CACHING DERIVED DATA IN OBJECT-ORIENTED DATABASES, AND AN INTELLIGENT SYSTEM DESIGN FOR SELECTING THEIR MATERIALIZATION STRATEGIES

BY

WILLIAM ERIC VOSS

B.A., University of Tennessee, 1988
M.S., University of Illinois at Urbana-Champaign, 1994

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1998

Urbana, Illinois

Acknowledgments

I especially wish to thank my advisor Geneva Belford for her many many hours of help.

I would also like to thank, Marianne Winslett for serving on my Ph.D. committee, for being my masters degree advisor, and for sending me to Geneva when I was seeking a Ph.D. advisor. Sharad Mehrotra for serving on my committee, and for providing some especially helpful starting points before my preliminary exam. My friend David Hull, for his LaTeX "uiucthesis" style package, and for proofreading help on my preliminary statement. Vicki E. Jones for helpful discussions during the early stages of my Ph.D. work. Ian Chai for proofreading help on my preliminary statement. Michael Lake and Joel Jones for helpful discussions while I was searching for a topic. I would also like to thank my office mate Hewijin Jiau, for being a great office mate, and a nice friend.

Table of contents

List of tables

List of figures

Chapter 1

Introduction

1.1 My dissertation

> Once upon a time, in an MIS department not actually that long ago... One of the early database applications stored some numbers in a database, then later wanted to know the sum total of those stored numbers. That early application simply added up the total when it was needed. Later someone wrote a library routine which computed total, and this worked very well as long as total was not needed very often. However over time, since it was so simple to call the library routine, it became popular to use total more often. Eventually someone did a usage analysis which showed that a large percentage of the mainframe's time was spent computing total. A programmer was assigned to improve the library routine. The programmer wrote code which kept a running total instead, and changed the system routine to simply fetch the stored total, instead of fetching many records and recomputing the total. A new usage analysis showed almost no time spent computing total, and the programmer was promoted to the main office. Later another programmer was instructed to write code which stored more numbers in the database, unfortunately this programmer did not know about updating the total. The error was not caught until after *insert favorite catastrophe here,* and the programmer *insert favorite dire consequence here.*

In my little story, total was a **derived data item,** which could be computed from other information stored in the database. Initially the derived data item was recomputed from scratch when needed. If a derived data item is rarely needed while the "other information" changes frequently, recomputing is often the best choice. However when the derived data is frequently accessed, and the underlying data is rarely accessed, some form of incremental update or cache system will be more efficient. Unfortunately keeping a derived data item current with ad-hoc methods is difficult, and extremely error prone.

Supporting derived data is a fairly classic problem. For example by 1971 CODASYL [Oll78] supported recomputing from scratch using `''IS VIRTUAL RESULT ...,''` and supported maintaining a running-total-style current value using `''IS ACTUAL RESULT ....''` Later in [KP81], Shaya Koenig and Robert Paige further investigate supporting derived data, especially automatic incremental maintenance. They also report in [KP81] that the IS/1-PRTV system [Tod76] cached derived relations which it computed from scratch.

What has not been well addressed is having the database choose between recomputing from scratch, maintaining a current value, or selecting some other option. These choices are referred to as **materialization strategies.** My thesis will be developed in the context of an object-oriented database; it assumes that databases should:

1. explicitly support derived data,
2. support multiple materialization strategies, and
3. run as fast as is practical.

My thesis will try to show that if some additional information is specified during program development, then by also using runtime information about the current state and recent history of the database, an active database can do a "better job" of dynamically selecting a materialization strategy than a programmer could at compile time. The key idea is that the programmer cannot know the database's future operating conditions. Will collection X be large, or small? Will there be many queries about Y, or more updates of Z?

Obviously, these things could be specified individually in an ad-hoc manner, however doing so would be very expensive and error prone. Instead my thesis will show how a database company could provide a system or framework to support the use of derived values.

David Botzer and Opher Etzion [BE96] write:

> A substantial amount of work was dedicated to the creation of a technology that enables materialized derived data, but only

> modest effort has been invested towards deciding in which way a given derived data element should be materialized.

Therefore I will only incidentally work on developing materialization strategies. A focus of my work will instead be choosing materialization strategies.

Ashish Gupta and Inderpal Singh Mumick [GM95] write in their "Open Problems" section:

> Should alternative algorithms be tried, based on a cost based model to choose between the options?

That is one way of describing my work. My work explores having the database choose between alternative algorithms based on cost estimates. I assume the literal answer to their question is *"sometimes."* My work builds a framework which answers the question of *"when?"*

1.2 Background and assumptions

The remainder of this chapter introduces terminology which will be used throughout this document. It specifies many of my general assumptions, and thus some of my work's domain limits. It also explicitly organizes some relevant aspects of **derived data** and **materialization strategies,** and discusses some of the existing related work.

1.3 Underlying system assumptions

My work will always assume a database management system (**DBMS**) which enforces the classic atomicity, consistency, isolation, and durability (**ACID**) constraints. This is a rather well understood problem, and I have not attempted to expand upon it. For my work, an important aspect is that by enforcing ACID the database provides every transaction with the illusion that it has the database entirely to itself. The two exceptions are that a transaction's "clock" may be stopped intermittently and unpredictably, and that a transaction may be unpredictably aborted.

My work assumes a behaviorally object-oriented programming language resembling Smalltalk [GR83] or Java [GJS96]. I assume the language has reliable messages. I also assume a "strictly" encapsulated object-oriented language, where all instance variable references are via method invocation.

Unlike many current object-oriented database systems, I assume that the database has complete control over the application code. Sometimes I will also assume that the database system is not *always* busy (that I can use the idle loop). I will always assume that the time users spend waiting is a primary concern.

1.3.1 The simple case is the common case

I consider it essential to support the full flexibility of a behaviorally object-oriented system. However I also assume that most of the time, most applications will not fully use that flexibility. So while the system should let the application designer build a table which contains a little of everything: antiques, buildings, children, ... zebras, I expect most tables to contain simple immutable objects such as integers. This assumption means that I am willing to incur a large overhead for the complex case, provided the simple case is not prohibitive.

1.4 Derived data

For the purposes of this dissertation, **derived data** consists of any data which can be computed from other data *previously* stored in the database. Examples of derived data include totals and indexes. An underlying data item from which a derived data item is computed is referred to as a **deriver.** Thus a derived data item such as *total* might have an underlying deriver **collection.** To help ensure consistency, and to possibly improve efficiency, my work assumes that a database system should explicitly support derived data. Because my work assumes an object-oriented system, derived values will "know" that they are derived values.

The most trivial form of derived data is simply a reference to the same data item already stored elsewhere in the database. In its simplest form the computation is simply a pointer dereference. **SOURCE items** in CODASYL [Oll78] provide an early database example. Object-oriented **OIDs** [Cat93] are a common modern example. This trivial form has proved so important that we now rarely think of it as derived data. Supporting it provides one of the main motivations for the recent popularity of object-oriented databases. This trivial form is important because it is the simplest example of the desire to explicitly store data only once. By only storing data once, the data is guaranteed to remain consistent. Ensuring consistency is one of the primary motivations for explicitly supporting derived data.

Somewhat less trivial and perhaps more interesting are derived data items computed by more complex functions. Examples of this range from CODASYL **RESULT items** [Oll78] through most object-oriented methods [GR83]. A classic object-oriented example is a class Point which provides methods to report its coordinates under both Cartesian and polar coordinate systems. The programmer simply uses whichever coordinate system is currently most convenient, without regard to which system the Point class uses internally to store the information. The programmer also need not worry that the two representations might be out of synchronization. This thesis assumes the underlying database supports such examples of derived data. However my work is not very interested in such trivial examples.

My work is more concerned with "expensive" examples. For example if the underlying deriver or the derived data item is a large composite object, and thus "expensive" to access, my work might be appropriate. Another "expensive" example is any case where computing the derived data item is computationally difficult.

A common example of a large deriver is the **total** of a large collection. Trees, hash tables, and other forms of indexing are perhaps the most common example where both the deriver and the derived are often large composite objects. The traveling-salesman problem [CLR90] is a computationally expensive example.

In the relational database literature there has been considerable work on derived relations [Adi81], especially views [GM95]. Ashish Gupta and Inderpal Singh Mumick write in [GM95]:

> *What is a view?* A view is a derived relation defined in terms of base (stored) relations. A view thus defines a function from a set of base tables to a derived table; this function is typically recomputed every time the view is referenced.

With views, the derivers or **base relations** are usually large, the computations can be expensive, and the view itself is also frequently rather large, though often not stored. If one thinks of a relational database as a special case of an object-oriented database, then relational database views are related to my work as a special case. However, though some of my work could presumably be modified and applied in a relational database context, I do not explore views or relational databases explicitly in my dissertation. I focus on general cases in an object-oriented database context.

1.5 Strategies

If a database supports derived data, it must at some point compute the derived data. Once the data has been computed it is often useful to store, or at least cache, the result. Computing and (at least temporarily) storing are referred to as **materializing** the derived data. My thesis is concerned with selecting *how* the data is computed, *when* the data is computed, and *if, where,* and *for how long* the result is stored.

1.5.1 How is the derived data computed?

When computing derived data, the system may cache the result, or not cache the result.

1. One possibility is to simply recompute the value from scratch every time it is needed. This is sometimes referred to as **lazy** evaluation, as **fully virtual,** or as **passive mode.** *Point* normally uses this technique to provide a polar coordinate interface. However for large collections, recomputing the value (for example *total*) from scratch every time it is needed can be a very expensive operation.

2. The other option is to cache the result of computing the derived value. The result may have been computed using some form of on-demand or lazy evaluation. Alternatively, the result may have been computed using **eager** evaluation. This is also known as **immediate materialization,** or **active mode;** under such approaches, the result is computed before it is actually required. To use any caching technique, some way must exist to notice *all* changes made to the underlying deriver(s):

    (a) If the cached value is being computed from scratch when necessary, then a change to an underlying deriver means the cached value must be invalidated.

    (b) Alternatively, the cached value may be maintained incrementally, for example a running-total. If this approach is followed, then a function must exist to update the derived value based on changes to its derivers.

The incremental option actually has an advanced sub-option. The code supporting an incremental update must obviously be able to handle the smallest possible unit of change to the deriver. Sometimes it is also useful to explicitly support higher level complex changes directly. Supporting complex changes is of course optional since this can always be simulated by repeatedly calling the fine grained implementation instead. For example, removing all of the entries in a table might either be done by incrementally removing each individual entry one at a time, or as a single batch operation which entirely reset the derived value.

1.5.2 When is the derived data computed?

In [BE96] on page 261, Botzer and Etzion indicate that [Tec89] enumerates the *when* choices as:

1. The *immediate* mode which performs the derivation immediately upon triggering the derivation.
2. The *deferred* mode, in which the derivation is deferred to the end of the transaction.
3. The *decoupled* mode in which a different transaction has been created for the derived data.

The *immediate* mode described is actually somewhat ambiguous, because it is not clear in [BE96] whether *triggering the derivation* refers to accessing the derived value, or modifying a deriver. Accessing a derived value is obviously straightforward, a simple case of lazy evaluation. However, using deriver modification to immediately trigger an update of the corresponding derived value can have some atomicity problems which I describe in 3.1. The *deferred* and *decoupled* modes are necessarily triggered by deriver modification. The biggest advantage to both modes is that both permit multiple deriver modifications with only one derived value update. For example, increment every element of a collection, then update total, instead of updating total after each increment operation. The *decoupled* mode has the further advantage of permitting the system to move some of the work into the idle loop, instead of doing all processing while a user is waiting. All of these options are discussed further in chapter 3.

1.5.3 Where is it stored?

The other important issue is *if, when, where,* and *for how long* a derived value is stored. At one extreme the value does not have to be stored at all. The big advantage of this extreme is that then the database never needs to invalidate or update the derived value, so the database does not need to monitor the derivers for changes. However, if the database is willing to monitor the derivers, then it can consider storing results to avoid recalculating them.

Store in RAM (volatile storage)?

When possible, one would ideally expect a derived data item to be at least temporarily stored in RAM. The basic choices are to use RAM as a **cache** or as a **buffer.** If the cache technique is followed, then the derived value will probably be discarded, either at the end of the transaction, or when RAM becomes scarce. If the RAM is being used as a buffer, then the derived value will be retained at least until it is saved to non-volatile storage. If the value is accessed extremely frequently, it may even cost less in dollar terms to keep a copy in RAM indefinitely [Gra87]. Unfortunately, because of the need to match derived value objects with their derivers, temporarily caching a derived value in non-durable storage is almost never a good choice. This is discussed in section 4.3.

Store to disk (non-volatile storage)?

After the result is in volatile storage, it is possible to also save it to non-volatile storage. Unfortunately storing information on disk is expensive. Reading and writing the disk tends to be a database bottleneck. The people who buy and back up disk storage also tend to consider disk storage expensive in purely monetary terms.

