) \mid o \leftarrow orders;\ oname\ \ o = \text{`}Brooks, B.\text{'};\ s \leftarrow suppliers;\ oitem\ \ o = sitem\ \ s].$$

If *sublist* is a function that checks that every element in its first argument list is present in its second, then a query to "Print the suppliers that supply every item ordered by Brooks", can be written

$$[s \mid s \leftarrow suppliers;\ sublist\ brooksitems\ (allitems\ (sname\ s))]$$
where
$$brooksitems = [oitem\ o \mid o \leftarrow orders;\ oname\ o = \text{`}Brooks, B.\text{'}]$$
$$allitems\ sn = [sitem\ s \mid s \leftarrow suppliers;\ sname\ s = sn].$$

List comprehension queries can be simply transformed into transaction functions suitable for the transaction processor described in earlier Chapters. Only two changes are necessary. The comprehensions must be made the body of a function that takes the database as an argument. Each relation name must be bound to a function that takes the database and returns the list of tuples in that relation. Using these conventions, a transaction function representing the first query can be written

$$negbal\ d = [name\ m \mid m \leftarrow members\ d;\ balance\ m < 0].$$

Selector functions have been used to locate the attributes of tuples so that any attribute not relevant to the query can be ignored. This is a substantial advantage for real databases that contain relations with many attributes. An alternative way of writing queries is to provide a pattern that matches all of the attributes. The first query can be written in this style as follows.

$$[(name) \mid (name, address, balance) \leftarrow members;\ balance < 0]$$

This is in fact the style that will be adopted for the translation and for query improvement. The styles are easily interchangeable. The two styles correspond to the variants of the relational calculus. Using selector functions corresponds to the tuple relational calculus, whereas using pattern matching corresponds to the domain relational calculus.

8.3.3 Extra-Relational Queries

List comprehensions are strictly more powerful than the relational calculus. Neither computation nor recursion can be expressed in the calculus and

queries requiring either recursion or computation are termed *extra-relational.* Other workers have also demonstrated that extra-relational queries can be easily expressed using list comprehensions and recursive functions. Comprehension solutions to two classic extra-relational queries from the database literature are presented next. Improving the efficiency of these queries is addressed in the next Chapter.

Date's Example

Date poses the following bill of material, or parts explosion, problem [32]. Given a relation such as

PARTS

Main Component	Sub-Component	Quantity
P1	P2	2
P1	P4	4
P5	P3	1
P3	P6	3
P6	P1	9
P5	P6	8
P2	P4	3

write a program to list all the component parts of a given part to all levels. This problem is recursive only, no computation is required.

A list comprehension solution is a function *explode* with a single argument *main*, the part to be exploded.

$$explode\ main\ = [p \mid (m, s, q) \leftarrow parts;\ m = main;\ p \leftarrow (s : explode\ s)]$$

The *explode* function works as follows. Each tuple in the *parts* relation is obtained by $(m, s, q) \leftarrow parts$. If a tuple's main component is the assembly being exploded, $m = main$, then the parts p returned are the subassembly itself s, and its subcomponents, *explode* s. This solution is far more concise than the 27-line SQL solution Date presents. It is also arguably clearer.

Atkinson and Buneman's Example

Atkinson and Buneman also chose a bill of material as a recursive and computational example [9]. A bill of material database is used to compare many different databases, programming languages and database programming languages. Their bill is more complex than Date's in that there are two types of parts, *composite* and *base*. Composite parts are assembled from other parts, whereas base parts are not. A base part has a name, a mass, a cost and a list of suppliers. A composite part has a name, a cost increment (assembly cost), a mass increment and a list of the parts required to assemble it, including the quantity required of each sub-part. Assuming that the subsidiary types such as *Mass* have already been defined, the type of parts can be specified as the following abstract data type.

$$\begin{aligned} part \;::=\; & Base\ \ name\ \ mass\ \ [suppliers]\ \mid \\ & Comp\ \ name\ \ [(part, qty)]\ \ costinc\ \ massinc \end{aligned}$$

The task set is to compute the total cost and total mass of a composite part.Clearly this requires both recursion and computation. The task proves impossible in most relational query languages. Four auxiliary efficiency goals are identified by Atkinson and Buneman and these are addressed in the next Chapter.

Let us define *sumpair* as a function that performs addition on a list of pairs,

$$sumpair\ \ abs = (sum[a \mid (a, b) \leftarrow abs], sum[b \mid (a, b) \leftarrow abs]).$$

Given this, a simple list comprehension solution is as follows.

$$costandmass\ \ p = cm\ \ (lookup\ \ p\ \ parts)$$

$$\begin{aligned} & cm\ \ (Base\ \ p\ \ c\ \ m\ \ ss) = (c, m) \\ & cm\ \ (Comp\ \ p\ \ pqs\ \ ci\ \ mi) \\ & \quad = sumpair\ \ ((ci, mi) : [(q * c, q * m) \mid (p, q) \leftarrow pqs;\ (c, m) \leftarrow [costandmass\ \ p]]) \end{aligned}$$

The solution works in a straightforward manner. The *costandmass* function simply calls a subsidiary function, *cm*, with the part record corresponding to the given part number. The *cm* function computes the cost and mass of a part record. The cost and mass of a base record are simply the cost and mass attributes of the record. The cost and mass of a composite part are the sum of a list of pairs of costs and masses. The first cost and mass pair is the cost increment and the mass increment for this assembly stage, (ci, mi). The remainder of the list of cost, mass pairs are the costs and masses of the subcomponents. Each subcomponent is retrieved by $(p, q) \leftarrow pqs$. The cost and mass of each subcomponent is calculated, $(c, m) \leftarrow [costandmass\ p]$. Finally the costs and masses of each subcomponent are multiplied by the quantity of the subcomponent required, $(q * c, q * m)$.

This solution is as short as any of those presented by Atkinson and Buneman. It is considerably shorter than most of the solutions given, for example the 9-line ML solution and the 43-line Pascal solution. The solution is also clearer than many of those given in Atkinson and Buneman's paper.

Atkinson and Buneman's query is of additional interest in that it entails processing a non-flat structure. Both base and component part records contain lists of data. Although the task of querying non-flat bulk data types is not seriously explored in this thesis it is a topic of some interest, particularly to the object-oriented database community.

8.4 Syntactic Correspondence

List comprehensions bear a close resemblance to relational calculus queries. Such a close correspondence to a declarative query specification notation makes list comprehensions clear. The similarity arises because both notations are based on Zermelo-Fraenkel set theory. The correspondence is particularly evident between domain relational queries and pattern-matching list comprehension queries. In Section 8.3 a pattern-matching list comprehension expressing Ullman's first example query was given as

$$[(name) \mid (name, address, balance) \leftarrow members;\ balance < 0].$$

The domain relational specification of the same query is

$$\{(name) \mid (name, address, balance) \in members \wedge balance < 0\}.$$

The similarity is not always this strong. For example, if $+\!\!+$ is list concatenation, the calculus query

$$\{(t) \mid (t) \in R \vee (t) \in S\}$$

corresponds to the comprehension

$$[(t) \mid (t) \leftarrow R +\!\!+ S].$$

To be more precise, naively following the translation presented in the next Section produces

$$[(t) \mid (t) \leftarrow [(t) \mid (t) \leftarrow R] +\!\!+ [(t) \mid (t) \leftarrow S]].$$

This can be simplified to the above result using the identity $[x \mid x \leftarrow A] = A$.

8.5 Translation Rules

8.5.1 Outline

The translation entails two stages.

- Translation rules are provided to translate a domain relational query, with its formula in a restricted form, into a list comprehension.

- An algorithm for translating any relational formula into the restricted form completes the process.

The restricted, or *generative*, form simplifies the translation. Essentially it ensures that the range of each variable in the query is determined before the variable is used. Any formula can be manipulated into generative form. Its definition depends on a distinction between relational calculus atoms. Atoms are either generators or filters. With the exception that any negated formula is a filter, generators are

- assertions that a tuple of domain varibles is drawn from a relation.
- assertions that a domain variable is equal to some value.

The generator is $(name, address, balance) \in members$ in the example from the previous Subsection. The filter is $balance < 0$. A formula is generative if it has three properties.

- It is in prenex form. This is a well-known normal form [44] in which the quantifiers occur on the left of the expression. It resolves variable scoping issues, and Date [32] recommends it as a natural way of expressing queries. The example query above has no quantifier and remains unchanged.
- The quantifier free part of the formula is in disjunctive normal form, which is also well known [44]. As there are no disjuncts in the example query, it satisfies this by default.
- All generators in each conjunct occur before all of the filters. Formulae can always be manipulated to make this true because conjunction is commutative. The generator in the example query already occurs before the filter, so no change is required.

Note that safety and generativity are independent. There are generative formulae that are not safe, and safe formulae that are not generative.

The translation starts by defining the syntax of the relational calculus and of list comprehensions. A suite of translation functions from the relational calculus syntax into list comprehension syntax is then given. The translation functions represent a constructive proof of the relational completeness of comprehension notation.

8.5.2 Relational Calculus Syntax

Syntactic Categories

Query	Relational Calculus Query	Q
Exp	Relational Formula	E
Atom	Relational Calculus primitive	A
Op	Comparison Operator	ω
Ide	Domain Variable Identifier	I
RIde	Relation Identifier	R
Const	Constant	k

Abstract Syntax

$$Q = \{(I, \cdots, I) \mid E\}$$

$$E = E \wedge E \mid E \vee E \mid \neg E \mid (I, \cdots, I) \in R \mid$$
$$A\ \omega\ A \mid \exists(I, \cdots, I){:}R\,.\,E \mid \forall(I, \cdots, I){:}R\,.\,E$$

$$A = I \mid k$$

$$\omega = < \mid > \mid \leq \mid \geq \mid \neq \mid =$$

8.5.3 List Comprehension Syntax

This is the target syntax of the translation and it is assumed that the identifiers and constants used in the calculus are valid in it.

Syntactic Categories

Comprehension	List Comprehension	C
Qual	Qualifiers	Qs
Qualifier	Single Qualifier	Q
Plexp	Programming Lang. Expression	E
Gen	Generator	G
Patt	Pattern	P

Abstract Syntax

$C = [E \mid Qs]$

$Qs = Q \mid Q;\ Qs$

$Q = G \mid E$

$G = P \leftarrow E$

E is any expression in the language
P is any pattern in the language, to be matched with elements in the list.

8.5.4 Translation Functions

Translation Types

Simple = Ide + Const
Env = List Ide

The environment contains the domain variables which are known to be already bound.

$\mathcal{Q}$:Query → Comprehension

$\mathcal{Q}$ is a function which translates a domain relational query into a list comprehension. The comprehension computes the list of tuples in the database which satisfy the query.

$$\mathcal{Q}[\![\{(I_1, \cdots, I_n) \mid E\}]\!] = [\![\, [(I_1, \cdots, I_n) \mid e] \,]\!] \qquad (\mathcal{Q})$$
where
$$(\rho, e) = \mathcal{E}[\![E]\!]\ []$$

$\mathcal{E}$:Exp → Env → (Env × Qual)

The $\mathcal{E}$ function is given a list of variables already bound, and translates a relational formula into a qualifier and a new list containing the variables bound in the formula. See the notes on individual equations below.

$$\mathcal{E}[\![E_0 \wedge E_1]\!]\ \rho = (\rho_0 +\!+\ \rho_1, [\![\ e_0; e_1\ ]\!]) \qquad (\mathcal{E}\wedge)$$
where
$$(\rho_0, e_0) = \mathcal{E}[\![E_0]\!]\ \rho$$
$$(\rho_1, e_1) = \mathcal{E}[\![E_1]\!]\ (\rho +\!+ \rho_0)$$

$$\mathcal{E}[\![E_0 \vee E_1]\!]\ \rho = (\rho', [\![\ (\rho') \leftarrow [(\rho') \mid e_0] +\!+ [(\rho') \mid e_1]\ ]\!]) \qquad (\mathcal{E}\vee)$$
where
$$(\rho', e_0) = \mathcal{E}[\![E_0]\!]\ \rho$$
$$(\rho', e_1) = \mathcal{E}[\![E_1]\!]\ \rho$$

$$\mathcal{E}[\![\neg E]\!]\ \rho = ([], [\![\ [\rho' \mid e] = []\ ]\!]) \qquad (\mathcal{E}\neg)$$
where
$$(\rho', e) = \mathcal{E}[\![E]\!]\ \rho$$

$$\mathcal{E}[\![A_0\ \omega\ A_1]\!]\ \rho = \Theta\ [\![\omega]\!]\ (\mathcal{A}[\![A_0]\!]\rho)\ (\mathcal{A}[\![A_1]\!]\rho)\ \rho \qquad (\mathcal{E}\omega)$$

$$\mathcal{E}[\![(I_1, \cdots, I_n) \in R]\!]\ \rho = ([I_1, \cdots I_n], [\![\ (I_1, \cdots, I_n) \leftarrow d\ R\ ]\!]) \qquad (\mathcal{E}\in)$$

$$\mathcal{E}[\![\exists(I_1,\cdots,I_n):R\ .\ E]\!]\ \rho = (\rho', [\![\ (I_1,\cdots,I_n) \leftarrow d\ R; e\ ]\!]) \qquad (\mathcal{E}\exists)$$
where
$$(\rho', e) = \mathcal{E}[\![E]\!]\ (\rho +\!\!+ [I_1,\cdots I_n])$$

$$\mathcal{E}[\![\forall(I_1,\cdots,I_n):R\ .\ E]\!]\ \rho = (\rho', [\![\ (\rho') \leftarrow \mathit{intersect}\ [[\rho' \mid e] \mid (I_1,\cdots,I_n) \leftarrow d\ R]\ ]\!]) \qquad (\mathcal{E}\forall)$$
where
$$(\rho', e) = \mathcal{E}[\![E]\!]\ (\rho +\!\!+ [I_1,\cdots I_n])$$

Notes

$\mathcal{E}\vee, \mathcal{E}\forall$ If $\rho = [I_1,\cdots I_n]$ we write (ρ) for $(I_1,\cdots I_n)$, trusting that this leads to no confusion.

$\mathcal{E}\neg$ The filter produced for a negated expression excludes any values which satisfies the expression, effectively using negation by failure.

$\mathcal{E}\in$ In this formulation of the domain relational calculus only unbound identifiers may be asserted to be members of a relation. To represent membership assertions with constants or bound identifiers, $(A_1,\cdots A_n) \in R$, we write $(I_1,\cdots I_n) \in R \wedge I_1 = A_1 \wedge \cdots I_n = A_n$.

$\mathcal{E}\in, \mathcal{E}\exists, \mathcal{E}\forall$ The database d is a free variable of the translation. It is a function from relation identifiers to the list of tuples currently in that relation.

$\mathcal{A}$:**Atom** $\rightarrow$ **Env** $\rightarrow$ (**Env** $\times$ **Simple**)

$\mathcal{A}$ translates a relational calculus primitive into a list comprehension identifier or constant. It also returns an environment indicating whether an unbound identifier has been encountered.

$$\mathcal{A}[\![k]\!]\ \rho = ([], [\![\ k\ ]\!]) \qquad (\mathcal{A}k)$$
$$\mathcal{A}[\![I]\!]\ \rho = ([], [\![\ I\ ]\!]), \text{ if } I \in \rho \qquad (\mathcal{A}I)$$
$$= ([I], [\![\ I\ ]\!]), \text{ otherwise}$$

Θ :**Op** $\rightarrow$ **(Env** $\times$ **Simple)** $\rightarrow$ **(Env** $\times$ **Simple)** $\rightarrow$ **Env** $\rightarrow$ **(Env** $\times$ **Qual)**

Θ translates a relational calculus comparison into a qualifier. If both of the identifiers are bound, the result is a comparison filter. If, however, an unbound identifier is asserted to have a value, the translation produces a generator $a \leftarrow [a']$. In the comprehension this binds a to the values taken on by a'. This is a form of unification.

$$\Theta[\![<]\!]\ ([], a_0)\ ([], a_1)\ \rho = ([], [\![\ a_0 < a_1\ ]\!]) \qquad (\Theta <)$$

Similarly for $>, \leq, \geq, \neq$.

$$\begin{aligned}\Theta[\![=]\!]\ (\rho_0, a_0)\ (\rho_1, a_1)\ \rho\ &= ([], [\![\ a_0 = a_1\ ]\!]), \text{ if } \rho_0 = \rho_1 = [] \\ &= (\rho_0, [\![\ a_0 \leftarrow [a_1]\ ]\!]), \rho_0 \neq [] \\ &= (\rho_1, [\![\ a_1 \leftarrow [a_0]\ ]\!]), \rho_1 \neq []\end{aligned} \qquad (\Theta =)$$

Auxilliary Function

Intersect performs list intersection on a list of lists.

$$intersect\ xss = [x \mid xs \leftarrow xss;\ x \leftarrow xs;\ memberall\ x\ xss]$$

$$memberall\ x\ xss = and\ [member\ x\ xs \mid xs \leftarrow xss]$$

8.5.5 Example

Let us return to the example query and translate it into a comprehension.

$$\mathcal{Q}[\![\{(name) \mid (name, address, balance) \in members \wedge balance < 0\}]\!]$$

$= \{\mathcal{Q}\}$

$[(name) \mid e]$
where
$(\rho, e) = \mathcal{E}[\![(name, address, balance) \in members \wedge balance < 0]\!]\ []$

$= \{\mathcal{E}\wedge\}$

$[(name) \mid e]$
where
$(\rho, e) = (\rho_0 +\!\!+ \rho_1, e_0 +\!\!+ e_1)$
$(\rho_0, e_0) = \mathcal{E}[\![(name, address, balance) \in members]\!]\ []$
$(\rho_1, e_1) = \mathcal{E}[\![balance < 0]\!]\ []$

$= \{\mathcal{E}\in, \mathcal{E}\omega\}$

$[(name) \mid e]$
where
$(\rho, e) = (\rho_0 +\!\!+ \rho_1, e_0 +\!\!+ e_1)$
$(\rho_0, e_0) = ([name, address, balance], (name, address, balance) \leftarrow members)$
$(\rho_1, e_1) = \Theta\ [\![<]\!]\ (\mathcal{A}[\![balance]\!]\rho_0)\ (\mathcal{A}[\![0]\!]\rho_0)\rho_0$

$= \{\mathcal{A}I, \mathcal{A}k\}$

$[(name) \mid e]$
where
$(\rho, e) = (\rho_0 +\!\!+ \rho_1, e_0 +\!\!+ e_1)$
$(\rho_0, e_0) = ([name, address, balance], (name, address, balance) \leftarrow members)$
$(\rho_1, e_1) = \Theta\ [\![<]\!]\ ([], balance)\ ([], 0)\rho_0$

$$= \{\Theta <\}$$

$$[(name) \mid e]$$
where
$$(\rho, e) = (\rho_0 +\!\!+ \rho_1, e_0 +\!\!+ e_1)$$
$$(\rho_0, e_0) = ([name, address, balance], (name, address, balance) \leftarrow members)$$
$$(\rho_1, e_1) = ([], balance < 0)$$

$$= \{\text{Subst. } \rho_0, e_0, \rho_1, e_1\}$$

$$[(name) \mid e]$$
where
$$(\rho, e) = ([name, address, balance],$$
$$(name, address, balance) \leftarrow members; balance < 0)$$

$$= \{\text{Subst. } \rho, e\}$$

$$[(name) \mid (name, address, balance) \leftarrow members; balance < 0]$$

The syntax of the comprehensions produced by the translation is close to that of several programming languages. Hence the resulting comprehensions can be evaluated after trivial syntactic changes. As might be expected the mechanically generated comprehensions are often slightly more complex than a hand crafted comprehension that performs the same task.

8.6 Semantics

As the relational calculus is closely based on set theory it does not need a semantics to describe its meaning. The purpose of the denotational semantics is to bridge a gap between database and programming language theories. The rift occurs because database theory is founded on sets, or relations [32, 91], and relations are not central to any programming language semantics [82].

The semantics presented in Appendix C is denotational. The meaning of a relational calulus query is given as a function that takes a database and

returns the set of tuples in the database that satisfy the query. An interpreter for relational calculus queries has been constructed. The interpreter is closely based on similar semantics to those described here.

The semantics is closely related to the translation into list comprehensions. In some sense the translation of relational calculus queries into list comprehensions, that in turn have a semantics, gives a semantics to the calculus. Indeed a semantics can be derived by composing the translation into comprehensions with the semantics of comprehensions. The resulting semantics is more complex than the semantics given because it distinguishes between translation-time and evaluation-time environments. Also, because the derived semantics is based on lists the tuples are ordered and the possiblity of duplicates is introduced. Both ordering and duplicates are foreign to the notion of relations and neither is introduced by the semantics given. For these reasons a semantics that directly links the calculus to a domain of sets is preferred to the derived semantics.

The potential benefits of such a semantics are well known, but have yet to be exploited. Implementations might be proved to match their denotational specifications. The semantics may suggest new optimisations and existsing optimisations can be proved to preserve correctness. A particularly appealing use might be to give semantics to an extended relational calculus that supports computation and recursive functions.

Chapter 9

Improving Queries

This Chapter covers the improvement of list comprehension queries. For each major improvement strategy identified in the database literature an equivalent improvement is given for comprehension queries. This means that existing database algorithms that improve queries using several of these strategies can be applied to improve comprehension queries. Extra-relational queries can also be improved. An example of each improvement is given.

9.1 Introduction

List comprehensions were recommended as a clear, powerful, concise and well-integrated query notation in the previous Chapter. In this Chapter the sound mathematical basis of comprehensions is used to develop transformations to improve the efficiency of comprehension queries.

The relational database literature cited in Chapter 3 identifies two classes of improvement strategies, algebraic and implementation-based. Algebraic improvements are the result of transforming a query into a more efficient form using identities in the relational algebra. Implementation-based improvements are obtained by using information about how the data is stored. This information may include the size of the relations and what indices exist.

To improve relational queries Ullman [91] identifies four algebraic and two physical implementation strategies. For each of these strategies an equivalent improvement is given for list comprehension queries. As a result existing database algorithms that use these improvements can be followed to improve comprehension queries. Most of the improvements entail transforming a simple, inefficient query into a more complex, but more efficient form. The transformations are presented and illustrated using examples drawn from the database literature. This work has also been reported in [87].

There is a class of useful queries that cannot be expressed in the relational model. These entail recursion or computation and are termed *extra-relational.* The two extra-relational queries from the previous Chapter are improved.

The remainder of this Chapter is structured as follows. Section 9.2 describes the assumptions made about the environment that the queries are evaluated in. Section 9.3 describes the algebraic improvements. Section 9.4 gives an example of emulating an algebraic improvement algorithm. Section 9.5 describes implementation-based improvements. Section 9.6 demonstrates the improvement of extra-relational queries.

9.2 Improvement Environment

Queries are improved under certain assumptions about the environment in which they will be evaluated. Because this work spans the database and programming language worlds, assumptions are made about both. A programming language assumption is that the queries are evaluated under a lazy regime. It is also assumed that the lists processed by the comprehensions represent relations that have been retrieved from secondary, or permanent storage. The remaining assumptions concern the underlying database. They are typical of those found in conventional improvers. Further, the functional database described earlier provides the functionality assumed.

It is assumed that permanent storage is provided by disks. Disks store data as *blocks* or convenient-sized chunks. The *size* of a list is the number of blocks

in it. Each disk access retrieves a block. Thus traversing a list will require a number of disk accesses proportional to its size. It is assumed that the implementation supports cacheing.

The number of disk accesses required to evaluate a query is the cost metric. This is because the time required for an access is typically three or four orders of magnitude greater than the time to execute a machine instruction. As a result it is often faster to perform additional computation if this will reduce the number of accesses required.

It is assumed that the implementation supports indices. An index on a relation typically consists of a sorted tree of values. The tree can be searched for a value in time proportional to the logarithm of the size of the relation. Information such as the size of the relations and the nature of the indices is also assumed to be available.

It is assumed that the order of the tuples in the result of a query is not significant. This is consistent with the relational model, and means that *bag equality*, written $\cong$, can be used between lists. Two lists are bag equal if they contain the same elements, although possibly in different orders.

Because disk access is central to this cost model, not all of the transformations presented in the following Sections will improve comprehensions that do not perform disk accesses. Finally, note that the conventional improvements emulated are not guaranteed to be optimal for all possible instances of the database.

9.3 Algebraic Improvements

This Section contains Subsections describing each of the conventional algebraic improvements and how an equivalent list comprehension improvement is obtained. The description of conventional improvement techniques is closely based on that given by Ullman *op. cit.* Most of the improvements are obtained using transformations that are analogous to identities in the relational algebra. Some of the transformations described are examples of classes of transformations. Not every member of these classes is described.

9.3.1 Selections

Performing selection as early as possible is the most important improvement. It reduces the size of the intermediate results by discarding tuples that are not required. Ullman illustrates this with a query that prints the A components of those tuples in the AB and CD relations that have the same values in B and C, and have a D value of 99. This may be expressed in the relational algebra as $\Pi_A(\sigma_{B=C \wedge D=99}(AB \times CD))$. It can also be written as the comprehension

$$[a \mid (a,b) \leftarrow AB;\ (c,d) \leftarrow CD;\ b = c;\ d = 99].$$

The following transformation, first proposed by P.L. Wadler, can be used to construct queries that perform selections earlier.

Qualifier Interchange states that any two qualifiers q and q' can be swapped, if they don't refer to variables bound in each other. Using $\cong$ to denote bag equality, it may be stated

$$[e \mid q_0;\ q;\ q';\ q_1] \cong [e \mid q_0;\ q';\ q;\ q_1].$$

Rewriting programs so that selection occurs as soon as possible is a well-known program transformation strategy called *filter promotion* [30]. Qualifier interchange is a generalisation of filter promotion as it allows us to change the order of generation as well as the order of filtration. This generality is also reflected by the fact that qualifier interchange is analogous to several relational algebra identities. These are the identities governing the commuting of products and selections. □

In the example above the filters are '$b = c$' and '$d = 99$'. Note that '$b = c$' cannot be promoted over '$(c, d) \leftarrow CD$' because '$b = c$' refers to c which is bound in '$(c, d) \leftarrow CD$'. It is, however, possible to interchange '$d = 99$' and '$b = c$', giving

$$[a \mid (a, b) \leftarrow AB;\ (c, d) \leftarrow CD;\ d = 99;\ b = c].$$

Now, as '$(a, b) \leftarrow AB$' doesn't bind c or d, '$(c, d) \leftarrow CD;\ d = 99$' can be promoted over it, giving

$$[a \mid (c, d) \leftarrow CD;\ d = 99;\ (a, b) \leftarrow AB;\ b = c].$$

This is considerably more efficient than the original query. If the size of AB and CD is n, then the time complexity of the original query is $O(n^2)$. Usually the number of tuples with d value 99 is much smaller than n. If we assume that it is a small constant, i.e. independent of n, the new query is $O(n)$.