While expensive, it is still sometimes very desirable to store information to disk. When the derived information is being maintained incrementally and the derivers are being discarded, storing to disk is mandatory. However usually derived data does not *have to* be maintained incrementally. My thesis focuses on having the database choose the materialization strategy for derived data elements. Thus any derived data element which the programmer knows **must** be maintained using **active mode** or **must** be maintained using **passive mode** is outside the scope of my dissertation. The programmer must provide code for both approaches so that my system will have choices available. Otherwise the programmer should instead use existing technology to implement the fixed choice.

I do not explore this optimization in my dissertation. However, when recomputing a derived value from scratch is possible, that data item does not absolutely have to be stored. Most derived values could always be recomputed from scratch if necessary, thus generally writing a derived data item to stable storage is not absolutely required. If storing an item is not absolutely required, then in principle it is possible to commit a transaction without waiting for the write to complete. A database could also take advantage of this by using two separate stable storage files, one for "normal" data which is treated conventionally, the other only for such recomputable derived values. Writes to the derived value file could all be performed asynchronously without delaying transaction commits; the system would also not need to make regular backups of the derived data file.

My dissertation develops a pragmatic engineering solution to handling the question of when to store to disk. Finding an optimal solution to this question is beyond the scope of my dissertation.

1.6 Monitoring derivers

If the database wishes to store the results of calculating derived values and to maintain consistency, it must monitor derivers. To support this monitoring, I assume that the database controls the application code. I also assume that the database language enforces "strict" object-oriented encapsulation. Thus the *only* way to access an instance variable is via an access method.

1.6.1 Code Inside

I assume that the database system controls the application code. The application developer submits source code to the database, and the database then compiles it internally. The database obviously has the option of altering the source code before compilation if that is desirable.

This does not necessarily mean that the database designer must also be a compiler writer. It is in principle perfectly possible for the database to call an external compiler. The application programmer submits code to the database; the database then calls the external compiler. For simplicity of exposition, I assume a more closely integrated compilation system in this thesis. I assume something resembling the Smalltalk-80 programming environment [Gol84]. The key point is that the database knows about and can if it wishes to alter all of the code.

I make this assumption because having control of the code makes monitoring updates to derivers straightforward. Under my "strict" encapsulation assumption, the *only* way to access an instance variable is via an access method. With control of the code, the database can add extra code to those access methods. Thus the database can notice when data is accessed, and when it is altered. (Of course most access methods will be inlined by the compiler, so there is a minimal performance hit.)

Other advantages of code inside

Having the database control the code provides benefits besides allowing simpler monitoring of derivers. Some of these benefits are not related to my work. Having control of the code means security is possible. (Especially assuming a "safe" language.) Having control of the code means the database can guarantee that all applications always use the most recent version of all code.

Some of the benefits are related to my work. Having control of the code means the database knows it may *recompile* as needed. For example the database could compile/recompile using/not-using an index, as it changes its mind about keeping an index around. Thus instead of:

```
If (index exists) { use index version }
else { make do without an index version };
```

the database can simply compile:

```
{ use index version };
```

or

```
{ make do without an index version };
```

This obviously lets the database move the cost of an "If (index exists)" branch to the idle loop. The larger savings here may be avoiding having to load both *use index* and *no index* versions of the code, though checking "if (index exists)" is also a potentially expensive operation.

With control of the code, the database can also insert instrumentation code into the programmer's code. Thus the database can collect data to assist its decisions about things such as indexing, materializations, and recompilations. Much of this work would take place in the idle loop, and thus be "free" in terms of user response time.

1.6.2 Other monitoring approaches

I am certain that by modifying the user's code, my proposed system can catch all changes. However modifying the user's code is not the only available approach. Díaz, Paton, and Gray report in [DPG91] that [MP90] reviews the following approaches for supporting rules (an equivalent problem):

- method-based mechanisms: the rule is precompiled into each place in the code where it might be activated. Alternatively, commands could be planted to fire the rule whenever applicable. [*This is the approach I initially reinvented.*]
- object-based mechanisms: enlarging the object description to indicate which rule to invoke whenever message sending takes place. [*This is the approach followed in [DPG91].*]

- external mechanism: additional structures are defined which support checking when some event occurs.

[DPG91] lists a number of criticisms of method-based mechanisms; however, most of the criticisms seem to only be serious problems when a programmer makes the code alterations by hand. Just as I would not propose that a programmer compile his programs manually, I also do not intend to have a human perform the monitoring modifications by hand. Thus I think most of the [DPG91] criticisms do not apply to my situation. Actually, [DPG91] would probably describe my work as a hybrid, as I both alter the code and enlarge some object descriptions. However, I recommend reading [DPG91] for more detail. They also describe their object-oriented rule management system which appears to scale up especially well.

1.7 Other work

My framework defines a form of **active database**. Currently the most popular active database approach is to support event-condition-action (ECA) rules. My work does not use ECA explicitly. I instead assume the minimum functionality absolutely required. Much, in some cases all, of that functionality could be provided by existing or proposed ECA systems. [Cha95] and [FT95] provide nice starting points for reading about ECA rule systems.

Some derived data can be computed using a number of different algorithms. For example [RSS96] investigates optimizing the incremental maintenance of SQL views; they develop a memoing solution which basically involves incrementally maintaining some additional views representing intermediate results. However, my thesis does not deal at all with such questions. Under my framework, the programmer is expected to explicitly specify what should be derived, and what algorithms should be used to perform the derivations.

In *An Adaptive Algorithm for Incremental Evaluation of Production Rules in Databases* [FRS93] Françoise Fabret, Mireille Regnier, and Eric Simon address a problem similar in some respects to the problem I address. In their work, they are balancing the information maintained by Rete discrimination networks [For82] against the information maintained by Treat discrimination networks [Mir87]. The Treat algorithm is faster when deletions are common, but the Rete algorithm is more efficient at handling insertions. They write:

> There is an inherent tradeoff in incremental algorithms. Main-

> taining information may help for computing new rule instantiations but may hurt performance when rule instantiations are deleted from the conflict set.

My work must balance a similar conflict concerning materializing derived values; caching a derived value may speed up later computations, but performance is hurt when the cache must be invalidated. Like my own work, their algorithm balances costs against benefits. However, we have very different underlying system assumptions, and thus very different actual costs and benefits. They describe their paper's contribution as *"an original adaptive algorithm that takes as input a program of rules and returns a discrimination network representing an expected good compromise."*

I did find one paper [QW91] by Xiaolei Qian and Gio Wiederhold which, based on the abstract, initially seemed to have done part of what I proposed:

> *Abstract*—Database updates are small and incremental compared to database contents. It is therefore desirable that recomputations of active relational expressions—such as views, derived data, integrity constraints, active queries, and monitors—can be performed incrementally as well. We present an efficient algorithm for the incremental recomputation of active relational expressions...

Fortunately for me, their approach is totally different, and inherently limited to relational databases. They are actually reporting on an algorithm which *"derives, by update propagation, the minimal incremental relational expressions that need recomputation."* Their system takes a relational expression of the database update, and then derives a relational expression which expresses the change to the database. Their algorithm unfortunately *"does not guarantee that the derived incremental expressions are always cheaper to evaluate than recomputation from scratch."* They do not suggest any solutions to that problem.

1.8 Choosing and changing materialization strategies

Very little work has been published regarding choosing or changing materialization strategies. R. G. Cattell writes on page 177 of [Cat93]:

> Unfortunately, few implementations of derived data have been done in ODMSs, and no work has yet been reported on caching and update.

Since then David Botzer and Opher Etzion [BE96] have addressed the problem of "tuning" a database application's materialization strategies. However their approach is to provide a batch tool to assist the database administrator in periodically adjusting the database. They also apply their work to only the simplest of derived data, and do not address the storage question at all. On page 265 they write:

> space considerations are marginal relative to performance issues and thus there is no case where a [Persistent Derived Information] value is calculated and the calculated value is not stored in the database.

David Botzer and Opher Etzion [BE96] also find very little other work in this area. They write:

> In the *active database* discipline there is no reported work of optimizing the *materialization strategy.*

They go on to describe other related research:

> These [*materialization strategy*] studies are concerned primarily with the implementation of these *materialization strategies* and not with the decision problem of choosing among them. An analytical view of this decision problem is presented in [SR88].
>
> ...
>
> In the area of *expert database systems,* Segev and Zhao [SZ91] focus on *data-driven rules.*

Neither [SR88] nor [SZ91] closely resembles my work. Jaideep Srivastava and Doron Rotem [SR88] assume a relational database system in which changes are immediately written to the base relations, and those changes which might alter the materialized view are written to a temporary file. They analyze how often the materialized view should be refreshed with the changes from the temporary file. Arie Segev and J. Leon Zhao [SZ91] are concerned with *expert database systems* (the integration of DBMS and *expert system* technologies). Their article focuses on dealing with rules which upon activation derive data which in turn affects other rules. I support derived data which is derived from other derived data, however this is certainly not central to my work.

In [GM95] Ashish Gupta and Inderpal Singh Mumick write:

> [RCK$^+$95] describes the ADMS system that implements and maintains simple materialized views.... and addresses questions of caching, buffering, access paths, etc..

The [RCK+95] article says:

> ADMS is trying to satisfy a need that has been neglected, that is, learning to perform better with experience obtained by query execution.

This is similar to my own goal. I believe that the database should take into account recent query history, and the current state, to improve its operation. However we differ in that the ADMS project only supports very simplified views, I support arbitrary functions.

1.9 Some rejected approaches

1.9.1 Behaviorally object-oriented

I assume a programming language similar to Smalltalk or Java. Unfortunately, I have found that semantic information is especially hard to come by in a behaviorally object-oriented system with dynamic type lookup, and, in the case of Smalltalk, very limited type information. For example, given more semantic information it might have been practical for my system to further automate the running-total example. Addition has some nice mathematical properties, which make summations of a collection especially easy to manipulate. However, under my assumptions the programmer can overload the definition of for example $+$, and is not forced to support the generally accepted meaning. Thus program analysis quickly becomes very impractical. As a programmer, I generally prefer to program in such a flexible system. However during this project I have found that as a system designer, it is very difficult to provide the application programmer with such flexibility. Had I restricted myself to supporting only structurally object-oriented systems (records), much of my work would have been simpler, but obviously less general, and thus presumably less useful.

1.9.2 Specifying derivers

One of my original design principles for this project was *that humans should provide information to solve the* **hard***er problems.* For example I originally intended to require the programmer to explicitly specify each derived value's derivers. Specifying derivers is definitely a **hard** problem. So by making this task a programmer responsibility, I could exclude it from the scope of my thesis. Unfortunately, it proved to be too difficult a task for the programmer!

While creating some small example problems by hand, I attempted to specify all of the derivers. However, when I later actually traced through the code involved, I often encountered derivers I had failed to specify. This was such a common occurrence, that I decided the programmer should not be given this critical task. Manually specifying derivers seems to have been so difficult, in part, because it violates abstraction and encapsulation. As an experienced programmer, when I call an existing function, or use an existing class, I automatically mostly ignore the internal implementation details. Automatically using abstractions this way is one of the harder things to teach a new programmer. However, unlearning this thought pattern also seems to be rather difficult. Unfortunately, those little implementation details are critical when attempting to manually specify **all** of the derivers. Thus, when I manually traced through example problems, and examined every line of code the computer would execute, I often found details I normally never consider. Although clearly a difficult problem, I concluded that specifying all of the derivers was apparently the sort of picky detail work computers are much better at than humans. Much of my work will involve developing algorithms which will permit my system to safely match derived value objects with their derivers.

1.9.3 The semantics dead end

After I decided that matching derived values with their derivers could not be a programmer responsibility, I attempted to create a system where the computer could statically analyze code, and thus determine the derivers. We know that in the general case, having a computer program derive the semantics of an arbitrary user-supplied function "`f()`" is as difficult as the halting problem. However it is in theory possible to restrict the user so that a computer program could fairly efficiently determine some useful semantic information about user-supplied functions. In the simplest terms, compilers do this all the time. A more complex example of related work is John M. Lucassen's PhD. thesis on effects [Luc87]. Unfortunately, if I restricted the allowable code sufficiently for the computer to analyze it, the allowable code was not very useful. However, if there were minimal restrictions on programmer coding freedom, a solution to my "find all derivers" problem would have implied a solution to the "halting" problem.

1.9.4 Instead of semantics

Eventually I accepted that a semantic analysis solution was beyond the scope of a PhD. dissertation. I stopped trying to give my system the responsibility of reasoning about the user-supplied code at “compile time.” Instead I developed runtime algorithms which although rather naive and brute-force, would work or fail fairly gracefully for any user code. My solution is described in chapter 2. The algorithms rest upon certain assumptions.

> **Assumption:** User-supplied “derived value” code will halt, and will correctly compute the derived value.

Obviously the application programmer must provide correct code!

> **Axiom:** The set of a derived value’s derivers includes only those “memory locations” accessed while computing the derived value from scratch.

> **Corollary:** If and only if a deriver is changed may a corresponding derived value change.

The derived value is only dependent upon the code used, and the values read to compute the derived value from scratch. If and only if some piece of that code or one of those values changes, will the system need to recompute the derived value.

> **Axiom:** Everything is an object.

It is rather awkward to repeatedly write “some piece of code, or one of the values read.” In some object-oriented languages, such as Smalltalk, everything in the system is also some form of object; that includes the code itself. Thus I shall sometimes use “object” to refer generically to anything which might be stored in a “memory location.”

1.10 Outline

The remainder of this dissertation is organized as follows. Chapter 2 describes the derived-deriver dependency links that are fundamental to detecting when a cached derived value object must have its cache invalidated.

Chapter 3 describes when a derived data value should be recomputed. This is partly an atomicity or safety problem, and partly an efficiency problem.

Chapter 4 is devoted to storage. It discusses what items are stored. It also considers where and for how long these items should be stored.

Chapter 5 is concerned with how my system will decide which derived values to cache, and with how my system will select between lazy evaluation and the various eager evaluation choices. Chapter 5 presents the cost formulas which form the basis of these choices, how the data necessary to compute the formulas may be collected, and ultimately an algorithm which uses the formulas.

Chapter 6 extends my work from derived values which are always recomputed from scratch to "delders," derived values which are incrementally updated such as running totals.

Finally chapter 7 summarizes my dissertation's contributions, draws some conclusions, and presents areas for further work.

Chapter 2

How the derived data is computed?

The simplest way to support derived data is to recompute each derived data element whenever it is needed (lazy, fully virtual, passive mode). Since the derivers are reaccessed every time, any changes to the derivers are automatically reflected in the derived data item's value. Thus lazy evaluation is the simplest option to implement. For my system, in any particular case, **lazy** is **always an option.** Sometimes a particular derived data object **must** always use lazy evaluation. A derived data object stores that fact as part of its internal state when it *"changes derived state into* "must be lazy" *mode."*

However in general my system also needs to support alternatives which retain (cache) some of the work between accesses (eager, immediate materialization, active mode). I am not exploring relaxed correctness systems, so if my system caches any of the work involved in computing a derived data element, it must also invalidate the cached work when the corresponding derivers are changed. To do this requires matching derived to deriver, and propagating cache invalidation messages appropriately. This is described below in section 2.1. Sometimes supporting eager evaluation can also lead to infinite loop problems, because eagerly evaluating one derived data item might trigger the eager evaluation of another derived data item, which triggers the eager evaluation of another, eventually forming a cycle, and thus an infinite loop. Section 2.2 below describes how my system detects this possibility, and avoids eager evaluation under those conditions.

2.1 Matching derived to deriver

To support caching alternatives, the system needs to match derived value objects (henceforth abbreviated **derived value**) with their corresponding derivers. Statically matching derived values with their derivers at "compile time" proved to be a very difficult problem, which in the general case appears to be as difficult as the halting problem. Informally:

```
Derived = if (f()) then deriver-A else deriver-B.
```

In this tiny example, either "`deriver-A`" or "`deriver-B`" is a deriver for "`Derived`," but which one depends upon the return value of "`f()`" for some arbitrary user supplied function "`f()`."

Although my system must in general support both the lazy and eager alternatives, in any specific case, it is acceptable for the system to support only lazy mode. Instead of solving the halting problem, I designed my system to dynamically match derived to derivers at runtime instead. My system also pessimistically avoids the caching alternatives when it becomes impractical to match the derivers with the derived. Basically, when it is hard to do, don't do it. This is actually a reasonable engineering choice, because in such cases the cost of supporting a cache oriented alternative includes the expense of matching derivers with the derived, and thus the cost will tend to be greater than the savings of avoiding the non-caching alternative.

As I just described, when my system supports a derived data element using one of the caching alternatives, it must match derived values with their derivers. For derived values which are recomputed from scratch (supporting incremental updates is described in chapter 6) the simplest version of my algorithm is discussed in the following sections.

2.1.1 Creating the derived-deriver links

When a derived value is computed from scratch using a caching alternative, first a unique time-stamp-version-number is created, and stored with the derived value. Then, during the computation of the derived value, the algorithm notices every value read (implicitly this includes the code which is run). Each object accessed is sent the message `youAreDeriverFor: derivedValueOOP at: timestampVersionNumber`. This message is implemented differently depending upon the nature of the object.

- **Immutable objects** simply ignore the message. My assumption (see 1.3.1) is that most objects will be immutable objects. *An object may be immutable because the programmer designed the type to be immutable, or because the type is a primitive type such as Integer, which conceptually has a different instance for each possible value, while each instance itself never changes value.*

- Items which are **already derivers,** add the pair `(derivedValueOOP, timestampVersionNumber)` to a table of all the derived values for which this object is a deriver. If `derivedValueOOP` is already in the table, `timestampVersionNumber` is simply replaced for that existing entry.

- Items which are **not already derivers** perform the necessary overhead to create a table for the derived values, and to in the future "notice" when they mutate, then they duplicate the behavior of **already derivers** described above.

- When other **derived values** are accessed, they perform similarly to **already derivers** as described above. However they also immediately send a message to `derivedValueOOP` which effectively creates a bidirectional link instead of the unidirectional link used by other derivers. If the accessed derived value (DV2) is not current, then this algorithm is recursively invoked to compute DV2. While DV2 is being computed, it will send `youAreDeriverFor: derivedValueOOP at: timestampVersionNumber` for each object accessed using its own OOP, and its own unique time-stamp-version-number.

At the end of computing the derived value from scratch, every "memory location" read will, if mutable, have a deriver-derived dependency registered, either directly, or indirectly via a chain of other derived values. The one exception is that if the algorithm just described accesses another derived value which for some reason must be stored using lazy evaluation instead of a "caching" alternative, then neither derived value may use a "caching" alternative. This is discussed further in section 2.2.3.

2.1.2 Using the derived-deriver links

When a deriver mutates, it immediately sends `IveChanged:` to each `derivedValueOOP` entry in its table of derived dependents. The message also includes the `timestampVersionNumber` it stored. Because we have reliable messages, after sending each `IveChanged:` message the deriver will never need to send `IveChanged:` with the same `timestampVersionNumber` again. Therefore after sending `IveChanged:` the deriver removes the corresponding row from its table. If this object is still a deriver when the derived value(s) are next recomputed, the lines will soon be added back. *Once the deriver has emptied its table of dependents, it may either mutate back into a* **not already a deriver** *object, or remain an* **already a deriver** *object* *with an empty set of derived value dependents. Which choice is better will depend upon the low level overhead associated with performing the mutation, versus the overhead of continuing to "notice" mutation. Both questions are dependent on very low level implementation details, and either might be the correct choice for any given implementation.*

When a derived value receives such an `IveChanged:` message it first compares the `timestampVersionNumber` to its own current time-stamp-version-number. If the two values do not match, the `IveChanged:` message is ignored. If the two values do match, then it resets its time-stamp-version-number to nil, and marks itself as needing update internally. It then informs a global "operations to be performed at commit object" (described in 5.6.2) that it will need to be updated, and includes in that message its list of derived value dependents (including this list facilitates an optimization described in 3.4). Lastly it sends its own `IveChanged:` messages to any derived value dependents following the same deriver protocol as above. Eventually this derived value will either be recomputed from scratch in accord with the "rules" as described in 3.1.7, or switched from the current caching, recompute from scratch strategy, to some other strategy.

Computations which both read and write a deriver

The previous algorithm neglects the possibility that a computation might both read and write a deriver. This section addresses that possibility. Suppose the computation of a derived value reads a deriver, thus creating a derived-deriver link (as described in 2.1.1), then later writes that deriver, thus according to the previously described algorithm triggering the sending of IveChanged: messages. Clearly any other derived value dependents do need to receive IveChanged: messages; so that part of the algorithm must remain unchanged, otherwise some changes might be missed. However, if this derived value processed the IveChanged: message normally, it would reschedule itself for recomputation, which would probably again generate a read followed by a write of this deriver, which would result in another IveChanged: message... (see Loop, Infinite).

Thus when an IveChanged: message is received by a derived value during the computation of that derived value, the IveChanged: message must be handled specially. Fortunately, although I assume the underlying database supports multiple simultaneous users, I also assume that by enforcing traditional ACID constraints it provides every transaction with the illusion that it has the database entirely to itself (see 1.3). My framework operates within the ACID constraints, and does not use preemptive multitasking; thus if an

IveChanged: message is received during the computation of a derived value, the system knows that it must have been generated by that computation (or a sub-computation). However that does not necessarily mean the IveChanged: message may simply be ignored.

How the system should handle a derived value computation which reads and later writes a deriver is a somewhat arbitrary design decision. What semantics does the designer want? Totally prohibiting this rather common behavior is obviously excessively restrictive, and thus not a very desirable choice. Clearly handling the IveChanged: message "normally" would tend to generate infinite loops, and thus is again not a desirable choice. One design option would be to simply totally ignore the message. That is equivalent to saying that writing a deriver removes any existing derived-deriver dependency. However this does not seem to be a very natural choice, because sometimes it is very difficult for the programmer to know when this happens. It is a somewhat arbitrary choice, but the semantics which seem the most natural to me is to "ignore" changes generated by this computation, but to maintain the derived-deriver dependency, so that whenever the deriver is next changed by some other computation, this derived value will be notified.

That semantics is fairly simple to implement. The semantics say an existing derived-deriver dependency must be maintained after the derived writes the deriver. However the previous algorithm clears the table of dependencies when IveChanged: is sent. The solution is simply to again add this derived value as a dependent of the deriver. So the revised algorithm is as follows:

> *When a derived value receives an* `IveChanged:` *message it first compares the* `timestampVersionNumber` *to its own current timestamp-version-number. If the two values do not match, the* `IveChanged:` *message is ignored. If the two values do match, then* it checks to see if this derived value is currently being recomputed. If it is in the process of being recomputed, then it sends `youAreDeriverFor: derivedValueOOP at: timestampVersionNumber` to the deriver, causing the deriver to again add the pair to its recently emptied table of dependents. If this derived value is not currently being recomputed, *it resets its timestamp-version-number to nil, and marks itself as needing update internally. It then informs a global "operations to be performed at commit object" that it will need to be updated, and includes in that message its list of derived value dependents (including*

> *this list facilitates an optimization described in 3.4). Lastly it sends its own* `IveChanged:` *messages to any derived value dependents following the same deriver protocol as described above in 2.1.2. Eventually this derived value will either be recomputed from scratch in accord with the "rules" as described in 3.1.7, or switched from the current caching, recompute from scratch strategy, to some other strategy.*

The above semantics and revised algorithm remain unchanged for any combination of one or more reads followed by a write and then any combination of zero or more reads and/or writes.

Suppose that instead the computation of a derived value first writes a deriver, and then reads the deriver. This creates a presumably unnecessary derived-deriver dependency, but is otherwise harmless. Thus the existing revised algorithm may safely be used. A possible further optimization is to cause such initial writes to block the establishment of subsequent derived-deriver dependency links. However, that would obviously involve additional overhead for variable accesses. Currently I think this case is rare, and thus such overhead is not justified. I would not be inclined to add this optimization to an actual implementation of my framework. However, I currently have no reliable way to determine if this case would in practice actually be rare. Thus I will now present a possible implementation:

> When the computation of a derived value is started, a unique time-stamp-version-number is created (as previously described), and an initially empty table to list all the objects written is also created. As objects are written, their object-oriented pointers (OOPs) are added to the table. The previously described algorithm is then modified by performing a table lookup in the objects-written table before sending the `youAreDeriverFor: derivedValueOOP at: timestampVersionNumber` message. If the object is listed in the objects-written table, the `youAreDeriverFor:` message is not sent. When a derived value is accessed, thus creating a subcomputation which will generate its own time-stamp-version-number under the previously described algorithm, now it will also create its own table of all objects written. The objects-written table may be discarded at the end of the computation, it does not need to be retained. The previously described algorithm is otherwise unchanged.

Even that update to the algorithm does not catch the truly pathological case of accessing another derived value for which all the derivers had just been written without being read. However, while a programmer could in theory write code which implemented a "derived value" by first writing every value it reads, simply using a constant value would seem a much more reasonable implementation. However, even for this pathological case, my system only misses the presumably extremely rare optimization of noticing the programmer could have used a constant instead. My system still produces the "correct" results, so I do not consider the missed optimization a serious problem.

2.1.3 What to store

Chapter 4 discusses what values must be stored to non-volatile storage.

2.2 Mutations generated while computing derived data

Unlike functional programmers, object-oriented programmers tend to program using (nicely encapsulated) side effects. Thus when a non-trivial derived data item is computed, it will often mutate some derivers. This creates a few difficulties which my system must handle.