9.3.2 Converting Product into Join

If selections are combined with a prior cartesian product to make a join, performance is improved. This is because the cost of a cartesian product of two relations of size n is of $O(n^2)$, whereas the cost of a join, such as a natural join, is usually of $O(n \log n)$. The example query must be manipulated into a suitable form before the product can be converted. The following transformation is used to perform this housekeeping task.

Filter Hiding. Recall from Section 8.3 that $\overline{a}$ denotes a tuple of variables. If f_a is a filter involving only variables in $\overline{a}$, then

$$\begin{array}{l} \quad [e \mid q_0;\ \overline{a} \leftarrow A;\ f_a;\ q_1] \\ = \\ \quad [e \mid q_0;\ \overline{a} \leftarrow A';\ q_1] \\ \qquad \text{where} \\ \qquad\quad A' = [\overline{a} \mid \overline{a} \leftarrow A;\ f_a]. \end{array}$$

Even although the tuples satisfying f_a are drawn from both A and A', lazy evaluation ensures that they are read from secondary storage only once. This

is because, when a demand is made for an element of A', the demand is propagated immediately to a demand for an element of A satsfying f_a. The element of A may need to be retrieved from secondary storage, but once this is done, it can be passed directly to the expression demanding an element of A'. □

Applying filter hiding to '$(c, d) \leftarrow CD;\ d = 99$' in the example query produces

$$[a \mid (c, d) \leftarrow CD';\ (a, b) \leftarrow AB;\ b = c]$$
where
$$CD' = [(c, d) \mid (c, d) \leftarrow CD;\ d = 99].$$

In this instance, filter hiding has neither improved nor degraded the query. The query is, however, now in a form suitable for converting the cartesian product into a join. The transformation to do this is described next.

Product Elimination converts a cartesian product followed by an equality test into a natural join. It is the most common member of a class of transformations that generate the different relational joins.

A natural join takes two relations of arity r and s and constructs a new relation of arity $r + s - 1$, i.e. with one of the identical columns eliminated. To reflect this in the following definition, b_j represents the eliminated column, and $\overline{ab}$ is written for $(a_0, .., a_n, b_0, .., b_{j-1}, b_{j+1}, .., b_m)$. Any reference to the eliminated column must also be relaced by a reference to the identical column, a_i. The substitution of a_i for b_j in an expression e is written $e[a_i/b_j]$. Product elimintation can then be stated

$$[e \mid q_0;\ \overline{a} \leftarrow A;\ \overline{b} \leftarrow B;\ a_i = b_j;\ q_1]$$
$$=$$
$$[e[a_i/b_j] \mid q_0;\ \overline{ab} \leftarrow AB;\ q_1[a_i/b_j]]$$
where
$$AB = jmerge_{ij}\ (sort_i\ A)\ (sort_j\ B).$$

The $jmerge_{ij}$ function is defined as

$$
\begin{aligned}
&jmerge_{ij}\ []\ B = [] \\
&jmerge_{ij}\ A\ [] = [] \\
&jmerge_{ij}\ (\overline{a} : A)(\overline{b} : B) \ = \overline{ab} : jmerge_{ij}\ (\overline{a} : A)\ B, \text{if } a_i = b_j \\
&\qquad\qquad\qquad\qquad\quad = jmerge_{ij}\ (\overline{a} : A)\ B, \text{if } a_i > b_j \\
&\qquad\qquad\qquad\qquad\quad = jmerge_{ij}\ A\ (\overline{b} : B), \text{if } a_i < b_j.
\end{aligned}
$$

The transformation introduces a sort-merge to compute the join. Alternative algorithms could also be used. The $sort_i$ functions sort on ith component of the relation. Some existing solution can be used to resolve the typing problems raised by such joins. □

Applying product elimination to produces

$$
\begin{aligned}
&[a \mid (c, d, a) \leftarrow CDAB] \\
&\quad \text{where} \\
&\qquad CDAB = jmerge_{12}\ (sort_1\ CD')\ (sort_2\ AB) \\
&\qquad CD' = [(c, d) \mid (c, d) \leftarrow CD;\ d = 99].
\end{aligned}
$$

This has complexity of $O(n \log n)$, because AB is still of size n. The desirability of applying product elimination to the example depends on the ratio between $\log n$ and the number of tuples in CD having a d value of 99. Also note how the transformations have taken a simple query and produced a more efficient, but more complex query.

9.3.3 Combination of Unary Operations

In a naive processor, a relation may be traversed for each selection or projection encountered in a query. Efficiency is improved if a sequence of selections and projections can be evaluated in a single pass over a relation. Buneman, Frankel and Nikhil have shown that lazy evaluation causes this to occur automatically in functional query languages [21].

To illustrate the automatic combination, consider one of the examples from Chapter 8,

$$[name\ m \mid m \leftarrow members;\ balance\ m < 0].$$

This performs a select on balance, and a project onto name. Demand for the name is propagated to a demand for a member tuple with a balance less than zero. The next tuple in *members* is obtained, possibly from secondary storage. If the balance is less than zero, the corresponding name can be returned immediately. If the balance exceeds zero, the next tuple in *members* must be examined. The significant point is that the tuple is retrieved only once. The balance test and the projection onto name both occur while the tuple is in primary storage.

9.3.4 Common Subexpressions

It is clearly advantageous to compute only once a result that will be used many times. This is in fact what happens in a functional language with list comprehensions. If e_x is an expression referring to x more than once, and A is the relation produced by some complex computation, then $[e_x \mid x \leftarrow A]$ retains those parts of A that have been realised for some reference to x for as long as there is a reference to them. This is called sharing [70] in the functional language world.

Either *let* or *where* expressions can also be used to preserve common subexpressions, even between comprehensions. For example, consider the improvement of a query that computes the difference between two projections of a join. Using $\bowtie$ to denote join, this can be specified in the algebra as $\Pi_i(A \bowtie B) - \Pi_j(A \bowtie B)$. Writing $-$ for list difference, this might be expressed as

$$[a_i \mid \overline{a} \leftarrow A;\ \overline{b} \leftarrow B;\ a_k = b_l] -$$
$$[a_j \mid \overline{a} \leftarrow A;\ \overline{b} \leftarrow B;\ a_k = b_l].$$

Applying product elimination produces

$$([a_i \mid \overline{ab} \leftarrow AB] \text{ where } AB = jmerge_{kl}\ (sort_k\ A)(sort_l\ B)) -$$
$$([a_j \mid \overline{ab} \leftarrow AB] \text{ where } AB = jmerge_{kl}\ (sort_k\ A)(sort_l\ B)).$$

If we assume that the result of the join is of size n, then it is also reasonable to assume that computing the difference between the projections on the join costs $n \log n$. Under these assumptions the cost of the above query is $3n \log n$, as the join is performed twice before computing the difference. The redundant join can be eliminated using the definitions of where and substitution, to obtain

$$[a_i \mid \overline{ab} \leftarrow AB] - [a_j \mid \overline{ab} \leftarrow AB]$$
where
$$AB = jmerge_{kl}\ (sort_k\ A)(sort_l\ B)).$$

The left-hand comprehension costs $n \log n$ accesses as it constructs and consumes the join simultaneously. The right-hand comprehension need only traverse the result of the join, which costs n accesses. Finally, the difference also costs $n \log n$, giving a total cost of $2n \log n + n$ accesses.

9.3.5 Projections

Queries are improved if projections are performed as soon as possible. A projection reduces the size of the intermediate results because the tuples contain fewer components. Promoting projections is not a major source of improvement, but is included as it is used in Ullman's improvement algorithm which is illustrated in the next Section. A transformation to promote projections is described next.

Shrinking. If $\overline{a}'$ is a tuple containing only those variables of $\overline{a}$ that are free in q_1 or e, then

$$[e \mid \overline{a} \leftarrow A;\ q_1]$$
$$=$$
$$[e \mid \overline{a}' \leftarrow A';\ q_1]$$
where
$$A' = [\overline{a}' \mid \overline{a} \leftarrow A].$$

As described for filter hiding, the construction of the additional list A' is 'free'. □

Examples of the use of shrinking are found in the next Subsection. A projection may result in a relation with duplicate tuples. Removing these reduces the intermediate results further, but is an expensive operation. The cost is high because it is necessary to check that there are no duplicates for each tuple. Most query languages provide an option to remove duplicates. A list duplicate-removal function like *nub* can be used if this is required.

9.4 Algorithm Example

Because all of the main algebraic transformations can be emulated, algorithms that improve queries by applying several of these transformations can be utilised. An algorithm given by Ullman is used as an example. A query produced by the algorithm performs

- Selections as soon as possible.
- Projections as soon as possible.
- Joins in place of cartesian products.
- Sequences of selections and projections in a single pass over a relation.

He illustrates the algorithm using a library database that contains the relations BOOKS, PUBLISHERS, BORROWERS and LOANS. The relations have the following attributes.

BOOKS(TITLE,AUTHOR,PNAME,LC_NO)
PUBLISHERS(PNAME,PADDR,PCITY)
BORROWERS(NAME,ADDR,CITY,CARD_NO)
LOANS(CARD_NO,LC_NO,DATE)

To keep track of books, there is a view, or useful combination of these relations, called XLOANS. XLOANS is the natural join of BOOKS, BORROWERS and LOANS, and might be defined in the relational algebra as

$$\Pi_S(\sigma_F(\text{LOANS} \times \text{BORROWERS} \times \text{BOOKS}))$$
where
F = BORROWERS.CARD_NO = LOANS.CARD_NO
and BOOKS.LC_NO = LOANS.LC_NO
S = TITLE, AUTHOR, PNAME, LC_NO, NAME,
ADDR, CITY, CARD_NO, DATE

To list the books that have been borrowed before some date in the past, say 1/1/82, we might write $\Pi_{TITLE}(\sigma_{DATE<1/1/82}(\text{XLOANS}))$. The equivalent naive list comprehension query is

$$[(title) \mid (card_no, llc_no, date) \leftarrow loans;$$
$$(name, addr, city, bcard_no) \leftarrow borrowers;$$
$$(title, author, pname, bklc_no) \leftarrow books;$$
$$bklc_no = llc_no;\ bcard_no = lcard_no;\ date < 1/1/82].$$

Ullman's algorithm starts by moving the selections to occur as soon as possible. This can be emulated by promoting '$bcard_no = lcard_no$' and '$date < 1/1/82$' to obtain

$$[(title) \mid (card_no, llc_no, date) \leftarrow loans;\ date < 1/1/82;$$
$$(name, addr, city, bcard_no) \leftarrow borrowers;\ bcard_no = lcard_no;$$
$$(title, author, pname, bklc_no) \leftarrow books;\ bklc_no = llc_no].$$

Applying filter hiding to *loans* produces

$$[(title) \mid (card_no, llc_no, date) \leftarrow loans';$$
$$(name, addr, city, bcard_no) \leftarrow borrowers;\ bcard_no = lcard_no;$$

$$(title, author, pname, bklc_no) \leftarrow books;\ bklc_no = llc_no]$$

where

$$loans' = [(lcard_no, llc_no, date) \mid (lcard_no, llc_no, date) \leftarrow loans;\ date < 1/1/82].$$

Ullman's algorithm now promotes all of the projections. Applying shrinking to '$b \leftarrow borrowers$' produces

$$[(title) \mid (card_no, llc_no, date) \leftarrow loans';$$
$$(bcard_no) \leftarrow borrowers';\ bcard_no = lcard_no;$$
$$(title, author, pname, bklc_no) \leftarrow books;\ bklc_no = llc_no]$$

where

$$loans' = [(lcard_no, llc_no, date) \mid (lcard_no, llc_no, date) \leftarrow loans;\ date < 1/1/82]$$
$$borrowers' = [(bcard_no) \mid (name, addr, city, bcard_no) \leftarrow borrowers].$$

Applying shrinking to *loans'* and *books* produces

$$[(title) \mid (card_no, llc_no) \leftarrow loans'';$$
$$(bcard_no) \leftarrow borrowers';\ bcard_no = lcard_no;$$
$$(title, bklc_no) \leftarrow books';\ bklc_no = llc_no]$$

where

$$loans' = [(lcard_no, llc_no, date) \mid (lcard_no, llc_no, date) \leftarrow loans;\ date < 1/1/82]$$
$$loans'' = [(lcard_no, llc_no) \mid (lcard_no, llc_no, date) \leftarrow loans']$$
$$borrowers' = [(bcard_no) \mid (name, addr, city, bcard_no) \leftarrow borrowers]$$
$$books' = [(title, bklc_no) \mid (title, author, pname, bklc_no) \leftarrow books].$$