2.2.1 Motivation restated

Each derived data item can be considered the root of a tree representing the dependencies between derived and derivers (See figure 2.1). The tree may be compressed to exactly one level deep (root and leaves); except where the derived data item is in part derived from another derived data item, in which case every level of derivation adds an additional level to the tree. Handling additional levels under this conception is described in 2.2.3. The key point is that every leaf of such a tree is a simple deriver, every interior node is a derived value, and there is a path from the root to every leaf.

It is also easily possible that two different derived data items might have derivers (leaves) in common (See figure 2.2). This is not a problem so long as both derived values only read the derivers. However, if they both alter a deriver they have in common, then eager evaluation becomes an infinite loop. For example consider the function $inc()$ which increments (mutates) its argument. Suppose we had two derived values A and B defined as $A = C + D + inc(E)$ and $B = D + inc(E) + F$. Having the deriver D in common is not a problem because it is only read. However, eagerly

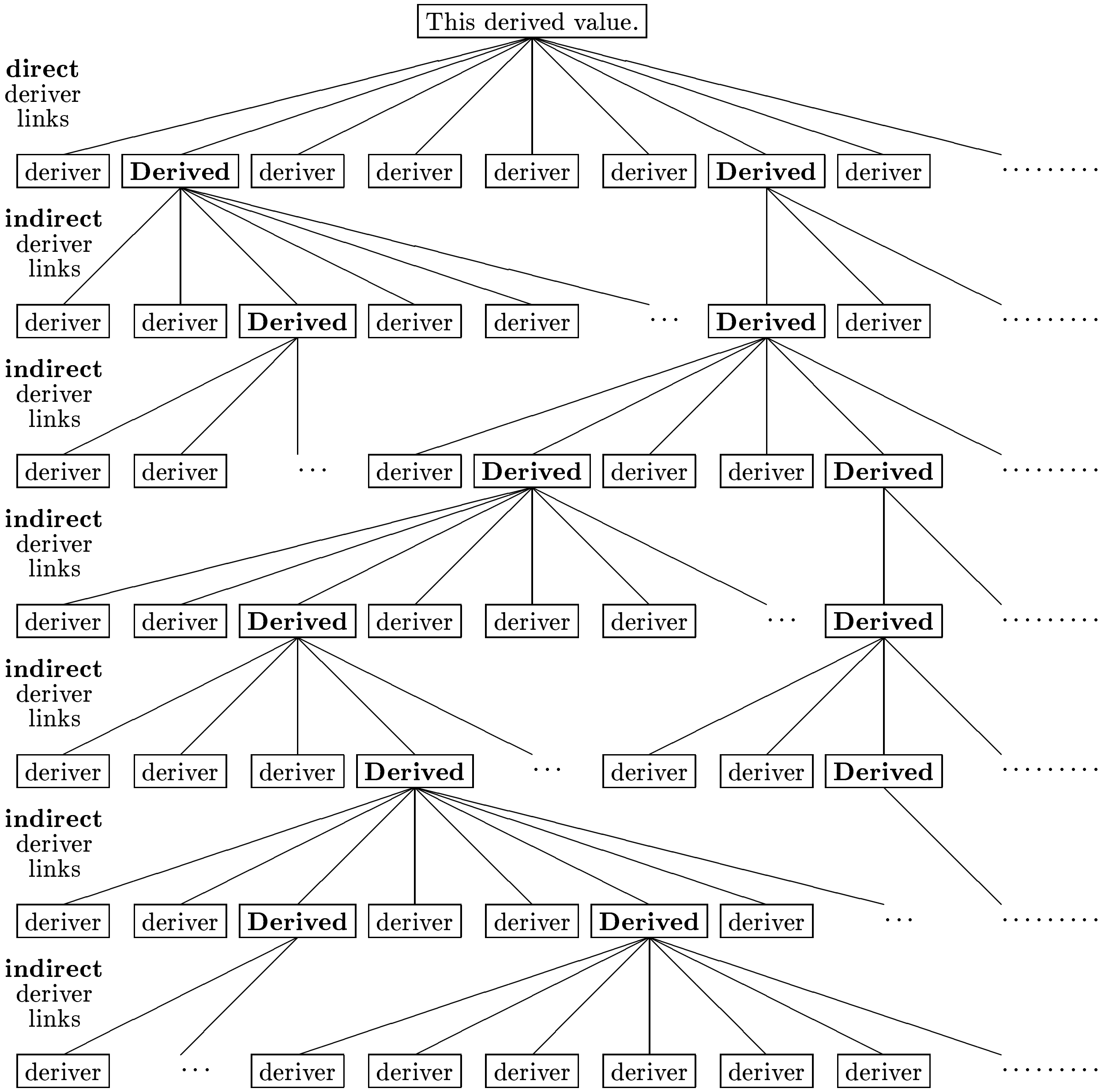

Figure 2.1: The leaves are simple derivers, the interior nodes are other derived values, and there is always a path from the root to any leaf.

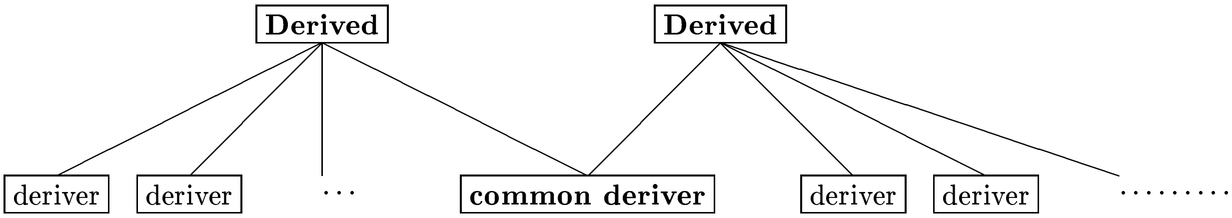

Figure 2.2: Common derivers are safe if merely read, a problem if modified.

computing A alters E which triggers an eager recomputation of B which alters E and thus triggers an eager recomputation of A...

One way to escape this problem is to never calculate such problem derived data values in an eager manner. If they are only calculated when the programmer explicitly references them, they will not cause an infinite loop inside of the automated system maintaining derived values. This effectively redefines the problem to one of detecting "such problem derived data values."

Strictly speaking, in my example above I could permit A or B but not both to eagerly mutate a leaf they have in common without causing a loop. However I do not currently see any clear criteria for preferring one derived value over another. I am also extremely reluctant to define the system in a non-deterministic manner, such as "the first access wins." Furthermore, I suspect that often when the programmer does define a derived value in such a way as to cause side affects, the programmer would prefer that the derived value not be randomly recalculated. All these considerations lead me to make an engineering choice, and currently **aggressively prohibit all eager evaluation of derived values which "read" shared derivers which are mutated by the calculation of a derived value.** Most of this section deals with a way of enforcing this prohibition automatically.

Advantages and disadvantages of this solution

The biggest advantage of this solution, is that it potentially permits the calculation of a derived value to create and arbitrarily mutate temporary or internal objects, provided they are never referenced directly by another derived value's computation. I consider this important for a useful system.

Unfortunately this solution involves considerable overhead. However, that appears to be an unavoidable corollary of allowing the programmer considerable freedom. For example, if the system insisted that all code which calculates derived values were otherwise read-only, this entire section would be unnecessary.

2.2.2 Definitions

Derived values may (among other things) be in one of two eligibility for evaluation states.

1. May be computed via lazy or eager evaluation.

2. Must be computed via lazy evaluation only upon access.

Every object in the system will at all times be in one of the following deriver states. (This deriver state information may be stored explicitly via an additional instance variable, or in some cases implicitly based on class membership.)

Clean An object which has not been accessed by a derived value computation, or that is immutable.

Private Deriver An object which has been

R read,

W written,

RW read and written,

by the computations associated with only one derived value.

Public Deriver An object which has been **read** by the computations associated with multiple derived values.

Volatile Deriver An object which has been **read and written** by computations associated with DIFFERENT derived values.

Objects in some of these deriver states will also store additional information.

Clean No additional information.

Private Deriver Which derived value.

Public Deriver Which derived values to contact if this object mutates.

Volatile Deriver No additional information.

2.2.3 Changing deriver state

As a derived value is computed from scratch, it will read and write various objects. Generally all objects accessed during the computation will change deriver state according to the rules below; exceptions are other derived values (described in 2.2.3 below), and "sanitized methods" (described in 2.3). Normally objects will change deriver state upon being accessed by a derived value computation as follows.

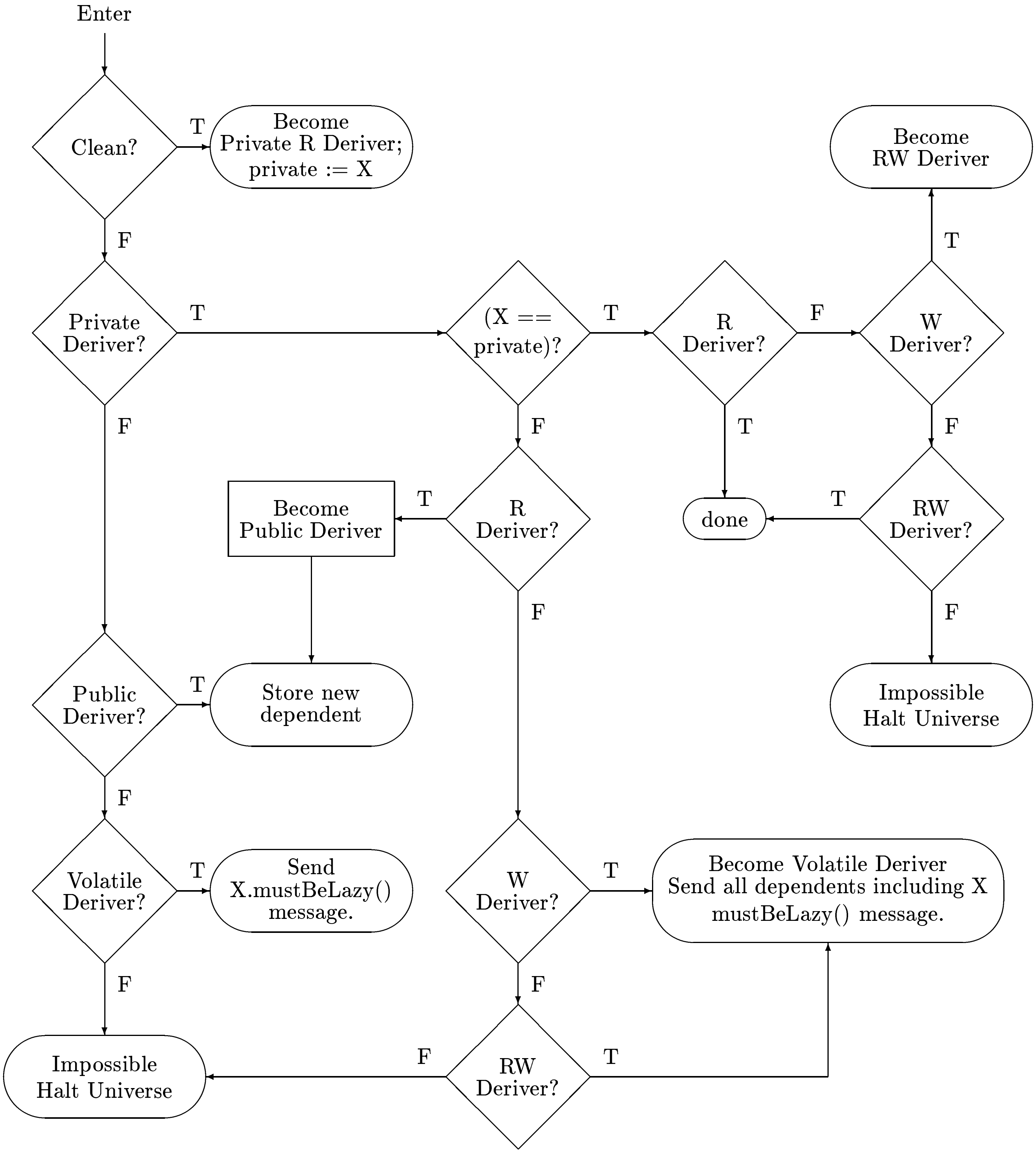

Figure 2.3: Upon a READ by X.

Upon a READ (see also figure 2.3)

Clean becomes Private R Deriver. (stores "private" derived value)

Private R Deriver

- If accessed by this object's "private" derived value computation, no deriver state change.
- otherwise becomes Public Deriver. (stores new dependent)

Private W Deriver

- If accessed by this object's "private" derived value computation, becomes Private RW Deriver.
- otherwise becomes Volatile Deriver. (sends all dependents, including this new one, messages forcing them to change derived state into "must be lazy" mode.)

Private RW Deriver

- If accessed by this object's "private" derived value computation, no deriver state change.
- otherwise becomes Volatile Deriver. (sends all dependents, including this new one, messages forcing them to change derived state into "must be lazy" mode.)