Product elimination can now be used to convert the product of *loans''* and *borrowers'* into a natural join, giving

$$[(title) \mid (card_no, llc_no) \leftarrow lb;$$
$$(title, bklc_no) \leftarrow books';\ bklc_no = llc_no]$$

where

$$loans' = [(lcard_no, llc_no, date) \mid (lcard_no, llc_no, date) \leftarrow loans;\ date < 1/1/82]$$
$$loans'' = [(lcard_no, llc_no) \mid (lcard_no, llc_no, date) \leftarrow loans']$$

$$
\begin{aligned}
borrowers' &= [(bcard_no) \mid (name, addr, city, bcard_no) \leftarrow borrowers] \\
books' &= [(title, bklc_no) \mid (title, author, pname, bklc_no) \leftarrow books] \\
lb &= jmerge_{11}\ (sort_1\ loans'')(sort_1\ borrowers')\ .
\end{aligned}
$$

Applying shrinking to lb gives

$$
\begin{aligned}
&[(title) \mid (llc_no) \leftarrow lb';\ (title, bklc_no) \leftarrow books';\ bklc_no = llc_no] \\
&\text{where} \\
&\quad loans' = [(lcard_no, llc_no, date) \mid (lcard_no, llc_no, date) \leftarrow loans;\ date < 1/1/82] \\
&\quad loans'' = [(lcard_no, llc_no) \mid (lcard_no, llc_no, date) \leftarrow loans'] \\
&\quad borrowers' = [(bcard_no) \mid (name, addr, city, bcard_no) \leftarrow borrowers] \\
&\quad books' = [(title, bklc_no) \mid (title, author, pname, bklc_no) \leftarrow books] \\
&\quad lb = jmerge_{11}\ (sort_1\ loans'')(sort_1\ borrowers') \\
&\quad lb' = [(llc_no) \mid (lcard_no, llc_no) \leftarrow lb].
\end{aligned}
$$

Product elimination again produces the final result, which has the same evaluation plan as the query produced by Ullman's algorithm. That is, it performs the same operations in the same order.

$$
\begin{aligned}
&[(title) \mid (llc_no, title) \leftarrow lbbk] \\
&\text{where} \\
&\quad loans' = [(lcard_no, llc_no, date) \mid (lcard_no, llc_no, date) \leftarrow loans;\ date < 1/1/82] \\
&\quad loans'' = [(lcard_no, llc_no) \mid (lcard_no, llc_no, date) \leftarrow loans'] \\
&\quad borrowers' = [(bcard_no) \mid (name, addr, city, bcard_no) \leftarrow borrowers] \\
&\quad books' = [(title, bklc_no) \mid (title, author, pname, bklc_no) \leftarrow books] \\
&\quad lb = jmerge_{11}\ (sort_1\ loans'')(sort_1\ borrowers') \\
&\quad lb' = [(llc_no) \mid (lcard_no, llc_no) \leftarrow lb] \\
&\quad lbbk = jmerge_{12}\ (sort_1\ lb')(sort_2\ books').
\end{aligned}
$$

9.5 Implementation-based Improvements

9.5.1 Preprocessing Files

The most important file processing ideas are sorting and the creation of indices. The product elimination transformation illustrated the introduction of sorting. Recall that the implemention of secondary indices in a functional database was described in Chapter 6. A transformation that allows an index to be used is presented below.

Index introduction. If e' is an expression, and there is an index $jindex_A$ on an attribute a_j, then

$$\begin{array}{l} \quad [e \mid q_0;\ \overline{a} \leftarrow A;\ a_j = e';\ q_1] \\ = \\ \quad [e \mid q_0;\ \overline{a} \leftarrow jindex_A\ \ e';\ q_1]. \ \square \end{array}$$

In the example from Subsection 9.3.1,

$$[a \mid (c,d) \leftarrow CD;\ d = 99;\ (a,b) \leftarrow AB;\ b = c]$$

index indroduction can be applied to '$(c,d) \leftarrow CD;\ d = 99$', to obtain

$$[a \mid (c,d) \leftarrow dindex_{CD}\ 99;\ (a,b) \leftarrow AB;\ b = c].$$

A second application gives

$$[a \mid (c,d) \leftarrow dindex_{CD}\ 99;\ (a,b) \leftarrow bindex_{AB}\ \ c].$$

This is a very efficient form of the query. If we continue to assume that the number of CD tuples with d value 99 is constant, then only a constant

number of $bindex_{AB}$ lookups need be performed. Each lookup requires $\log n$ accesses giving a total cost of $O(\log n)$.

Another file processing example Ullman gives is an efficiency improvement for cartesian products. With cacheing, efficiency is improved by choosing the smaller relation to vary more slowly, or be in the 'outer loop'. This is because, if the records of the smaller relation are cached, then, as they are combined with each record in the larger relation, they are used more often.

To make the improvement, the smaller relation's generator can be promoted using qualifier interchange. This is another example of the general power of qualifier interchange. If L is the larger relation, S is the smaller, and $\bar{ls}$ is written for $(l_0, .. , l_n, s_0, .. , s_m)$, then

$$[\bar{ls} \mid \bar{l} \leftarrow L; \bar{s} \leftarrow S]$$

becomes

$$[\bar{ls} \mid \bar{s} \leftarrow S; \bar{l} \leftarrow L].$$

9.5.2 Evaluating Options

It is often possible to compute a result in more than one way, either by reordering operations or by treating the operands of a binary operator differently. Time spent evaluating these options is usually much less than the time spent evaluating the query in an inferior way. Usually the cost of a large number of alternatives is considered, and the best of these is selected. As an example Ullman presents the options evaluated for simple selections in System R [6]. These have the form

SELECT A_1, ... A_n
FROM R
WHERE P_1 AND ... P_n.

The equivalent list comprehension form is $[\overline{a} \mid \overline{a} \leftarrow R;\ P_1;\ \ldots\ P_n]$.

To compare the evaluation options the system uses the following information.

- T, the number of tuples in R.
- B, the number of blocks in R.
- I, if there is an index on attribute a_j, the image size I is the number of different values of a_j in R.
- Whether or not an index is clustering. A clustering index is an index on an attribute such that tuples with the same value for that attribute reside in the same block.

A predicate of the form '$a_j = c$', where a_j is an attribute and c a constant is said to *match* an index on a_j. Ullman uses the Happy Valley Farmers Coop database described in Chapter 8 and improves a query that prints the order numbers of any orders for more than 5 pounds of Granola. A list comprehension expressing this is

$[order_no \mid (order_no, name, item, qty) \leftarrow orders;\ qty \geq 5;\ item = \text{`}Granola\text{'}]$.

The database parameters are that $T = 1000$, $B = 100$, there is a clustering index on '*name*', and a nonclustering indices on '*item*' ($I = 50$) and '*quantity*'. The alternatives in the System R algorithm that are relevant for the storage methods described in earlier Chapters are:

1. Get those tuples of R that satisfy a predicate of the form '$a_i = c$' that match a clustering index. Then apply the remaining predicates. This costs B/I block accesses. In the above example, if the '*item*' index was clustering, this would be 2 accesses. It cannot be applied as the item index, which is the only index matched by an equality predicate '*item* ='*Granola*", is not clustering.

2. Use a clustering index on a_i, where '$a_i\ \omega\ c$' is a predicate, and ω is $<, \leq, \geq$ or $>$ to obtain the subset of R that satisfies this predicate, then apply the remaining predicates. This costs $B/2$ block accesses, or 50 block accesses in the example. It cannot be applied as the *quantity* index, which is the only index matched by an inequality predicate '$qty \geq 5$', is not clustering.

3. Use a non-clustering index that matches a predicate '$a_i = c$' to find all of the tuples with a_i value c, and apply the other predicates to these tuples. This costs T/I, or 20 accesses. The item index (*anitemsorders*) fulfills these conditions.

4. Read all of the tuples of R and apply the predicates to each of them. This costs B block accesses, i.e. 100 in the example. It corresponds to the first comprehension in this Section.

In this case option 3 is best. To introduce the index we must first juxtapose the enumeration of orders, '$(order_no, name, item, qty) \leftarrow orders$', and the selection on the item ordered, '$item$ ='*Granola*". This can be achieved by using qualifier interchange to promote '$qty \geq 5$' over '$item$ ='*Granola*", giving

$$[order_no \mid (order_no, name, item, qty) \leftarrow orders;\ item = \text{'}Granola\text{'};\ qty \geq 5].$$

Index introduction now allows us to use *anitemsorders*, giving

$$[order_no \mid (order_no, name, item, qty) \leftarrow anitemsorders\ \text{'}Granola\text{'};\ qty \geq 5].$$

9.6 Extra-Relational Queries

Improvements based on implementation information are particularly useful for improving queries that are more complex than those permitted in the relational model. The extra-relational parts of these queries are not amenable to relational algebra transformations. In this Section the two extra-relational queries from Section 8.3 are improved.

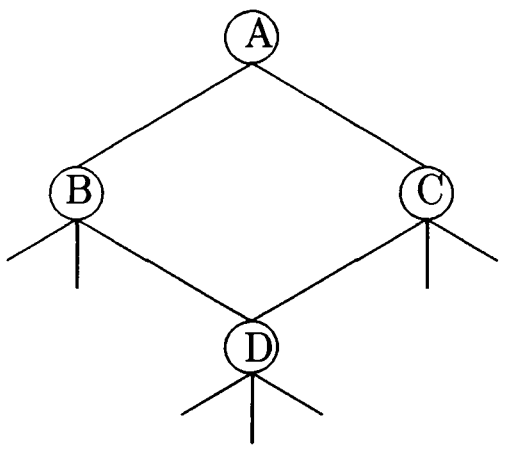
9.6.1 Date's Example

The list comprehension solution to Date's recursive query was written

$$explode\ main\ = [p \mid (m, s, q) \leftarrow parts;\ m = main;\ p \leftarrow (s : explode\ s)].$$

One source of inefficiency in *explode* arises because a bill of material is a directed acyclic graph (DAG). As written, *explode* will revisit any subcomponent that is common to two or more components in the bill. In the bill sketched in Figure 9.1, Node D, and its subcomponents, will be visited in the processing of both node B and node C.

Figure 9.1



The redundant processing can be eliminated by memoising *explode*. Recall the description of memoisation given in Subsection 6.5.1. On encountering a node that has already been processed, a memoised instance of *explode* can simply lookup the value already computed and need not reprocess the node.

As written, *explode* scans the entire relation to find the immediate subcomponents of each main component in the bill. Hence if the size of the parts relation is n, and the number of nodes in the bill being exploded is m then the memoised explosion requires mn block accesses. Date's SQL solution has a SELECT statement that locates the subcomponents of each main component. As described in the previous Subsection, the behaviour of such constructs depends on the existence and nature of indices. If there is an index *uses* on main components index introduction can be applied to obtain

$$explode\ main\ = [p \mid (m, s, q) \leftarrow uses\ main;\ p \leftarrow (s : explode\ s)]$$

This is far more efficient as it simply looks up the subcomponents of a part without having to scan the entire relation for them. If we assume that there are nearly as many main components as there are parts, then the index lookup requires $O(\log n)$ block accesses. As lookup is performed for each node in the bill in turn, the total cost is $O(m \log n)$ accesses.

9.6.2 Atkinson and Buneman's Example

Atkinson and Buneman's recursive and computational query was written

$$costandmass\ p = cm\ (lookup\ p\ parts)$$

$$cm\ (Base\ p\ c\ m\ ss) = (c, m)$$
$$cm\ (Comp\ p\ pqs\ ci\ mi)$$
$$= sumpair\ ((ci, mi) : [(q * c, q * m) \mid (p, q) \leftarrow pqs;\ (c, m) \leftarrow [costandmass\ p]]).$$

A minor improvement can be obtained by rewriting the *sumpair* function to use an accumulating parameter and hence to scan the list of cost, mass pairs only once. The *costandmass* function can also be memoised to avoid the recomputation of common subcomponents. The memoised solution meets all four efficiency subgoals that Atkinson and Buneman identify for the query. These are that the solution should

- avoid repeated recomputation of costs and masses of common subcomponents.
- compute the costs and masses in a single pass over the data structure.
- provide index support to locate the part to be exploded.
- not have to compute the cost and mass of every bill in the parts relation when only one is required.

The efficiency of the memoised version of *costandmass* can be calculated using m and n as defined in the previous Subsection. A lookup is performed to locate each of the m nodes in the bill. Each lookup costs $\log n$ accesses, giving a total cost of $m \log n$.

For the purposes of comparison let us examine the efficiency of the PS-Algol solution Atkinson and Buneman describe. Like the list comprehension solution, the PS-Algol solution is memoised to avoid recomputation of common subcomponents. The PS-Algol solution performs a linear search for the part to be exploded, requiring $n/2$ accesses. Once the target part is located, however, direct links are followed from composite parts to their subcomponents. Such links can usually be followed in a single access. Hence, once the target part is located, visiting each node in the bill requires m accesses. Thus the total cost to locate and then explode the part is $n/2 + m$ accesses. The cost of locating the part to be exploded could be reduced to $\log n$ by introducing a tree structure. This would, however, increase the complexity of the code required to express the query.

The efficiency of the PS-Algol solution compared with the comprehension solution depends on the ratio between m and n. The comprehension solution is faster if the parts relation, i.e. n, is large relative to the bill being exploded, i.e. m. To make these terms more concrete, consider the parts relation to be the tree-file from Subsection 4.3.2 that contains 10^4 records. In this case the comprehension solution is faster if the bill being exploded has fewer than 2469 parts.