Public Deriver Remains Public Deriver. (stores new dependent)

Volatile Deriver Remains Volatile Deriver. (Also sends this derived value a message forcing it to change derived state into "must be lazy" mode.)

Upon a WRITE (see also figure 2.4)

Clean becomes Private W Deriver. (stores "private" derived value)

Private R Deriver

- If accessed by this object's "private" derived value computation, becomes Private RW Deriver.
- otherwise becomes Volatile Deriver. (sends all existing "read" dependents messages forcing them to change derived state into "must be lazy" mode. The derived value computation performing this "write" *might* also have an existing "read" dependency,

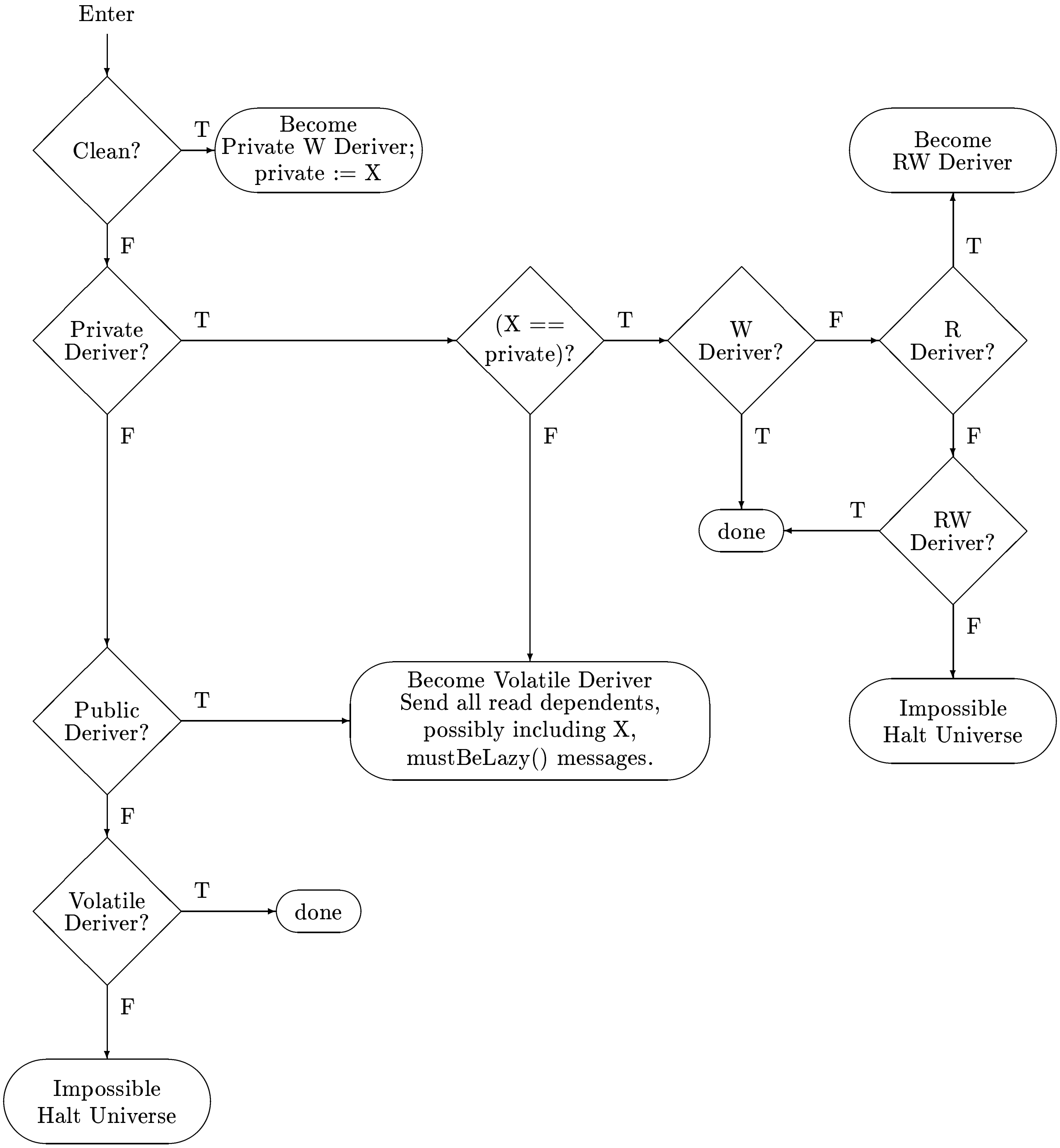

Figure 2.4: Upon a WRITE by X.

in which case it would also receive a "must be lazy" message which would even abort the computation if the computation was "eager.")

Private W Deriver

- If accessed by this object's "private" derived value computation, no deriver state change.
- otherwise becomes Volatile Deriver. (sends all existing dependents, not necessarily including this new one, messages forcing them to change derived state into "must be lazy" mode.)

Private RW Deriver

- If accessed by this object's "private" derived value computation, no deriver state change.
- otherwise becomes Volatile Deriver. (sends all existing dependents, messages forcing them to change derived state into "must be lazy" mode. Again this might not necessarily include the derived value computation performing this write operation, depending upon previous "read" operations.)

Public Deriver Becomes Volatile Deriver. (sends all existing dependents, not necessarily including this new one, messages forcing them to change derived state into "must be lazy" mode.)

Volatile Deriver Remains Volatile Deriver. (No Operation)

Other derived values as derivers

Unlike other derivers, derived values can in some sense never be written by another derived value's computation. Derived values are only written by their own derived value computation. Thus a derived value can never become a Private W Deriver or a Private RW Deriver. With those two possibilities eliminated, the Private R Deriver state represents unnecessary detail, and may be combined with Public Deriver. Thus derived values will only change between versions of the deriver states Clean, Public Deriver, and Volatile Deriver. Given two derived values A and B, regardless of B's deriver state, any computations required to compute B's current value will not generate direct dependencies for A. If a derived value A reads derived value B, then if B's deriver state, after having its value updated as needed, is:

Clean B changes to deriver state Public Deriver, and adds A as a dependent.

Public Deriver B simply adds A as another dependent.

Volatile Deriver B sends A a message forcing A to change derived state into "must be lazy" mode.

Derived of derived mode requirements

When a derived value depends upon another derived value, the two derived values must be maintained using compatible materialization strategies.

Derived	**Deriver**
Lazy	Lazy
Lazy	Caching
Caching	Caching

Table 2.1: The allowable derived-deriver mode combinations.

The prohibited combination is having a derived value use a caching strategy while one or more of the derived values it depends upon is using a lazy strategy.

	Derived	**Deriver**	
PROHIBITED	*Caching*	*Lazy*	**PROHIBITED**

Table 2.2: The one prohibited derived-deriver mode combination.

In figure 2.1 on page 25 I showed how a derived value's dependencies could be represented by a tree, with the interior nodes representing other derived values, and the leaves representing normal static values. When all of the derived values involved are using caching strategies, my system will create derived-deriver links as described in 2.1.1 between all of the nodes of such a tree which are subject to mutation. When any element of that tree is altered, IveChanged: messages will be propagated up the tree informing all dependents of the change. If an interior node of the tree is switched to a non-caching mode, there will be no derived-deriver links maintained for its derivers. Thus any other derived value dependent upon the non-caching derived value will not have a complete tree of derived-deriver links, and therefore it must not use a caching strategy. The permissible combinations are listed in table 2.1 and the prohibited combination is shown in table 2.2.

In previous sections I described the **Volatile Deriver** state which forces an object's dependents into "must be lazy" mode. The derived of derived mode requirement described in this section means that "must be lazy" mode must also be propagated up the dependency tree.

Because "lazy is always an option," it is possible that the system will select lazy mode for a derived value it could maintain using a caching strategy. If it does so, obviously it must also force the selection of lazy mode for any derived values further up the dependency tree. How my system selects among the available strategies is discussed further in chapter 5.

2.3 Sanitized methods

Some operations are conceptually (almost purely) writes, but implemented as a read and a write. For example incrementing a number requires reading its old value, but if the operation does not return the old or the new value, it is nearly a pure write operation. Another example is adding an object to an "infinite capacity" queue which is often implemented with something like

1. Read pointer containing `''last.''`
2. Write new value at `''last''` location.
3. Write `''last'' + 1` to pointer.

In both examples, the caller does not really care about the read, unless an exception is raised.

Instead of forcing such operations to somewhat artificially create additional "must be lazy" derived objects, I intend to permit the programmer to write special "Sanitized methods." A "Sanitized method" must:

- Handle all possible exceptions, or ABORT THE TRANSACTION. (THE POTENTIAL COST)
- Not return a value.
- Only examine the object identifier of any arguments, not send any messages to an argument, or use an argument as a reference.
- Only write its own instance and local variable storage.

If these rules are all followed, my system will treat a call to a sanitized method as a pure write on the receiver. The receiver will thus become a Private W Deriver or a Volatile Deriver. However without a read dependency, that will not force the derived value into "must be lazy" mode.

Sanitized methods provide a barrier so that the caller is not affected by any read operations, except for possibly having its transaction unexpectedly aborted. Thus no READ derived-deriver dependencies are created. The potential cost is that in the context of the transaction as a whole the programmer might have been able to handle some exceptions which within the restricted context of a "Sanitized method" must instead be handled by aborting the transaction. If the programmer considers this a serious problem, the programmer should of course not define many pieces of code as "sanitized."

2.4 Miscellaneous details

If the code assigns the "same" value to a deriver X as is already stored in X, then X will think it has mutated. This will occasionally result in some "unnecessary" computations. However the cost of always automatically checking for this rare case seems much greater than the cost of an occasional redundant computation. If the programmer considers this a problem, the programmer can obviously manually code a check for this possibility before any problematic assignment statements.

Chapter 3

When is the derived data computed?

In section 1.5.2 I described the idea that derived data may be computed using *immediate, deferred,* or *decoupled* mode. This chapter addresses issues involved with these three choices in more detail. Immediate mode computes the derived data "immediately," Deferred mode computes the derived data at the end of the transaction. Finally decoupled mode computes the derived data using a separate transaction.

These three choices must then be combined with the *lazy* and *eager* evaluation choices described in chapter 2. It is obviously impractical to combine lazy evaluation with the deferred, and decoupled modes. By its nature, when lazy evaluation is invoked, the system needs an answer "now," otherwise the system would continue to defer the evaluation, while both deferred and decoupled provide by nature delayed responses. However, lazy evaluation matches nicely with immediate mode, resulting in a combination that computes derived values on demand. That is basically the combination that a system which lacks something like my framework would probably support.

At first glance, eager evaluation could be matched up with immediate mode, resulting in a system which immediately recomputes derived values whenever a corresponding deriver is modified. For simple systems, where all derivers are simple atomic values, this combination is possible. However I assumed an object-oriented system, and object-oriented semantics generally imply that objects will only be accessed when they are in a consistent state. This leads to an atomicity problem, which I describe in 3.1 below.

My "solution" to the atomicity problem does permit matching eager evaluation with the deferred and decoupled modes. However there are still some other issues to consider. All but the most trivial derived data system will need to handle exceptions which might be raised during the computation of derived data. For combinations such as lazy with immediate, or eager with deferred, the simplest solution is simply to have the enclosing transaction handle the exception or abort itself. However, that solution is not available for the eager with decoupled combination; I describe my solution in section 3.2 below.

In sections 3.3 and 3.4 below I introduce some additional efficiency considerations regarding when invalidated derived data being maintained using a caching approach should be recomputed.

3.1 Atomicity and the semantics of consistent object state

My system needs to detect when deriver objects are modified, and update or invalidate the corresponding derived values. However, conventional object oriented semantics somewhat implicitly assume that access to an object will take place only:

1. When the object is in a consistent state.
2. When the accessor knows that the object is inconsistent because the accessor is causing the inconsistency.

The second point is the problem for my system. If a deriver is modified I must update the corresponding derived values. However, I do not wish to perform that update while the deriver object is in an inconsistent state.

A more concrete example may help to make this problem clearer. Consider an object-oriented rectangle which is internally represented by two points `lowerLeft` and `upperRight`. Further assume that it has the behaviors (methods) `size`, `moveLowerLeftTo: aPoint`, `moveUpperRightTo: aPoint` and `moveBy: aPoint` the last of which is implemented as:

```
moveBy: aPoint
        self moveLowerLeftTo: (lowerLeft + aPoint).
        self moveUpperRightTo: (upperRight + aPoint).
```

Using the `size` method is reasonable only when the rectangle object is in a consistent state. However, assuming that simply moving a rectangle should not change its size, `moveBy:` temporarily takes rectangle out of a consistent state.

In most programming systems, this temporary inconsistency is never an issue. However in my system, any derived data item which uses `size` as a deriver will need to be updated every time that `size` changes. For example suppose a list of rectangles sorted by size. The system should not need to resort that list **twice** every time that a rectangle is moved!