Queries over non-destructively maintained data structures cannot be made as efficient as queries over destructively maintained structures. In Chapter 6 non-destructive update was shown to preclude certain data structures. In a destructive world closely linked data structures can be maintained. These links, or pointers can be followed in a single access. To represent the same structure in a non-destructive world keys must be stored, and the cost of an index lookup incurred.

To illustrate this last point consider a destructive update solution that might be constructed for Atkinson and Buneman's example. The part relation may be stored in a hash table with each component linked to its subcomponents. Using the hash table the part to be exploded can be located in a constant

number of accesses. Once located, the bill can be traversed in just m accesses. This gives a total cost of $O(m)$. This destructive cost differs from the non-destructive cost of $O(m \log n)$ by a factor of $\log n$, exactly the additional cost of performing the index lookups.

Part V

Conclusion

Chapter 10

Conclusion

This Chapter summarises the results reported in the Thesis and concludes that functional languages have potential as database implementation, manipulation and query languages. Further research directions are also identified.

10.1 Summary

Preserving referential transparency is seen as the property that distinguishes functional languages from procedural languages. By examining a database implemented, manipulated and queried in a functional language the consequences of enforcing referential transparency in database languages have been explored.

To discover the impact of referential transparency in the implementation language a database manager has been constructed in a pseudo-parallel functional language. The manager supports efficient concurrent operations on large data structures and allows a version of the structure, or database, to be preserved cheaply. Some problems that seriously restrict concurrency have been overcome using new and existing primitives. Data dependency has been shown to offer a novel exclusion mechanism that allows an unusual degree of concurrency compared with conventional schemes such as locking. Support

for the relational and functional data models has also been demonstrated.

Transactions have been written as functions over the database. These functions are made atomic using the cheap multiple versions of the database generated under a non-destructive update regime. The transaction-functions provide concurrent and consistent manipulation of long-term data within the functional model of computation. The power and mathematical tractability of transaction functions has also been demonstrated.

List comprehensions are a referentially transparent query notation that other workers have recommended as clear, powerful, concise, mathematically sound and well integrated with its host language. The argument for the clarity of comprehension queries has been reinforced by illustrating their close resemblance to the relational calculus. The power, or relational *completeness* of list comprehensions has been proved. Database and programming language theory have been further integrated by describing the relational calculus in a programming language semantics. The sound mathematical basis of comprehensions has been used to develop transformations that improve the efficiency of list comprehension database queries.

In conclusion, fast evaluation and the ease of transformation make preserving referential transparency in a query language desirable. The suitability of referentially transparent languages for implementing and manipulating databases is less clear. The transaction language is attractive because of its power and mathematical tractability. It is, however, dependent on the implementation language providing cheap multiple versions of the database.

As an implementation language, a parallel functional language has sufficient concurrency and clean semantics. Access to some important data structures can be implemented efficiently — classes of data and secondary indices are two examples. However, the non-destructive update regime limits the choice of data structures to those that can be modified efficiently. Some desirable data structures, such as closely-linked graphs, cannot be modified efficiently and hence cannot be used in a functional database. The author believes that the data structures that can be modified efficiently are sufficient to support most database applications with acceptable efficiency. A more realistic implementation would provide a better understanding of the costs and benefits of enforcing referential transparency in the implementation and transaction

languages.

10.2 Future Directions

Three avenues of further research present themselves. The first is to construct a more realistic implementation of the functional database. The second is to explore the potential of data dependency as an exclusion mechanism. The third is to explore the costs and benefits of non-destructive update.

The first avenue is to provide more concrete evidence of the practical value of a functional database by constructing a more realistic implementation. The results obtained from a prototype implementation are promising. The existing implementation could be extended by implementing the *fwif* and *fwof* parallelism primitives. This would enable the parallelism possible to be further investigated and the primitives compared. The implementation is, however, far from realistic as the amount of data stored is small and the multiple processors are only simulated. Implementing a larger database on a multi-processor machine would provide more believable evidence of its practicality.

The practicality of the query notation can also be established. The notation and associated transformations are an attractive combination. List comprehensions are being included in a variant of PS-Algol that is being constructed by workers at Glasgow University. The comprehensions will be used to provide an object-oriented query language. The implementation should permit the interrogation of several megabytes of persistent data.

The second avenue to be explored is the non-destructive update model. The investigation might include discovering further uses for the multiple copies of the database generated by non-destructive update. For example the database might be made resilient in the face of machine failure. Further investigation is required to ascertain which data structures can be efficiently modified under a non-destructive update regime and which cannot.

The third avenue is to explore the exclusion provided by data dependency. A detailed comparison with conventional mechanisms is desirable. Data depen-

dency has several desirable properties that are immediately apparent, such as an unusual degree of concurrency and deadlock freedom. Further investigation may uncover other properties. It may be possible to convince the Flagship design team to use data dependent exclusion in their DebitCredit functional database. It may also be possible to prove that the exclusion mechanism is optimal in the sense that access is only prevented to those parts of the entities that are currently being modified.

Appendix A

Parallel Programs

This Appendix presents the concurrent LML programs. The LML programs are compiled into FLIC intermediate code [71]. An interpreter has been written to simulate the parallel reduction of FLIC code. Eager and optimistic primitives are added to the compiler-generated FLIC manually, and the LML programs below are annotated by [!] to indicate where this has occured.

The bulk data manager, associated operations like lookup, and the account database all reside in a seperate module. This module is linked into programs that use the operations it provides. Appendix A.1 gives two versions of the bulk data manager module, although not all of the example database is included. Appendix A.2 presents the programs that invoke a sequence of bulk data operations. Appendix A.3 presents programs that invoke a sequence of transactions.

A.1 Bulk Data Manager

A.1.1 Standard Bulk Data Manager

```
module
--      This module provides the bulk data manipulating functions
--      with associated types and an example class of data called
--      acct.

export  Rt, Dbt, Messaget, key, snd, fst, lookup, update, delete,
        manager, acct;

rec
        (type Rt = record Int    Int     Char  Int)
--                        Acctno Balance Class Credit-Limit
and     (type Dbt = tip Rt + node Dbt Int Dbt)
and     (type Messaget = error (List Char) Int +    --Error with flag,
                                                    --key + explanation
                         ok (List Char)  Rt)        --Positive ack with
                                                    --explanatory note

--      key :: Rt -> Int
--      Extracts the key from a record, an integer in this case.

and     key (record a b c d) = a

and     snd (x,y) = y
and     fst (x,y) = x

--      lookup :: Int -> Dbt -> Out
--      Given a key and a database this retrieves the record
--      associated with that key in the obvious way.
```

```
and     lookup k' d = let rec
                        lookup' k' (tip r) =  ok "lkup, record = " r
                      ||
                        lookup' k' (node lt k rt) =
                          if k' < k then
                            lookup' k' lt
                          else
                            lookup' k' rt
                      in
                        (lookup' k' d,d)

--      insert :: Rt -> Dbt -> Out
-----------------------------------
--      Insert constructs a new database which contains an
--      additional record. It thcweamths and thcweamths if the
--      record already exists.

and     insert r' (tip r) =
           if key r' = key r then
             (error "ins, rec exists= " (key r), tip r)
           else
             if key r' < key r then
               (ok "ins, " r', node (tip r') (key r) (tip r))
             else
               (ok "ins, " r', node (tip r) (key r') (tip r'))
||      insert r' (node lt k rt) =
          if key r' < k then
            let
              (m,lt') = insert r' lt
            in
              (m, node lt' k rt)
          else
            let
              (m,rt') = insert r' rt
            in
              (m, node lt k rt')
```

```
--      update :: Rt -> Dbt -> Out
---------------------------------
--      Update constructs a new database containing a
--      different record in place of an original record.

and     update r' (tip r) = (ok "upd " r', tip r')
||      update r' (node lt k rt) =
          if key r' <  k then
            let
              (m,lt') = update r' lt
            in
              (m, node[!] lt' k rt)
          else
            let
              (m,rt') = update r' rt
            in
              (m, node[!] lt k rt')

--      delete :: Int -> Dbt -> Out
---------------------------------
--      Delete constructs a new database which is
--      identical to the original except it excludes
--      a record. It examines 4 cases - a node with
--      2 tips, a left tip, a right tip, or two nodes
--      for children.

and     delete k' (node (tip r0) k (tip r1)) =
            if k' = key r0  then
                (ok "del " r0, tip r1)
            else
              if k' = key r1  then
                (ok "del " r1, tip r0)
              else
                (error "del- key missing " k',
                  node (tip r0) k (tip r1))
||      delete k' (node (node lt k2 rt) k (tip r)) =
          if k' < k then
```

```
            let
              (m,lt') = delete k' (node lt k2 rt)
            in
              (m, node lt' k (tip r))
          else
            if k' = key r then
              (ok "del " r, node lt k2 rt)
            else
              (error "del- key missing " k',
                node (node lt k2 rt) k (tip r))
||      delete k' (node (tip r) k (node lt k2 rt)) =
          if k' < k then
            if k' = key r then
              (ok "del " r, node lt k2 rt)
            else
              (error "del- key missing " k',
                node (tip r) k (node lt k2 rt))
          else
            let
              (m,rt') = delete k' (node lt k2 rt)
            in
              (m, node (tip r) k rt')

||      delete k' (node lt k rt) =
          if k' <  k then
            let
              (m,lt') = delete k' lt
            in
              (m, node lt' k rt)
          else
            let
              (m,rt') = delete k' rt
            in
              (m, node lt k rt')
```

```
--      manager :: Dbt -> (List Dbt ->
---------------------------------------
--                      (Messaget >< Dbt)) -> List Out
--                      ------------------------------
--      Manager is a stream processing function that
--      consumes a stream of database manipulating
--      functions and produces a stream of
--      outputs. It retains control of the tree.

and     manager d (f.fs) = let
                             (m,d') = f d
                           in
                             (m .[!] manager d' fs)
||      manager d [] = []

and     acct = node (node (node (node (node (node
        (node (node (node (tip (record 1000 24 'B' 100))
                          1010
                        (tip (record 1010 523 'D' 500)))
                    1020
                   (node (tip (record 1020 37 'A' 50))
                         1030
                        (tip (record 1030 (-33) 'E' 50))))
                1040
               (node (node (tip (record 1040 (-51) 'B' 150))
                           1050
                          (tip (record 1050 1022 'A' 500)))
                     1060
                    (node (tip (record 1060 75 'A' 150))
                          1070
                         (tip (record 1070 381 'C' 250)))))

-- ... 512 account records ...

                1040
               (node (node (tip (record 8740 (-51) 'B' 850))
                           8750
```

```
                        (tip (record 8750 8022 'A' 500)))
                  8760
                 (node (tip (record 8760 75 'A' 850))
                       8770
                       (tip (record 8770 381 'C' 850)))))))))
end
```

A.1.2 Bulk Data Manager with Disk Delay

In order to simulate the effect of disk-delayed access to the entities at the leaves of the tree the following *access* function is added to the manager and both *lookup* and *update* are modified. The *access* function simply wastes time by counting.

```
        access 0 = 1
||      access n = access (n-1)

--      lookup :: Int -> Dbt -> Out
--      Given a key and a database this retrieves the record
--      associated with that key in the obvious way. Screams
--      if there is no such record. Incorporates two calls
--      to the access function to simulate a disk delay.

and     lookup k' d =
          let rec
            lookup' k' (tip r) =
              if (key r = k') & (access 50) = (access 50) then
                ok "lkup, record = " r
              else
                ok "lkup, record = " r
            ||
            lookup' k' (node lt k rt) =
              if k' < k then
                lookup' k' lt
```

```
              else
                lookup' k' rt
          in
            (lookup' k' d,d)

--      update :: Rt -> Dbt -> Out
---------------------------------
--      Update constructs a new database containing a
--      different record in place of an original record.
--      Incorporates two calls to the access function to
--      simulate a disk delay.

and     update r' (tip r) =
          ([!] ok "upd " r',
            tip[!] (if (access 50) = (access 50) then r' else r'))
||      update r' (node lt k rt) =
          if key r' <  k then
            let
              (m,lt') = update r' lt
            in
              (m, node[!] lt' k rt)
          else
            let
              (m,rt') = update r' rt
            in
              (m, node[!] lt k rt')
```

A.2 Bulk Data Operations

A.2.1 Lookups

```
--      This program performs 30 lookups to different entities.

#include "esops.t";

        manager acct [lookup 8230;
                      lookup 1540;
                      lookup 3730;
                      lookup 5610;
                      lookup 6300;
                      lookup 7530;
                      lookup 2670;
                      lookup 4750;
                      lookup 1060;
                      lookup 8050;
                      lookup 4230;
                      lookup 5730;
                      lookup 6370;
                      lookup 8650;
                      lookup 7560;
                      lookup 4350;
                      lookup 1430;
                      lookup 3230;
                      lookup 8550;
                      lookup 1670;
                      lookup 2340;
                      lookup 5350;
                      lookup 3450;
                      lookup 6670;
                      lookup 7750;
                      lookup 4350;
                      lookup 8560;
```

```
                    lookup 6570;
                    lookup 2750;
                    lookup 5640]
```

A.2.2 Updates

```
--      This program performs 30 updates of the same entity

#include "esops.t";

        manager acct [update  (record 1400 745 'A' 40);
                      update  (record 1400 345 'A' 40);
                      update  (record 1400 745 'A' 40);
                      update  (record 1400 345 'A' 40);
                      update  (record 1400 345 'A' 40);
                      update  (record 1400 745 'A' 40);
                      update  (record 1400 345 'A' 40);
                      update  (record 1400 745 'A' 40);
                      update  (record 1400 345 'A' 40);
                      update  (record 1400 345 'A' 40);
                      update  (record 1400 745 'A' 40);
                      update  (record 1400 345 'A' 40);
                      update  (record 1400 745 'A' 40);
                      update  (record 1400 745 'A' 40);
                      update  (record 1400 345 'A' 40);
                      update  (record 1400 745 'A' 40);
                      update  (record 1400 345 'A' 40);
                      update  (record 1400 345 'A' 40);
                      update  (record 1400 745 'A' 40);
                      update  (record 1400 345 'A' 40);
                      update  (record 1400 745 'A' 40);
                      update  (record 1400 345 'A' 40);
                      update  (record 1400 345 'A' 40);
                      update  (record 1400 745 'A' 40);
```

```
                    update  (record 1400 345 'A' 40);
                    update  (record 1400 745 'A' 40);
                    update  (record 1400 345 'A' 40);
                    update  (record 1400 345 'A' 40);
                    update  (record 1400 345 'A' 40)]
```

A.2.3 Updates with Disk Delay

```
--        This program performs 15 updates of different entities
--        using the data manager with a simulated disk delay.
--        Note that the data manager with the disk delay has the
--        same type as the standard module with no delay.

#include "esops.t";

        manager acct [update  (record 1010 745 'A' 40);
                      update  (record 3230 345 'A' 40);
                      update  (record 5320 745 'A' 40);
                      update  (record 8450 345 'A' 40);
                      update  (record 6540 345 'A' 40);
                      update  (record 3710 745 'A' 40);
                      update  (record 7330 345 'A' 40);
                      update  (record 1620 745 'A' 40);
                      update  (record 6750 345 'A' 40);
                      update  (record 4440 345 'A' 40);
                      update  (record 4750 745 'A' 40);
                      update  (record 7570 345 'A' 40);
                      update  (record 2640 745 'A' 40);
                      update  (record 1460 345 'A' 40);
                      update  (record 8650 345 'A' 40)]
```

A.2.4 Read and Write Programs

This Subsection presents four programs that illustrate the possible combinations of read and write operations. The programs use the data manager with a simulated disk delay. Note that the data manager with the disk delay has the same type as the standard module with no delay.

```
-- This program performs an update (write) followed by a
-- lookup (read) of the same entity.

#include "esops.t";

        manager acct [update  (record 1560 345 'A' 40);
                      lookup 1560]

--        This program performs a lookup (read) followed by an
--        update (write) of the same entity.

#include "esops.t";

        manager acct   [lookup 1560;
                        update  (record 1560 345 'A' 40)]

--        This program performs two lookups (reads) of the same entity.

#include "esops.t";

        manager acct   [lookup 1300;
                        lookup 1300]

--        This program performs two updates (writes) of the same entity.

#include "esops.t";

        manager acct [update  (record 1400 745 'A' 40);
                      update  (record 1400 345 'A' 40)]
```

A.2.5 Typical Mix

```
--        This program performs a sequence of 30 insert, delete, lookup
--        and update operations representative of a typical mix of
--        operations for some application.

#include "esops.t";

        manager acct
                [ update  (record 2770 746 'A' 40); lookup 2770;
                  insert  (record 4630 445 'A' 40); lookup 4630;
                  update  (record 6200 746 'A' 40);
                  insert  (record 1470 746 'A' 40); delete 1470;
                  lookup 7410;
                  update  (record 7310 446 'A' 40); lookup 7310;
                  update  (record 8560 446 'A' 40);
                  update  (record 2770 745 'A' 40); lookup 2770;
                  lookup 3660;
                  insert  (record 5200 745 'A' 40);
                  lookup 1260;
                  insert  (record 2310 342 'A' 40); delete 2310;
                  update  (record 4630 442 'A' 40); lookup 4630;
                  update  (record 6300 746 'A' 40);
                  insert  (record 5470 746 'A' 40); delete 1470;
                  lookup 7410;
                  update  (record 7370 446 'A' 40); lookup 7310;
                  update  (record 8260 446 'A' 40);
                  lookup 1360;
                  delete 5310;
                  update  (record 1260 342 'A' 40)]
```

A.3 Transactions

A.3.1 Five Bank Transactions

```
--        This program defines a bank deposit transaction.
--        It then performs 5 bank transactions - two deposits and
--        three balance enquires.

#include "esops.t";

let rec

        isok (ok m r) = true
||      isok msg = false

and     deprec (ok m (record ano bal class crl)) n =
          (record ano (bal+n) class crl)
||      deprec (error m k) n = (record 0 0 'A' 0)

and     dep a n d =
          let (o1,d1) = lookup a d in
          let (o2,d2) = update (deprec o1 n) d in
            if[!] (isok o1) & (isok o2) then
              (o2,d2)
            else
              (error "dep" 0,d)

in      manager acct   [dep 1600 10;
                        lookup 5250;
                        lookup 7530;
                        dep 4320 5;
                        lookup 2730]

--        This program performs the same operations as the above
--        program, except the operations are not packaged up as
```

```
--        transactions.

#include "esops.t";

        manager acct   [lookup 1600;
                        update (record 1600 345 'A' 40);
                        lookup 5250;
                        lookup 7530;
                        lookup 4320;
                        update (record 4320 213 'B' 50);
                        lookup 2730]
```

A.3.2 Two Long Transactions

```
--        This program performs two transactions both of which
--        update the same four entities.

#include "esops.t";

let rec
        isok (ok m r) = true
||      isok msg = false

and     lots d = let (o1,d1) = update (record 1040 320 'A' 150) d  in
                 let (o2,d2) = update (record 1240 320 'A' 150) d1 in
                 let (o3,d3) = update (record 1440 320 'A' 150) d2 in
                 let (o4,d4) = update (record 1640 320 'A' 150) d3 in
                 if[!] (isok o1) & (isok o2) & (isok o3) & (isok o4) then
                   (o4,d4)
                 else
                   (error "lots" 0,d)

in      manager acct [lots;
                      lots]
```

A.3.3 Two Transactions with a failing *Optif*

```
--        This program performs two transactions both of which update
--        four entities. The second update of 'lots2' fails because
--        the entity does not exist.

#include "esops.t";

let rec
        isok (ok m r) = true
||      isok msg = false

and     lots d = let (o1,d1) = update (record 1040 320 'A' 150) d  in
                 let (o2,d2) = update (record 1240 320 'A' 150) d1 in
                 let (o3,d3) = update (record 1440 320 'A' 150) d2 in
                 let (o4,d4) = update (record 1640 320 'A' 150) d3 in
                 if[!] (isok o1) & (isok o2) & (isok o3) & (isok o4) then
                   (o4,d4)
                 else
                   (error "lots" 0,d)

and     lots2 d = let (o1,d1) = update (record 1040 320 'A' 150) d  in
                  let (o2,d2) = update (record 8840 320 'A' 150) d1 in
                  let (o3,d3) = update (record 1440 320 'A' 150) d2 in
                  let (o4,d4) = update (record 1640 320 'A' 150) d3 in
                  if[!] (isok o1) & (isok o2) & (isok o3) & (isok o4) then
                    (o4,d4)
                  else
                    (error "lots2" 0,d)

in      manager acct [lots;
                      lots2]
```

A.3.4 Long Read Transaction

```
--        This program defines a transaction that updates four entities
--        and another that looksup the same four entities. The read
--        transaction is followed by the write transaction in the
--        manager's input stream.

#include "esops.t";

let rec
        isok (ok m r) = true
||      isok msg = false

and     lots d = let (o1,d1) = update (record 1040 320 'A' 150) d  in
                 let (o2,d2) = update (record 1240 320 'A' 150) d1 in
                 let (o3,d3) = update (record 1440 320 'A' 150) d2 in
                 let (o4,d4) = update (record 1640 320 'A' 150) d3 in
                 if[!] (isok o1) & (isok o2) & (isok o3) & (isok o4) then
                   (o4,d4)
                 else
                   (error "lots" 0,d)

and     lots2 d = let (o1,d1) = lookup 1040 d  in
                  let (o2,d2) = lookup 1240 d1 in
                  let (o3,d3) = lookup 1440 d2 in
                  let (o4,d4) = lookup 1640 d3 in
                  if[!] (isok o1) & (isok o2) & (isok o3) & (isok o4) then
                    (o4,d4)
                  else
                    (error "lots2" 0,d)

in      manager acct [lots2;
                      lots]
```

A.3.5 Long Read Transaction and Deposits

```
--        This program defines a transaction that looksup four entities
--        and also a bank deposit transaction. The manager is invoked
--        with the lookup transaction followed by  two deposits to two
--        of the accounts being looked up.

#include "esops.t";

let rec
        isok (ok m r) = true
||      isok msg = false

and     lots2 d = let (o1,d1) = lookup 1040 d  in
                  let (o2,d2) = lookup 1240 d1 in
                  let (o3,d3) = lookup 1440 d2 in
                  let (o4,d4) = lookup 1640 d3 in
                  if[!] (isok o1) & (isok o2) & (isok o3) & (isok o4) then
                    (o4,d4)
                  else
                    (error "lots2" 0,d)

and     deprec (ok m (record ano bal class crl)) n =
          (record ano (bal+n) class crl)
||      deprec (error m k) n = (record 0 0 'A' 0)

and     dep a n d = let (o1,d1) = lookup a d in
                    let (o2,d2) = update (deprec o1 n) d in
                    if[!] (isok o1) & (isok o2) then
                      (o2,d2)
                    else
                      (error "dep" 0,d)

in      manager acct   [lots2;
                        dep 1240 5;
                        dep 1440 5]
```