Unfortunately "consistent state" is a context dependent **semantic** concept. Clearly the programmer will need to somehow inform the system when objects are inconsistent. This is basically the classic atomic access problem. Here the conflict is that the user's explicit code, and the system's implicit derived data maintenance code, both require access to the deriver inside of a single transaction. There are a number of classic solutions to atomicity problems.

3.1.1 Semaphores and monitors

One solution is simply to have the programmer explicitly specify where an object is in an inconsistent state. For example Java's "synchronized" statement allows the programmer to specify the boundaries of monitors [GJS96]. Unfortunately writing multi-threaded code using semaphores or monitors is much more difficult than writing single-threaded code. By enforcing serializability, a good database system mostly provides programmers with the illusion that their applications have the database to themselves. Forcing programmers to instead write multi-thread safe code would clearly be an unacceptable solution.

3.1.2 Parallel processing oriented

Systems such as ABCL [BDH+90] have fairly heavy weight objects, and asynchronous messages. While such systems make parallel programming easier when significant parallelism is involved, they are definitely much more difficult to program than a normal single transaction. For minimally parallel problems, I consider such systems to be even harder for the programmer to code in than the classic semaphore and monitor solution.

3.1.3 Non-preemptive multitasking

Systems such as Smalltalk [GR83] do not generally need to worry about momentary inconsistency. They also do not generally impose large burdens on the programmer. This suggests that a non-preemptive solution might be best.

3.1.4 Defining this problem away

If my system is using "lazy evaluation" to compute derived data as needed, the consistency problems do not appear to exist. Technically, there is still the potential for some programmer to directly or indirectly request a rectangle's `size` in the middle of `moveBy:`. However it seems reasonable to characterize that as a programmer's error, not a system failure. Making this intuition somewhat more formal, I define the rule:

> **Rule:** *Accessing directly or indirectly a derived value for which a deriver is currently in an inconsistent state is undefined.*

This rule restores the conventional object-oriented semantic implication that access to an object takes place only:

1. When the object is in a consistent state.
2. When the accessor knows that the object is inconsistent because the accessor is causing the inconsistency.

3.1.5 Downside

The biggest downside is that there is no simple way of enforcing the "access is undefined" restriction. However, that is in a sense the corollary of why the restriction is needed. If the system could easily detect when the restriction is violated, the system could probably do something nicer instead. Another downside is that in complex circumstances it might be difficult for the programmer to know when there is a derivation relationship between two objects. However, presumably the programmer will at least always know when the code has made an object inconsistent.

3.1.6 Another rule

For completeness below, I will also explicitly define another rule, that:

> **Rule:** *All objects must be in a consistent state when the transaction attempts to commit.*

I had been implicitly assuming this rule all along, and it seems to be a very reasonable restriction requiring no more explicit justification.

3.1.7 Applying the rules

If my two rules are followed, then it will always be safe to recompute derived values "when needed" (lazy evaluation). It will also be safe to recompute derived values when the transaction is otherwise ready to commit. Thus assuming my two rules are followed, and that my system only recomputes derived values during those two times, the derived value computations will not access any deriver objects which are in an inconsistent state. Except when the derived value computation itself causes the momentary inconsistency.

3.2 Exception system

Any database system must somehow support runtime exceptions, for example division by zero. For simplicity, I assume that the underlying database uses an exception system similar to Java and Smalltalk. Under such an exception system, a runtime condition may "throw" an exception, which is propagated up the call tree until "caught" by a corresponding exception handler. Under a transaction oriented database system, if an exception is not "caught" the exception causes the current transaction to abort.

This behavior is generally perfectly adequate and acceptable for my system. When a derived value is calculated in a lazy manner, any exceptions caused by the derived value calculation are raised and handled immediately. When a derived value is being eagerly calculated inside of a transaction because it altered a deriver, any exception caused by the derived value calculation is again raised and handled immediately.

My system only needs to extend the underlying exception system to support decoupled mode, where a derived value is calculated in a system generated transaction instead of inside a user transaction. My solution is simple deferal. I protect the decoupled calculation of a derived value with an exception handler which catches all possible exceptions. This exception handler "handles" the exception by storing all of the associated exception information and by changing the derived value's state. From then on, any attempt by a user transaction to access the problem derived value will generate an exception, until the underlying exception is properly handled. Therefore, all user transactions which access the derived value will be aborted, until a transaction catches and handles the deferred exception.

Thus in all cases, my system raises exceptions associated with the computation of derived values as soon as possible. Such exceptions are either handled by the user code which "caused" the exception to be raised, or the user's transaction is aborted.

An alternate, though probably slightly less efficient, approach would be to simply abort the idle loop "transaction" which raised the exception, and change the offending derived value to “must be lazy” mode.

3.3 Inside a deriver modifying transaction

Computer programs tend to exhibit locality of reference. If a transaction accesses one data item, it is likely to access that data item again, and also likely to access related data items. Although I do not have any actual statistics, it seems reasonable to assume that a transaction which modifies one of a derived value's derivers, is more likely than the average transaction to read and/or modify some of that derived value's other derivers.

Disk access represents one of the largest costs for a database system. If a transaction has accessed a number of a derived value's derivers, the transaction has already partly paid the disk access cost associated with recalculating that derived value.

Thus it is potentially advantageous to recalculate derived values eagerly inside of deriver modifying transactions using deferred mode. Though it would require a greater alteration to the underlying database system, a specialized decoupled mode could provide an even more advanced option. It would work by first committing the user transaction without waiting for the recomputation of derived values. Then the system would immediately start a decoupled transaction using the committed transaction's address space to recompute the affected derived values. Thus many of the derivers would already be loaded, permitting the system to have its cake and eat it to. To add icing to the idea, the decoupled transaction could be skipped or quickly aborted by the scheduler if the database was currently heavily loaded so that such decoupled transactions would not significantly delay **user** transactions. Strategy selection is discussed later in chapter 5 in far more detail.

3.4 Ordering the recomputation of derived data

Provided that cycles (infinite loops) are avoided, there is nothing preventing the programmer from defining a derived data item in terms of other derived data items. Chapter 2 discussed how my system forces derived data items to only be calculated using lazy evaluation if there is a possibility that eager evaluation would result in an infinite loop.

Opher Etzion points out in [Etz93a], [Etz93b] and [Etz94] that if an event-driven update approach is followed, it is possible that a single modification of a base deriver may result in multiple updates to higher level derived data items. His proposed solution is based on building a dependency graph, and performing the updates in a dependency-oriented or data-driven fashion. He discusses this work further in [GE95] with Avigdor Gal. Table 3.1 shows one of his redundant update examples. In [Etz93a] Etzion writes "[potentially] *if X is modified then W is modified four times. Once as a direct result of the update of X, once by Y, and twice by Z.*"

```
Y := X + 3
Z := X + Y + 5
W := X + 2*Y + 3*Z
```

Table 3.1: *"Redundancy Problem"* example from page 9 of [Etz93a].

Under my system, if the X in table 3.1 is modified, all derived values which are dependent upon X and being maintained using a caching alternative will be notified of the modification via their derived-deriver links as described in chapter 2. Thus in this example the derived values Y, Z, and W would all be scheduled for update. However, my system is not vulnerable to the "redundant update" problem. Regardless of which derived value is evaluated first (Y, Z, or W) any other derived values referenced during its computation will if necessary be immediately computed and cached. Thus if the system were to attempt to compute W first, it would need to access X (the simple deriver), then compute the derived value Y, then compute the derived value Z. When it was done, the values Y and Z would again be current, and thus would not need to be recomputed, and therefore would not trigger a second recomputation of W.

Under my system, Etzion's dependency graph solution is not necessary to avoid redundant updates. However, it still might provide a useful optimization if system resources are scarce. The example above began by computing W, but upon attempting to access Y and later Z had to suspend the computation of W to compute these derived value dependents. When the derived of derived tree is only two layers deep, as it is in that example, the memory overhead of holding parts of two computations in memory is not significant. However, potentially an application might have a derived of derived tree N layers deep for a rather large value of N. Under that situation, the system memory resources required to recompute the derivers in order from the top of the tree down could be significantly greater than recomputing the derivers from the leaves of the tree up.

As I described in section 2.1.2, when a derived value mutates it informs a global object that it needs to be updated. As part of that message, it includes a list of any derived values it is in turn dependent upon. Using these messages, it is obviously straight forward to construct an ordered dependency graph of derived values. The derived value updates should then be performed from the set of derived values which have no dependencies upon other derived values which also need to be updated. If computing that set is computationally expensive, then the simple heuristic of first computing those derived values which are themselves derivers might also be a reasonable approach.

Chapter 4

If, where, and for how long stored?

This chapter focuses on storing data. As we have previously seen, eagerly materializing a derived value and caching the result forces my system to maintain a large amount of support data, not merely the derived value result itself. This chapter will address such storage issues directly and in more detail than the previous chapters which sometimes touched on the topic. The closely related question of whether to materialize a derived value is addressed later in chapter 5; this chapter will examine the storage required to make the informed materialization choices that chapter 5 describes. This chapter will also begin considering the costs associated with the storage required to support caching strategies, and thus why some of the materialization approaches we might desire are unfortunately impractical to implement. Chapter 5 will also further address these storage costs.

For simplicity of exposition, in this chapter I will assume that RAM or "memory" is an unstable storage medium which does not satisfy the ACID **durability** requirement. I will also assume that disk is a form of durable storage system. I will neglect the possible existence of battery backed RAM, and the possible swapping of virtual memory pages to disk. The relevant distinction is durability, and such details do not significantly affect my fundamental discussion.

Chapter 2 addressed *"How the derived data is computed"* and made it clear that using **lazy** evaluation, and thus not storing the derived values at all, was "always" an option under my assumptions. Obviously when **lazy** evaluation is selected, nothing needs to be stored. However, most of chapter 2 addressed what the system must do when it does decide to materialize derived data. Likewise, most of this chapter will deal with the ramifications of materializing derived data. Clearly the costs can be rather substantial.

4.1 What is there to store?

First let us review the most important data items which would not necessarily exist in a database without my derived data extension. This review will not generally cover temporary data items which are normally never written to disk.

4.1.1 Immutable derivers

Nothing extra needs to be stored to support immutable derivers. Thus obviously my system is most useful when most derivers are immutable. That most objects would be immutable was one of my underlying system assumptions in section 1.3.1.

4.1.2 Mutable derivers

Supporting mutable derivers is complex and expensive, as the previous chapters have already demonstrated. Section 2.1.1 explained how the system would create derived-deriver dependency links. Thus derivers will need another instance variable to store collections of `(derivedValueOOP, timestampVersionNumber)` pairs to represent these links. Section 2.2.2 expanded upon this with the various deriver states *Clean, Private Deriver, Public Deriver,* and *Volatile Deriver.* Generally, keeping track of deriver state will require the addition of another instance variable to each mutable object. Some implementations might instead store the deriver state information implicitly based on class membership. However, as mutating an object to a different class is relatively expensive in most object-oriented systems, and the state information may tend to be fairly volatile, that is unlikely to be a good solution for most implementations. Because the collection of `(derivedValueOOP, timestampVersionNumber)` pairs must be checked to see if it contains any entries after almost every mutation (see 2.1.2 and 5.4.6), depending upon the expense of that check, it may even be reasonable to add another boolean instance variable to mutable objects indicating if the collection is empty. Finally, if the system permits the code associated with an object to be altered, there must also be additional storage of `(derivedValueOOP, timestampVersionNumber)` pairs, presumably at the "class" level, to represent the dependency upon the code. Obviously, if the code used to compute a derived value is changed, the derived value should be recomputed. Otherwise an old cached value which resulted from a computation using the old code might be read instead.

4.1.3 Intelligent system variables

My system attempts to intelligently determine which derived values should be maintained using caching and non-caching strategies. This topic is primarily addressed in chapter 5. Here it is merely important to know that making an intelligent choice requires keeping track of some basis upon which to make the choice. Thus for every derived value the system will need to maintain values for the following variables, which are defined in chapter 5:

- smoothed(previousAttempts)
- smoothed(previousInsideAttempts)
- smoothed(readsBeforeMutation)
- smoothed(readsBeforeMutationOrCommit)
- smoothed(probabilityOfIdleTime)
- smoothed(monitorCost)

The smoothing of values is described in section 5.2.1, here it is only important to know that updating a smoothed value actually requires three values: an *actual value,* a *previous estimate,* and a *smoothing constant.* Although having custom *smoothing constants* has some advantages, to minimize storage overhead it would be possible to have only one *smoothing constant* per derived value, or even to use a system wide constant. The *actual value* is only used during the update computation itself, so in principle only the smoothed value itself (*previous estimate*) must be retained indefinitely.

Although they are tiny, and normally only their sum is saved at commit time, a derived object may also eventually receive a *TimeCostObject*, as described in section 5.4.8, from every mutable object it has ever caused to create a derived-deriver dependency link. I mention *TimeCostObjects* here because they are sent whenever a derived-deriver dependency link is traversed during a user transaction, thus there may be a large number of these tiny objects.