Appendix B

ML File Manager

```
(*            AN IMPLEMENTATION OF DATABASE VIEWS            *)
(*            ***********************************            *)
(* The following is a small example of how a generic data    *)
(* manager that supports views can be constructed using      *)
(* SIGNATURE/STRUCTURE and FUNCTOR mechanisms provided by     *)
(* ML in a purely functional manner.                         *)

(*Auxiliary functions*)

fun fst (x,y) = x;

fun snd (x,y) = y;

fun filter p (x::xs) = if (p x) then
                         (x::filter p xs)
                       else
                         filter p xs
  | filter p [] = [];
```

```
signature BKR =
 sig
  type Kt
  type Rt0
  val NullRt :Rt0
  val eq :Kt * Kt -> bool
  val le :Kt * Kt -> bool
 end

structure acctkr:BKR =
 struct
  type Kt = int                        (*Acctno*)
  type Rt0 = real * string * real (*Balance,Account Class,Credit Limit*)
  val NullRt = (0.0,"",0.0)
  val eq :Kt * Kt -> bool = op =
  val le :Kt * Kt -> bool = op <=
end;

structure custkr:BKR =
 struct
  type Kt = string                          (*Name*)
  type Rt0 = int * string list * int (*Acctno,Address,Phone no*)
  val NullRt = (0,[],0)
  val eq :Kt * Kt -> bool = op =
  val le :Kt * Kt -> bool = op <=
 end;

signature PARAMDB =
 sig
  type Dbt
  structure bkr:BKR
  type Request
  val lookup :bkr.Kt -> Dbt -> (bkr.Rt0*Dbt)
  val update :(bkr.Kt * bkr.Rt0) -> Dbt -> (bkr.Rt0*Dbt)
  val insert :(bkr.Kt * bkr.Rt0) -> Dbt -> (bkr.Rt0*Dbt)
(*val delete : .... *)
  val flatten :Dbt -> (bkr.Kt * bkr.Rt0) list
```

```
  val manager :(Dbt*Request list) -> bkr.Rt0 list
  val initdb :Dbt
 end;

functor genericdb(bkr:BKR):PARAMDB=
 struct
  datatype Dbt =  tip of (bkr.Kt*bkr.Rt0)
                        | node of ( Dbt * bkr.Kt * Dbt)
                        | empty;
  type Request = Dbt -> (bkr.Rt0*Dbt)
  structure bkr = bkr
  fun lookup k' d =
    let
      fun lookup' (k', (tip(k,r))) = r
        | lookup' (k', (node (l,k,r))) = if bkr.le(k',k) then
                                           lookup' (k',l)
                                         else
                                           lookup'(k',r)
    in
      (lookup' (k',d),d) end;

fun insert (k',r') d =
  let
    fun insert' ((k',r'),tip(k,r)) = if bkr.le(k',k) then
                                       node (tip(k',r'),k',tip(k,r))
                                     else
                                       node (tip(k,r),k,tip(k',r'))
      | insert' ((k',r'),node(l,k,r)) = if bkr.le(k',k) then
                                          node (insert'((k',r'),l),k,r)
                                        else
                                          node (l,k,insert'((k',r'),r))
      | insert' ((k',r'),empty) = tip(k',r')
  in
    (bkr.NullRt,insert'((k',r'),d)) end;
```

```
  fun update (k',r') d =
    let
      fun update' ((k',r'),tip(k,r)) = if bkr.eq(k',k) then
                                         tip(k',r')
                                       else
                                         tip(k,r)
        | update' ((k',r'),node(l,k,r)) = if bkr.le(k',k) then
                                            node (update'((k',r'),l),k,r)
                                          else
                                            node (l,k,update'((k',r'),r))
    in
      (bkr.NullRt,update'((k',r'),d))
    end;

(*fun Delete k' d = .... *)

  fun flatten d =
(*Flatten returns the relation sorted in (descending) key order*)
      let fun flat (rel,tip(k,r)) = (k,r)::rel
            | flat (rel,node(l,k,r)) = flat ((flat (rel,l)),r)
      in
        flat ([],d)
      end;

  fun manager (d,f::fs) = let val (outp,d') = f(d) in
                            outp::manager(d',fs) end
    | manager (d,[]) = [];
  val initdb = empty
 end;

structure acct:PARAMDB = genericdb(acctkr); (*Instances of parameterised*)
                                            (*Database                  *)
structure cust:PARAMDB = genericdb(custkr);

signature ATMKR =                  (*Hide parts of the keys and*)
 sig                               (*Records from atm's view   *)
  structure bkr:BKR
```

```
  type AtmRt
  val atmattr :bkr.Rt0 -> AtmRt
  val atmupd :AtmRt * bkr.Rt0 -> bkr.Rt0
 end;

structure atmkr:ATMKR=
 struct
  structure bkr = acctkr
  type AtmRt = real * string
  fun atmattr (bal,ac,cl) = (bal,ac)
  fun atmupd ((bal',ac'),(bal,ac,cl)) = (bal',ac',cl)
 end;

signature ATMDB =
(* Allows us to hide most of a PARAMDB, also hide most of the*)
(* key and record info, except that needed by the view       *)
(* constructor functions                                     *)
sig
 type Dbt
 structure bkr:BKR
end;
```

```
signature ATMVIEW =
(*This signature allows lookup and update on an ATMDB, but*)
(*neither addition nor creation*)
 sig
  structure a :ATMDB                (*hiding of accts functionality*)
  structure akr :ATMKR
  val lookup :a.bkr.Kt -> a.Dbt -> (akr.AtmRt * a.Dbt)
  val update :(a.bkr.Kt * akr.AtmRt) -> a.Dbt -> (akr.bkr.Rt0*a.Dbt)
 end;

functor makeav(acct:PARAMDB, atmkr: ATMKR):ATMVIEW=
 struct
  structure a = acct
  structure akr = atmkr
  fun lookup k d = let val (r,d') = a.lookup k d
```

```
                        in (akr.atmattr r,d') end
  fun update (k,r) d = let val (r',d') = a.lookup k d
                           in a.update (k,akr.atmupd (r,r')) d
                       end
end;
                                        (*A view instance   *)
structure atmv:ATMVIEW = makeav(acct,atmkr);

                                        (*Build toy database*)
val adb = snd(acct.insert (12068,(30.0,"Dep",400.0)) acct.initdb);
val adb = snd(acct.insert (12032,(89.0,"Cur",250.0)) adb);
val adb = snd(acct.insert (12021,(342.0,"Sav",1250.0)) adb);
val adb = snd(acct.insert (12492,(430.0,"Cur",250.0)) adb);
val adb = snd(acct.insert (12472,(21.0,"Cur",250.0)) adb);

                                        (*Typical usage of views*)
atmv.lookup 12021 adb;
atmv.update (12068,(40.0,"Dep")) adb;


signature CREDCTRLKR =                  (*Hide different parts*)
 sig                                    (*for cred controller *)
  structure bkr :BKR
  type CredctrlRt
  val credctrlattr :bkr.Rt0 -> CredctrlRt
  val credctrlupd :CredctrlRt * bkr.Rt0 -> bkr.Rt0
  val credctrltuplev :bkr.Kt * CredctrlRt -> bool
 end;

structure cckr: CREDCTRLKR =
 struct
  structure bkr = acctkr
  type CredctrlRt = real * string * real * real
                                        (*4th field = safety margin*)
  fun credctrlattr (bal:real,ac,cl:real) = (bal,ac,cl,bal+cl)
  fun credctrlupd ((bal',ac',cl',sm'),(bal,ac,cl)) = (bal',ac',cl')
  fun credctrltuplev (k,(bal,ac,cl,sm)) = bal <= 50.0
```

```
 end;

signature CCDB =
(* Allows us to hide most of a PARAMDB-Also hide most of the key*)
(* and record info, except that needed by the  cred control view*)
(* constructor  functions.                                      *)
sig
  type Dbt
  structure bkr :BKR
end;

signature CCVIEW =
(* This signature allows lookup and update on a CCDB, but*)
(* neither addition nor creation                        *)
 sig
  structure a :CCDB          (*hiding of accts functionality*)
  structure cckr :CREDCTRLKR (*include CC's record type+functions*)
  val lookup :a.bkr.Kt -> a.Dbt -> (cckr.CredctrlRt * a.Dbt)
  val update :(a.bkr.Kt * cckr.CredctrlRt) -> a.Dbt -> (a.bkr.Rt0*a.Dbt)
  val flatten :a.Dbt -> (a.bkr.Kt * cckr.CredctrlRt) list
 end;
```

```
functor makeccv(acct:PARAMDB, cckr:CREDCTRLKR):CCVIEW=
(* Note that attrv2 actually constructs information from*)
(* that present in  the complete record.                *)
 struct
  structure a = acct
  structure cckr = cckr
  fun lookup k d = let val (r,d') = a.lookup k d
                     in (cckr.credctrlattr r,d') end
  fun update (k,r) d = let val (r',d') = a.lookup k d
                         in a.update (k,cckr.credctrlupd (r,r')) d
                       end
  fun flatattrv2 (k,r) = (k,cckr.credctrlattr r)
                                  (*Version of credctrlattr*)
                                  (*that covers keys*)
  fun flatten d = filter cckr.credctrltuplev (map flatattrv2 (a.flatten d))
```

```
end;

structure ccv:CCVIEW = makeccv(acct,cckr);

                                    (*usage of a credit*)
                                    (*controlers view  *)
ccv.lookup 12068 adb;

ccv.update (12032,(2.0,"Sav",2.0,3.0)) adb;

ccv.flatten adb;
```

Appendix C

Denotational Semantics

We present below a denotational semantics of the domain relational calculus. The semantics uses Stoy's notation [82] and assumes that the query to be described is both safe and generative. The syntactic categories for, and abstract syntax of the relational calculus are given in Subsection 8.5.2.

C.1 Semantic Domains

We use Set, Tuple and List constructors without definition.

$\text{Val} = \{\text{Unb}\}_{\uparrow} + \text{Num} + \text{String} + ...$

$\text{Dbase} = \text{RIde} \rightarrow \text{Set (Tuple Val)}$

$\text{Env} = \text{Ide} \rightarrow \text{Val}$

$\text{Simple} = \text{List Ide} \times \text{Val}$

C.2 Semantic Functions

$\mathcal{Q}$:Query → Dbase → Set Tuple

The $\mathcal{Q}$ function is given a database and a domain relational query and returns the set of tuples in the database that satisfy the query. It does so by determining all of the possible environments, or bindings of values in the database which satisfy the relational formula, $\rho \in \mathcal{E}[\![E]\!]\ \delta\ \{I_1 \mapsto Unb, .. I_n \mapsto Unb\}$. The value of the result tuple is then extracted, $(\rho[\![I_1]\!], .. \rho[\![I_n]\!])$. Note that initially each variable, I_j, is unbound.

$$\mathcal{Q}[\![\{(I_1, .. I_n)|E\}]\!]\ \delta = \{(\rho[\![I_1]\!], .. \rho[\![I_n]\!])|\ \rho \in \mathcal{E}[\![E]\!]\ \delta\ \{I_1 \mapsto Unb, .. I_n \mapsto Unb\}\}$$

$\mathcal{E}$:Exp → Dbase → Env → Set Env

The $\mathcal{E}$ function is given a relational formula, a database and the environment constructed so far. From these it constructs a set of environments which satisfy the formula.

$$\mathcal{E}[\![E_0 \wedge E_1]\!]\ \delta\ \rho = \{\rho_1|\rho_0 \in (\mathcal{E}[\![E_0]\!]\delta\rho) \wedge \rho_1 \in (\mathcal{E}[\![E_1]\!]\delta\rho_0)\}$$

$$\mathcal{E}[\![E_0 \vee E_1]\!]\ \delta\ \rho = \{\rho_0|\rho_0 \in (\mathcal{E}[\![E_0]\!]\delta\rho) \vee \rho_0 \in (\mathcal{E}[\![E_1]\!]\delta\rho)\}$$

$$\mathcal{E}[\![\neg E]\!]\ \delta\ \rho = \mathit{filter}\ (\mathcal{E}[\![E]\!]\delta\rho = \phi)\ \rho$$

$$\mathcal{E}[\![(I_1, .. I_n) \in R]\!]\ \delta\ \rho = \{\rho \oplus \{I_1 \mapsto v_1, .. I_n \mapsto v_n\}|(v_1, .. v_n) \in \delta[\![R]\!]\}$$

$$\mathcal{E}[\![A_0\ \omega\ A_1]\!]\ \delta\ \rho = \Theta\ [\![\omega]\!]\ (\mathcal{A}[\![A_0]\!]\rho)\ (\mathcal{A}[\![A_1]\!]\rho)\ \rho$$

$$\mathcal{E}[\![\exists(I_1, .. I_n) : R\ .\ E]\!]\ \delta\ \rho = \{\rho_1|(v_1, .. v_n) \in \delta[\![R]\!] \wedge \rho_1 \in (\mathcal{E}[\![E]\!]\ \delta\ \rho \oplus \{I_1 \mapsto v_1, .. I_n \mapsto v_n\})\}$$

$$\mathcal{E}[\![\forall(I_1, .. I_n) : R\ .\ E]\!]\ \delta\ \rho = \{\rho_1|(v_1, .. v_n) \in \delta[\![R]\!] \wedge \rho_1 \in \cap(\mathcal{E}[\![E]\!]\ \delta\ \rho \oplus \{I_1 \mapsto v_1, .. I_n \mapsto v_n\})\}$$

Θ :Op → Simple → Simple → Env → Set Env

The Θ function generates a set of environments that satisfy a relational calculus comparison. For the $<, \leq, \geq, >, \neq$ operators it simply filters out those environments that do not satisfy the comparison. For equality, however, it may be required to extend the environment. This occurs if one of the identifiers in the comparison is unbound, and is a form of unification.

$$\Theta[\![<]\!]\ (I_a, a)\ (I_b, b)\ \rho = \mathit{filter}\ (a < b)\ \rho$$

Similarly for $>, \leq, \geq, \neq$.

$$\begin{aligned}\Theta[\![=]\!]\ (I_a, a)\ (I_b, b)\ \rho = (&\mathit{unb}?\ a \rightarrow \{\rho \oplus \{\mathit{hd}\ I_a \mapsto b\}\};\\ &(\mathit{unb}?\ b \rightarrow \{\rho \oplus \{\mathit{hd}\ I_b \mapsto a\}\};\\ &\quad \mathit{filter}\ (a = b)\ \rho))\end{aligned}$$

$\mathcal{A}$:Atom → Env → Simple

The $\mathcal{A}$ takes a relational calculus constant or identifier and returns a value and a list possibly containing an identifier. If the atom is an identifier it appears in the result list and may be used to extend the environment in the Θ function.

$$\mathcal{A}[\![\ K\ ]\!] = ([], \mathcal{K}[\![\ K\ ]\!])$$

$$\mathcal{A}[\![\ I\ ]\!] = ([I], \rho[\![\ I\ ]\!])$$

Auxilliary Function

filter :Bool → Env → Set Env

$$\mathit{filter}\ p\ \rho = (p \rightarrow \{\rho\}; \phi)$$

Bibliography

[1] Abitboul S. Grumbach S. COL: A Logic-based Language for Complex Objects. Proceedings of the Workshop on Database Programming Languages, Roscoff, France (September 1987), 301-333.

[2] Albano A. Cardelli L. Orsini R. Galileo: A Strongly Typed Interactive Conceptual Language. ACM Transactions on Database Systems 10,2 (June 1985), 230-260.

[3] Flagship Project — Alvey Proposal. Document Reference G0003 Issue 4 (May 1985).

[4] American National Standards Institute Inc. *The Programming Language Ada Reference Manual.* Springer Verlag LNCS 155 (1983).

[5] Argo G. Fairbairn J. Hughes R.J.M. Launchbury E.J. Trinder P.W. Implementing Functional Databases. Proceedings of the Workshop on Database Programming Languages, Roscoff, France (September 1987), 87-103.

[6] Astrahan M.M. Blasgen M.W. Chamberlin D.D. Eswaran K.P. Gray J.N. Griffiths P.P. King W.F. Lorie R.A. McJones P.R. Mehl J.W. Putzolu G.R. Traiger I.L. Wade B.W. Watson V. System R: Relational Approach to Database Management. ACM Transactions on Database Systems 1,2 (June 1976), 97-137.

[7] Atkinson M.P. Programming Languages and Databases. Proceedings of the 4th International Conference on Very Large Databases (1978), 408-419.

[8] Atkinson M.P. PS-Algol Reference Manual 2nd Ed. University of Glasgow Computing Science PPR Report 12 (1985).

[9] Atkinson M.P. Buneman O.P. Types and Persistence in Database Programming Languages. ACM Computing Surveys 19,2 (June 1987), 105-190.

[10] Augustsson L. Johnsson T. Lazy ML User's Manual (1988).