4.1.4 Derived values

In addition to the intelligent system variables described in 4.1.3 each derived value will need to store some other values. Derived values may themselves be derivers. When they are derivers, they will need to store the same collection of `(derivedValueOOP, timestampVersionNumber)` pairs representing dependencies as described in section 4.1.2 above. Though their possible states are more restricted than normal derivers, derived values will also need an instance variable to distinguish between the deriver states *Clean, Public Deriver,* and *Volatile Deriver.* Additionally a derived value must keep track of whether it has been forced into "must be lazy" mode. However, the "must be lazy" mode state change may be sufficiently rare and permanent to justify mutating the object into an instance of another class.

Eagerly maintained derived values

My system obviously expects to maintain some of the derived values using a caching strategy. Such derived values will, as previously described in section 2.1.1, need to store their current `timestampVersionNumber`, and a collection of `derivedValueOOP` links to their derived value derivers.

Derived values which, using the approach described in section 3.2, generate exceptions while being computed during the idle loop will need to store the exception information until the exception is handled.

Obviously, eagerly maintained derived values will also need to store their cached value!

4.1.5 Eager update scheduling

Though relatively minor in comparison to the other storage issues just described, some additional storage is also needed by the support objects which schedule updates at commit and during the idle loop.

The commit object introduced in section 2.1.2 and more fully discussed in section 5.6.2 will, during a transaction, need to keep a collection of those derived values which have had their cached value invalidated during the transaction. At commit it will process those derived values, possibly scheduling some of them for recomputation during the idle loop.

There will also need to be an idle-loop scheduler object (described further in section 5.7), which will keep a queue of the derived values needing to be eagerly computed as time becomes available.

4.2 Where is it stored?

As I described in section 1.5.3, derived values may not be stored at all, may be stored in RAM, or may be stored to disk. Because my system is only concerned with derived values which it may always recompute from scratch, it is never mandatory to store a derived value.

However, my system does find it highly desirable to store quite a large amount of information. For example, to intelligently choose among the possible materialization strategies, it is necessary to store the "intelligent system variables" described in 4.1.3 for each derived value. Discarding those variables is equivalent to starting the intelligent choice subsystem over from scratch. With no history data upon which to base its choices, the intelligent system would again need to explore the available alternatives, and would otherwise be basically guessing blindly.

When using the "intelligent system variables" causes my system to select a caching strategy for some derived value, that derived value must obviously store the variables described in 4.1.4. The largest storage problem though is that all of the derived value's mutable derivers **must** store either the variables described in section 4.1.2, or, if the mutable deriver is also a derived value, the variables described in 4.1.4. If a deriver has not done this, then any corresponding derived value must have its cache invalidated. Unfortunately, it is obviously rather difficult to determine what derived value corresponds with a lost dependency link! So once my system decides to use a caching strategy for some derived value, my system treats such storage requirements as absolutes, even though in theory storing the derived value and its associated dependencies is optional, since the system could recover from their loss. In practice, the choice of lazy or eager evaluation forces either not saving much support data at all, or saving all of the support data to disk.

This is unfortunately a typical problem. In theory, cached derived values may be maintained anywhere on the spectrum from not stored, through stored in RAM for a period of time, through treated as data subject to full ACID constraints. Obviously the middle-ground choices are theoretically appealing. For example, caching a derived value in RAM but not committing it to disk. However, such middle-ground choices tend to incur almost the same overhead as the more extreme choices, but with fewer benefiting transactions over which to amortize their costs. It can also be very difficult to take full advantage of the associated potential optimizations. Thus, for example, as the next section will explore in some detail, while in theory caching derived values in RAM is very attractive, in practice it is not a good choice.

4.3 Is it cached in RAM?

By the nature of computer systems, if a derived data item is materialized at all, it will for at least a while be cached in RAM. The real question is will it only be cached in RAM? In theory this alternative is very attractive. In practice I don't expect my system to ever use it. However, why it is attractive, and why it is impractical, is rather interesting.

If a cached value is discarded at commit, then the cached value will probably only be stored in RAM, and never saved to disk. The larger benefit of this is that then the dependency information also need not be saved to disk. This is perhaps not a large immediate virtue for the derivers written during the transaction, since the incremental cost of storing a few additional bytes is negligible using current storage technology. However, for the derivers only read, and not written, this approach might save a write operation. The write could be saved, because while the deriver itself might not have been written, creating the deriver-derived dependency link does involve a write operation. The deriver-derived dependency algorithm also potentially stores many obsolete `(derivedValueOOP, timestampVersionNumber)` pairs in tables representing the derived-deriver dependencies. Fortunately, from a storage standpoint, such tables are cleaned whenever the corresponding object mutates[1]. Unfortunately, from a time standpoint, such cleaning adds additional overhead to those transactions which cause such mutations. Obviously avoiding such costs altogether by simply not storing the links to disk is very attractive.

Unfortunately, this attractive alternative will remain impractical until a way to efficiently "temporarily" write the deriver-derived dependency link tables is more fully developed. Supporting deriver-derived links in an object-oriented system is probably most naturally done by adding an additional instance variable containing the collection of links to the deriver objects. All the alternative implementations appear to require a table lookup.

Unfortunately, adding a deriver-derived link collection (DDLC) variable means that when the system reads a deriver, and also needs to create a link, it must write the DDLC. That has some unfortunate implications. It implies that for a derived value to READ a deriver requires a WRITE lock on at least the DDLC, possibly the whole object. Because any mutation of the deriver object must check to see if the DDLC is empty (see 2.1.2 and 5.4.6) a WRITE lock on the DDLC effectively blocks anyone else from

[1] Obsolete derived-deriver dependency links may also be cleaned using an "off-line" procedure in the idle loop.

writing the whole object. Although, if the lock granularity is small enough, other transactions might be able to simultaneously read the deriver object provided they were not also computing a derived value at the time. Writing the DDLC also implies that we must save the DDLC to disk at commit just because the system created (and then removed) a dependency link in the DDLC. Under some object-oriented implementations, writing the DDLC might also require writing the deriver object itself, and thus the deriver itself might also need to be saved to disk at commit.

However, the alternatives which do not use a DDLC variable all seem to require a very annoying (expensive) table lookup. This would be a significant performance issue. Table lookups get expensive fast. Even though using separate tables instead of DDLC variables would make it possible to not write the derivers when they were read, separate tables would tend to just move the write contention problem from the DDLC to the table, while adding a table lookup.

What is really desired is a way to write the deriver during the transaction, then erase that write as a final stage prior to commit; so the existing commit process perceives the deriver to have merely been read, not written. We want and need a read lock, but not a write lock, and we do not want the cost of writing the deriver back to disk either. Unfortunately, I know of no existing system which supports such an option. It appears that implementing support for such behavior will be extremely dependent upon how the database system handles locking, and how it determines which objects are "dirty." As such, it is outside of the scope of this dissertation. I do present some preliminary ideas on how it might be done in section 7.3.2 under future work.

Until an efficient means exists to "temporarily" write a deriver object, creating derived-deriver links will "dirty" the corresponding DDLCs. This will require writing the DDLCs to disk at commit.

4.4 If it is saved to disk

If the system is going to write the DDLCs to disk anyway, as is currently required for any caching strategy, then the system has already done most of the work required to save the derived value to disk. Therefore normally if the system is going to cache a derived value at all, it should go ahead and also save the derived value to disk. By doing so it may potentially amortize the overhead involved over many transactions instead of just the current transaction. These relative cost calculations are discussed in chapter 5 more formally and in greater detail.

4.5 Storage overhead

As we have seen, when a derived value is materialized (its value is cached), it is not sufficient to merely cache the result of computing the derived data item. All of the associated deriver-derived dependency links and support data must also be retained to correctly handle the possibility that the "cache" might need to be invalidated, to prevent infinite loop problems, and to collect data for determining how to materialize the data item in the future. When most of the derivers are immutable objects this overhead is relatively trivial. However, when many of the derivers are mutable, the overhead can be very significant. This section examines the overhead required for the derived-deriver dependency links.

First there is the overhead of creating the dependency links themselves. This obviously slows the computation of the derived value. Unless the computation is performed during the idle loop, this overhead obviously conflicts with my system's goal of minimizing user wait times. If the computer's CPU utilization is near saturation, consuming more cycles may also create a bottleneck adversely affecting transaction throughput.

Storing the link information also consumes RAM. Though less critical during the idle loop, usually RAM is a scarce resource. In general, using more RAM means that other data must be swapped out, which again may adversely affect transaction throughput.

Though less critical inside of the idle loop, if the link information is stored to disk, potentially scarce disk bandwidth is consumed. As long as the system wishes to retain a materialized derived value, the system must also retain the dependency link information. Thus under many possible implementations, if the system wishes to retain a materialized derived value across transaction boundaries, it will probably also be desirable to save the value to stable storage. Regardless of when the information is stored to disk, doing so always consumes fiscally expensive disk storage. As I will briefly describe under future work in section 7.3.6, it might be possible to allocate some of that storage to otherwise unused scratch space. However, the algorithms described in this document do not consider such an optimization. Some of the disk consumption will consist of esthetically annoying old useless links, which must either be intentionally scavenged by a garbage collection process, or scavenged as a side-affect when they are encountered by user transactions. Obviously, in general my system prefers to perform this garbage collection during the idle loop when possible, instead of delaying user transactions. However, except on the most lightly loaded of systems, that will not always be possible.

Simply creating and storing the dependency links may potentially cause CPU, RAM, and/or disk bandwidth bottlenecks. Using the links also incurs similar costs. When a deriver is mutated, it must use its dependency link(s) to inform its derived data dependent(s) that it has changed. While the first such notification a derived data item receives is obviously critical, later notifications could simply be ignored without affecting the user's results. However, to know that such a message may be ignored, at least part of the derived data item must be loaded into RAM, again consuming CPU, RAM, and disk bandwidth resources. The algorithm described in chapter 2 permits ignoring obsolete `IveChanged:` messages. However, the extension described in section 5.4.8 measures the cost of all `IveChanged:` message sends, and thus may not entirely ignore obsolete `IveChanged:` messages. Therefore, mutating a deriver will require at least partially loading any derived value dependents.

Using materialized derived data may also cause conflicts between transactions. My system assumptions include the assumption that the database will enforce the traditional ACID constraints. Unfortunately, my derived data system can potentially negatively interact with the ACID system in the same sense that indexes (a form of derived data) may make ACID systems more difficult to operate efficiently. Updating one data item may involve "hidden" updates of other data items, thus increasing the potential for contention between transactions. This obviously may result in lower system throughput.

With so much potential overhead, carefully selecting which derived values to eagerly materialize is obviously critical to the system's performance. How my system decides that is the next chapter's topic.

Chapter 5

Strategy selection

My system selects among different combinations of caching and materialization choices in an attempt to minimize the time users spend waiting. The basis for its choices is minimizing time, and section 5.1 formalizes that concept. To make its choices, my system depends upon pragmatic estimates of what the various alternatives will cost in terms of time, and selects whichever approach yields the lowest cost estimate. Section 5.2 addresses such estimates in general. Section 5.3 presents the specific cost estimate formulas for the materialization and caching combinations considered. Then section 5.4 discusses how my system may collect the actual values needed by the section 5.3 formulas. Once cost formulas are available to estimate the different cost possibilities, this chapter addresses the details of making the actual choices. Section 5.5 discusses when the system must make its choices. Finally section 5.6 presents the actual algorithm.

5.1 Minimizing costs

My system will attempt to minimize the total time that users spend waiting. Thus if t_i is the total transaction time for the interactive transaction i, then my system will attempt to minimize the formula $\sum_{i=1}^{\infty} t_i$ over the i affected transactions[1]. My system attempts to minimize the summation of transaction times, not the individual transaction times. Obviously, when making decisions, my system does not actually consider **all** of the user transactions the system will ever process. For example, the value of t_i is clearly fixed for all previously committed transactions. However my system also avoids the other extreme; it does consider its effects on more than just the next

[1] Technically the full formula is $\sum_{i=1}^{\infty} t_i$ **over the i affected transactions** or $\frac{\sum_{i=1}^{\infty} t_i}{\infty}$. Much of the following discussion considers only the summation numerators, and omits the "transactions affected" denominator for simplicity.

transaction. Although in theory any current decision might have some effect on the durations of all future transactions, that effect will tend to be negligible or unpredictable on transactions far in the future. Therefore, my system cost estimates attempt to guess how many transactions will be significantly affected, and to estimate how they will be affected. My system then tries to minimize the resulting local summation $\sum_{i=n}^{n+x} t_i$ over the affected transactions. Many of my system's decisions are actually made in terms of derived data **accesses** rather than **transactions.** This simply means that each affected transaction will include one or more affected accesses, so the actual summation could be expressed as $\sum_{j=m}^{m+y} a_j$. However there is a straight many to one mapping from a_j to t_i, and the value of each t_i includes the sum of the associated a_j values. Thus minimizing $\sum_{j=m}^{m+y} a_j$ minimizes the local summation $\sum_{i=n}^{n+x} t_i$ which will in turn minimize the global summation $\sum_{i=1}^{\infty} t_i$. In summary, longer individual access times resulting in longer individual transaction times are acceptable, provided the total cost $\sum_{i=1}^{\infty} t_i$ pro-rated over all affected transactions is minimized.