[11] Backus J. Can Programming be Liberated from the von Neumann Style? Communications of the ACM 21,8 (August 1978), 613-641.

[12] Bancilhon F. Briggs T. Khosafian S. Valduriez P. FAD, A Powerful and Simple Database Language. Proceedings of the 13th International Conference on Very Large Databases, Brighton, England (September 1987), 97-107.

[13] Bayer R. McCreight E. Organisation and Maintenance of Large Ordered Indexes. Acta Informatica 1,3 (1972), 173-189.

[14] Birtwistle G.M. Dahl O.J. Myhraug B. Nygaard K. *Simula Begin.* Auerbach, Philadelphia (1973).

[15] Bird R.S. Hughes R.J.M. The KRC Users Guide. Oxford University Programming Research Group Manual (September 1984).

[16] Bird R.S. The Promotion and Accumulation Strategies in Transformational Programming. ACM Transactions on Programming Languages and Systems 6,4 (October 1984), 487-505.

[17] Bird R.S. Wadler P.L. *Introduction to Functional Programming.* Prentice Hall (1988).

[18] Blasgen M.W. Eswaran K.P. Storage and Access in Relational Databases. IBM System Journal 16,4 (1977).

[19] Breuer P.T. A Data Language — DL. Cambridge University Engineering Department CUED/F-INFENG/TR16 (June 1988).

[20] Breuer P.T. Applicative Query Languages. Cambridge University Engineering Department (1988).

[21] Buneman P. Nikhil R. Frankel R. An Implementation Technique for Database Query Languages. ACM Transactions on Database Systems 7,2 (June 1982), 164-187.

[22] Burton F.W. Sleep M.R. Executing Functional Programs on a Virtual Tree of Processors. Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture, Portsmouth, New Hampshire (1981).

[23] Ceri S. Gottlob G. Translating SQL into Relational Algebra: Optimisation, Semantics and Equivalence of SQL Queries. IEEE Transactions on Software Engineering (April 1985), 324-345.

[24] Clack C. Peyton Jones S.L. Generating Parallelism from Strictness Analysis. Proceedings of the Workshop on Implementation of Functional Languages. University of Göteborg and Chalmers University of Technology Report 17, (February 1985), 92-131.

[25] Clark K.L. Darlington J. Algorithm Classification through Synthesis. The Computer Journal 23,1 (1979), 61-65.

[26] Codd E.F. A Relational Model of Data for Large Shared Data Banks. Communications of the ACM 13,6 (June 1970), 377-387.

[27] Codd E.F. *Relational Completeness of Database Sublanguages.* Prentice Hall (1972).

[28] Cox S. Glaser H. Reeve M. Compiling Functional Languages for the Transputer. Proceedings of the Glasgow Workshop on Functional Programming, Fraserburgh, Scotland (August 1989).

[29] Darlington J. A System which Automatically Improves Programs. Acta Informatica 6,1 (January 1976), 41-60.

[30] Darlington J. A Synthesis of Several Sorting Programs. Acta Informatica 11,1 (January 1978), 1-30.

[31] Darlington J. Reeve M. ALICE: A Multiprocessor Reduction Machine for the Parallel Evaluation of Applicative Languages. Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture, Portsmouth, New Hampshire (1981).

[32] Date C.J. *An Introduction to Database Systems* 4th Ed. Addison Wesley (1976).

[33] Eswaran K.P. Gray J.N. Lorie R.A. Traiger I.L. The Notions of Consistency and Predicate Locks in a Database System. Communications of the ACM 19,11 (November 1976), 624-633.

[34] Farber D.J. Griswold R.E. Polonsky I.P. SNOBOL, a String Manipulating Language. Journal of the ACM 11,1 (1964), 21-30.

[35] Freytag J.C. Translating Relational Queries into Iterative Programs. Ph.D. Thesis, Harvard University (September 1985).

[36] Friedman D.P. Wise D.S. Aspects of Applicative Programming for File systems. Sigplan Notices 12,3 (March 1977), 41-55.

[37] Goldberg A. Robsen D. *Smalltalk 80 The Language and its implementation.* Addison Wesley (1983).

[38] Gonnet G.H. *Handbook of Algorithms and Data Structures.* Addison Wesley (1984).

[39] Gordon M.J.C. *The Denotational Description of Programming Languages.* Springer Verlag (1979).

[40] Gray J.N. A Transaction Model, in *Automata, Languages and Programming.* Springer Verlag LNCS 85 (July 1980).

[41] Gray J.N. The Transaction Concept - Virtues and Limitations. Proceedings of the 7th International Conference on Very Large Databases (October 1981).

[42] Gries D. *The Science of Programming* Springer Verlag (1981).

[43] Hall P.A.V. Optimisation of a Single Relational Expression in a Relational Database System. IBM Journal of Research and Development 20,3 (1976), 244-257.

[44] Hamilton A.G. *Logic for Mathematicians.* Cambridge University Press (1978).

[45] Harper R. Introduction to Standard ML. Edinburgh University Technical Report ECS-LFCS-86-14 (November 1986).

[46] Hecht M.S. Gabbe J.D. Shadowed Management of Free Disk Pages with a Linked List. ACM Transactions on Database Systems 8,4 (December 1983), 503-514.

[47] Held G.D. Stonebraker M.R. Wong, E. Ingress a Relational Database System. Proceedings of the 44th National Computing Conference (May 1975).

[48] Henderson P. *Functional Programming Application and Implementation.* Prentice Hall (1980).

[49] Henderson P. Purely Functional Operating Systems in *Functional Programming and its Application.* Darlington J. Henderson P. Turner D.A. (Eds) Cambridge University Press (1982).

[50] Henderson P. Jones G.A. Jones S.B. The Lispkit Manual. Oxford University Programming Research Group Technical Monograph PRG-31 (1983).

[51] Hudak P. Bloss A. The Aggregate Update Problem in Functional Programming Systems. 12th ACM Symposium on the Principles of Programming Languages (January 1985), 300-314.

[52] Hudak P. Wadler P.L. (Eds) Report on the Functional Programming Language Haskell. Draft (April 1989).

[53] Hughes R.J.M. The Design and Implementation of Programming Languages. D.Phil. Thesis, Oxford University Programming Research Group Technical Monograph PRG-40 (July 1983).

[54] Hughes R.J.M. Lazy Memo Functions, in *Proceedings of the 2nd International Conference on Functional Programming Languages and Computer Architecture* Nancy, France, Springer Verlag LNCS 201, (September 1985), 129-146.

[55] Hughes R.J.M. Why Functional Programming Matters. Report PMG-40 Programming Methodology Group, Chalmers University of Technology, Sweden (1984).

[56] Jarke M. Koch J. Query Optimisation in Database Systems. ACM Computer Surveys 16,2 (June 1984), 111-152.

[57] Jones S.B. Abstract Machine Support for Purely Functional Operating Systems, Oxford University Programming Research Group Technical Monograph PRG-34 (August 1983).

[58] Kowalik *Parallel MIMD Computation.* MIT Press (1985).

[59] Kung H.T. Robinson J.T. On Optimistic Methods for Concurrency Control. Carnegie Mellon Technical Report CMU-CS-79149 (October 1979).

[60] Lacroix M. Pirotte A. Domain Oriented Relational Languages. Proceedings of the 3rd International Conference on Very Large Databases (October 1977).

[61] Lampson B.W. Paul M. Siegert H.J. (Eds) *Distributed Systems, Architecture and Implementation.* Springer Verlag (1981).

[62] Liskov B. Scheifler R. Guardians and Actions: Linguistic Support for Robust, Distributed Programs. Proceedings of the 9th Annual Symposium on Principles of Programming Languages, Albuquerque, New Mexico (January 1982), 7-19.

[63] Mathews D.C.J. A Persistent Storage System for Poly and ML. University of Cambridge Technical Report 102 (January 1987).

[64] Michie D. 'Memo' functions and machine learning. Nature 218 (April 1968).

[65] Mueller E.T. Moore J.D. Popeck G.J. A Nested Transaction Mechanism for LOCUS. Proceedings of the 9th ACM Symposium on Operating System Principles, New Hampshire, USA (October 1983), 71-89.

[66] Nikhil R. Functional Databases, Functional Languages. Proceedings of the Appin Workshop, University of Glasgow Persistent Programming Research Report 16 (August 1985), 299-313.

[67] Nikhil R. Semantics of Update in a FDBPL. Proceedings of the Workshop on Database Programming Languages, Roscoff, France (September 1987), 365-383.

[68] O'Donnel J.T. An Architecture that Efficiently Updates Associative Aggregates in *Proceedings of the 2nd International Conference on Functional Programming Languages and Computer Architecture* Nancy, France, Springer Verlag LNCS 201 (September 1985), 164-189.

[69] Peyton Jones S.L. Using Futurebus in a Fifth Generation Computer. Microprocessors and Microsystems 10,2 (March 86), 69-76.

[70] Peyton Jones S.L. *The Implementation of Functional Programming Languages.* Prentice Hall (1987).

[71] Peyton Jones S.L. FLIC - a Functional Language Intermediate Code. Department of Computer Science, University College, London, Internal Note 2048 (February 1987).

[72] Poulovassilis A.P. FDL: An Integration of the Functional Data Model and the Functional Computational Model. Proceedings of the 6th British National Conference on Databases (BNCOD 6) (July 1988), 215-236.

[73] Quine W.V. *Word and Object* MIT Press (1960), 141ff.

[74] Reed D.P. Naming and Synchronisation in a Decentralised Computer System. Ph.D. Thesis, Massachusetts Institute of Technology MIT/LCS/TR-205 (September 1978).

[75] Robertson I.B. Hope$^+$ on Flagship. Proceedings of the 1989 Glasgow Workshop on Functional Programming, Fraserburgh, Scotland (August 1989).

[76] Sawyer T. Serlin O. DebitCredit Benchmark - Minimum Requirements and Compliance List. Codd & Date Consulting Group, San Jose.

[77] Schmidt D. Detecting Global Variables in Denotational Specifications. University of Edinburg Internal Report CSR-143-83 (September 1983).

[78] Sedgewick R. *Algorithms* 2nd Ed. Addison Wesley (1988).

[79] Sheard T. Stemple D. Automatic Verification of Database Transaction Safety. University of Massachusetts Computer and Information Science Technical Report 86-30 (1986).

[80] Shipman D.W. The Functional Data Model and the Data Language DAPLEX. ACM Transaction on Database Systems 6,1 (March 1981) 140-173.

[81] Smith J.M. Chang P.Y-T. Communications of the ACM 18,10 (October 1975), 568-579.

[82] Stoy J.E. *Denotational Semantics.* MIT Press (1977).

[83] Stoy W. The Implementation of Functional Languages using Custom Hardware. Cambridge University Ph.D. Thesis (December 1985).

[84] Trinder P.W. The Provision of Store in Functional Languages. Qualifying Dissertation, Oxford University Programming Research Group (February 1987).

[85] Trinder P.W. Wadler P.L. List Comprehensions and the Relational Calculus. Proceedings of the 1988 Glasgow Workshop on Functional Programming, Rothesay, Scotland (August 1988), 115-123.

[86] Trinder P.W. Referentially Transparent Database Languages. Proceedings of the 1989 Glasgow Workshop on Functional Programming, Fraserburgh, Scotland (August 1989).

[87] Trinder P.W. Wadler P.L. Improving List Comprehension Database Queries. Proceedings of TENCON'89, Bombay, India (November 1989), 186-192.

[88] Turner D.A. Recursion Equations as a Programming Language in *Functional Programming and its Application.* Darlington J. Henderson P. Turner D.A. (Eds) Cambridge University Press (1982).

[89] Turner D.A. Miranda System Manual, Research Software Limited (1987).

[90] Ullman J.D. *Fundamental Concepts of Programming Systems.* Addison Wesley (1976).

[91] Ullman J.D. *Principles of Database Systems*, Pitman (1980).

[92] von Bültzingsloewen G. Translating and Optimising SQL Queries Having Aggregates. Proceedings of the 13th International Conference on Very Large Databases, Brighton, England (September 1987), 235-245.

[93] Wadler P.L. An Introduction to Orwell. Oxford University Handbook (December 1985).

[94] Wadler P.L. A New Array Operation for Functional Languages. Oxford University Programming Research Group internal document (October 1986).

[95] Wadler P. L. List Comprehensions. Chapter 7 of Peyton Jones S.L. *The Implementation of Functional Programming Languages.* Prentice Hall (1987).

[96] Wadler P. L. Deforestation: Transforming Programs to Eliminate Trees. European Symposium on Programming, Nancy, France (January 1988).

[97] Wadsworth C.P. Semantics and Pragmatics of the Lambda Calculus. D.Phil. Thesis, Oxford University Programming Reasearch Group (1971).

[98] Whitehead A.N. Russell B. *Principia Mathematica* 2nd Ed. Vol I, Cambridge University Press (1925), 665ff.

[99] Wirth N. *Algorithms + Data Structures = Programs.* Prentice Hall (1976).

[100] Yao S.B. Optimisation of Query Evaluation Algorithms. ACM Transactions on Database Systems 4,2 (June 1979) 133-155.

[101] Yeh R.T. *Current Trends in Programming Methodology* Vol I. Prentice Hall (1971), 40-42.

[102] Zloof M.M. Query By Example. Proceedings of the 44th National Computing Conference (May 1975).