5.2 Estimating costs

My system depends heavily on estimating future costs. The underlying assumption is that the future will resemble the past. So if last time an operation required T units of time, then next time it will probably again require about T units of time.

Obviously, using only the single value of the most recent operation's duration provides a rather crude estimate. However, at the other extreme it is also not desirable to average the durations of all previous operations. Such a massive average would respond with glacial slowness to changes in the database. Somewhat better would be a simple moving average of the past N values. However, that requires storing N values, and is still rather crude. The ideal would be a computationally simple formula which incorporates information about all previous operations, but which gives greater weight to the more recent past.

5.2.1 Geometric smoothing

One of the most popular solutions to this problem is referred to by various authors as simple (or first-order, or basic, or single) geometric (or exponential) smoothing:

the weights decrease geometrically with the age of the observations, so the name geometric smoothing has been attached to this procedure. Because these weights decline exponentially when connected by a smooth curve, the name exponential smoothing has also been used to describe this method.

[RW81] page 67

Calculating the next smoothed value for a time series x_N requires only three values. It requires the actual value x_N at time N. It needs the previous estimate S_{N-1} of that actual x_N value. Finally it requires α, the *smoothing constant,* a fraction which controls how much weight is given to recent values in the time series. The actual weights will be

$$\alpha, \alpha(1-\alpha), \alpha(1-\alpha),^{2}, \alpha(1-\alpha),^{3}, \ldots, \alpha(1-\alpha),^{N-1}, \alpha(1-\alpha),^{N}.$$

The geometric smoothing technique is to modify the old estimate S_{N-1} by the fraction α of the forecast error in predicting x_N using S_{N-1}. The forecast error is

$$x_N - S_{N-1}$$

so the new estimate is

$$S_N = S_{N-1} + \alpha(x_N - S_{N-1}).$$

A more detailed explanation of the mathematics involved may be found in section 20.3, *Arithmetic and Geometric Smoothing,* of [RW81]. Even more detail may be found in chapter 3, *Exponential Smoothing Methods,* of [MJ76].

5.2.2 The smoothing constant

Geometric smoothing's responsiveness is determined by the value of α. Small values result in a sluggish response to changes, while large values may cause the system to overreact to noise. The selection of α for particular applications is generally heuristically or experientially governed. The rough heuristic guidance is generally something like:

In general, the value of α should lie between 0.01 *and* 0.30.

[MJ76] page 52

The most popular experience-based alternative seems to be applying a minimum sum of squared errors criteria to tests of various α values on historical time series data. Some practitioners reportedly [MJ76] use several values. They use a higher value initially, and whenever the system seems to be changing rapidly, otherwise, they use a smaller value.

[Cho65] suggests an automated adaptive solution which uses three separate values for α, "high," "normal," and "low." Each solution is evaluated on the basis of some user supplied error criterion. If the "normal" value does not provide the "best" result based on the error criterion, the "outer" forecast with the best result becomes the new "normal," then new values for "high" and "low" are selected, and the process repeats. The complexity of this solution lies in the "user supplied error criterion." I know of no computationally simple general purpose error criterion.

5.2.3 My initial design

My system will thus behave heuristically, and allow the programmer to provide guidance and override the system defaults. My system can relatively easily collect historical data, and this is fairly inexpensive if the data is only read during the idle loop. So my system could assign an initial value to α based on idle-loop analysis of other initial behavior historical data. Once the initial period had passed, the system could then cheaply switch to an average value for all the α values computed from historical data. Eventually, once a significant store of historical data for this time series and surplus idle-loop time are both available, a custom value of α may be selected by applying the minimum sum of squared errors criteria to the historical data for this time series. Periodically, the values of α may be updated using the same procedure. If the predictions are found to be acceptably accurate, the system might also simply accept the current value as reasonable, and discontinue even the collection of historical data for such well behaved data items.

Alternatively the user might specify initial and normal values for α. In principle it might also be desirable to allow the user to provide a custom error criterion. However, where that is desirable it might be better for the user to entirely replace the default behavior, as described immediately below in section 5.2.4.

5.2.4 Other alternatives

I have selected "simple" geometric smoothing as a reasonable default approach for my system. It is certainly fairly simple to implement, and I expect it to be generally adequate. However there are "not so simple" alternatives available. For example section 3.3 of [MJ76] describes *higher-order* *smoothing.*

Although **geometric** smoothing may generally be adequate, it is based on a sometimes unwarranted assumption. Geometric smoothing assumes that observations consist of the mean plus a random error component. Some time series are instead highly dependent upon successive observations. Although geometric smoothing will often produce fairly good results on such time series, other forecasting methods can take advantage of such dependencies, and could thus produce even better results. [MJ76] suggests that if the optimum value of α is greater than 0.3, then such alternatives should be considered. Further detail may be found in chapter 9, *The Box-Jenkins Models,* of [MJ76].

> *Many of these forecasting techniques are based on recent developments in time series analysis recently consolidated and presented by Box and Jenkins [BJ70], and are called* **Box-Jenkins models.**
>
> [MJ76] page 189

In principle my system could be extended to offer such alternatives. Even more desirable would be extending my system so the user could provide their very own *cost estimate* functions, selectively replacing the system defaults.

5.3 The "modes"

My system must support and choose among a number of different "modes," each of which answer the question: *"How is the derived data computed?"* To choose among them, the system must estimate their relative costs. As described in chapter 2, the alternative "modes" my system supports (for both the "from scratch" and "incremental" approaches) are:

- Lazy
- Lazy with a "committed" cache.
- Lazy with a cache retained only during the transaction.
- Eager inside of the deriver-modifying transaction.
- Eager in the idle loop.

Their respective costs may be estimated as described below.

5.3.1 Lazy

For simple **lazy** mode, the expected cost per read equals the smoothed value of the previous attempts to compute from scratch, divided by ONE. The *"smoothed value of previous attempts"* (previousAttempts) represents the system's best guess of how much time will be required to compute the value next time. This number is divided by *"ONE"* to represent that only one **access** will benefit from this computation. Under this approach, it is possible that the system may be forced to recompute the value from scratch multiple times within the same transaction, despite none of the derivers having mutated! The simple formula (5.1) is shown below.

$$\frac{\text{smoothed(previousAttempts)}}{1} \tag{5.1}$$

5.3.2 Lazy with a "committed" cache

For **lazy** mode with a "committed" cache, the expected cost per read equals the smoothed value of the previous attempts to compute from scratch, plus the expected cost to monitor for mutations, all divided by the number of reads of the cache expected before the next mutation, plus one to represent the single immediate access associated with any "lazy" computation. Again, the *"smoothed value of previous attempts"* (previousAttempts) represents the system's best guess of how much time will be required to compute the value next time. Any "caching" alternative must know when to invalidate its cache, the *"expected cost to monitor for mutations"* (monitorCost) represents this cost. This sum is divided by *"reads before mutation"* (readsBeforeMutation) plus *"ONE"* to represent the expected number of accesses which will benefit from the costs incurred.

$$\frac{\text{smoothed(previousAttempts)} + \text{smoothed(monitorCost)}}{\text{smoothed(readsBeforeMutation)} + 1} \tag{5.2}$$

5.3.3 Lazy with a "temporary" cache

For **lazy** mode with a "temporary" cache, the expected cost per read equals the *"smoothed value of the previous attempts to compute from scratch"* (previousAttempts), plus the *"expected cost to monitor for mutations"* (monitorCost), all divided by the *"the number of reads expected before the next mutation* **or commit**" (readsBeforeMutationOrCommit) again plus *"ONE"* to represent the immediate "lazy" read. The difference between this formula and the previous "committed" cache formula is that under a "temporary" cache strategy, the system will discard the cache at commit. Thus the system may only balance its costs against the benefits received during this transaction.

$$\frac{\text{smoothed(previousAttempts)} + \text{smoothed(monitorCost)}}{\text{smoothed(readsBeforeMutationOrCommit)} + 1} \tag{5.3}$$

Unfortunately, lazy with a cache retained only during the transaction may be difficult to implement efficiently. One of the large costs associated with caching strategies is the need to monitor for mutations. Under a non-caching strategy, computing a derived value would obviously read all of the derivers, but it might mutate few if any of the derivers. However, my algorithm for supporting caching strategies requires tracking every derived-deriver dependency by updating a secondary structure associated with each deriver. The obvious way to implement this in an object-oriented language is by adding an additional instance variable to each object. Unfortunately, this natural solution causes all of the derivers to be written every time the derived is calculated using a caching strategy as was discussed in section 4.3. This is not a major issue when what is written (the deriver-derived dependency information) must be safely committed to stable storage anyway. However, if the information is "temporary" and will be discarded at commit, then efficiency requires that the system not naively believe that it must again save all of the derivers to stable storage, to update their "obsolete" values. A straightforward implementation could easily result in derivers being loaded from disk into memory, and only being read not written by the computation of a derived value; however, because an associated temporary derived-deriver-dependency structure is created or modified by the system to support a caching strategy, the underlying database might believe that the loaded object had been mutated, and thus must be stored to disk. The optimal action would in contrast be to notice that nothing needs to be written to disk because the data loaded from disk was not altered, and the temporary data structure should be discarded at commit anyway. How an object-oriented database keeps track of which values are "dirty" is generally outside the scope of my dissertation. I do discuss some preliminary ideas on how a database could support temporary writes in 7.3.2 under future work. However, for now in this case, my framework may interact with the underlying database's actions so adversely that the *Lazy with a "temporary" cache* option might never be useful, because the *Lazy with a "committed" cache* option would provide more benefits at a very similar cost.

As I discussed in section 4.3, this write problem could obviously be avoided by using a separate secondary structure to store the dependency information. Unfortunately that approach introduces the need for a table lookup upon every object access. Thus the computational overhead of that alternative seems prohibitive.

5.3.4 Eager inside of the deriver-modifying transaction

This formula actually appears almost identical to the "**lazy** cached" formula (see above). For **lazy** mode with a "committed" cache, the expected cost per read equals the smoothed value of the previous attempts to compute from scratch, plus the expected cost to monitor for mutations, all divided by the number of reads expected before the next mutation.

For **eager** mode with the update computed inside of the deriver-modifying transaction, the expected cost per read equals the *"smoothed value of the previous eager* **inside** *attempts to compute from scratch"* (previousInsideAttempts), plus the *"expected cost to monitor for mutations"* (monitorCost), all divided by the *"the number of reads expected before the next mutation"* (readsBeforeMutation).

$$\frac{\text{smoothed}(\text{previousInsideAttempts}) + \text{smoothed}(\text{monitorCost})}{\text{smoothed}(\text{readsBeforeMutation})} \tag{5.4}$$

The difference is that inside of the transaction which changed a deriver, there is a fairly good chance that many of the derivers for the affected derived value are already loaded in memory, thus it may be cheaper to update "now" than to wait and update in a LAZY manner. This was previously discussed in section 3.3.

The other major difference is that unlike the LAZY options, the expected number of reads may be a fraction less than one! By definition, all of the LAZY choices will satisfy at least one access, however the cost of an EAGER evaluation might be entirely "wasted" if a deriver is modified before the derived value is next accessed.

5.3.5 Eager in the idle loop

My system attempts to minimize user waiting time, so computation time during the idle loop is not counted. Thus eagerly computing all derived values during the idle loop would at first seem to be the ideal strategy. However, recomputation within the idle loop is not always possible, since idle-loop time is a finite resource which is not always available between two transactions. Even when idle-loop time is available, and thus the eager recomputation is "free," an eager strategy also requires that the system monitor for deriver mutations. Because a previously computed derived value must appear to be immediately invalidated whenever a deriver is changed, it is not acceptable to simply invalidate the derived value whenever the system happens to get around to it during otherwise idle time. My system's solution to the monitoring problem is described later in this chapter; here the important point is that many of the costs involved in monitoring for deriver mutations can not be moved into the idle loop. Even if there was plentiful idle-loop time between every user transaction, some user transaction might modify a deriver and then wish to read the corresponding derived value; thus some of the monitoring cost must be incurred inside of user transactions. Therefore, even if idle-loop time is available for recomputation, the cost of monitoring could potentially be greater than the savings of recomputing in the idle loop.

To compute the expected cost for eager in the idle loop, the system will first compute the cost per read assuming that idle-loop time will be available. That cost is the *"expected cost to monitor for mutations"* (monitorCost) divided by the *"expected number of reads before the next mutation"* (readsBeforeMutation). The system will also need to compute the *"best "LAZY" alternative per read cost"* (lazyCostPerRead), the minimum of the formulas previously described in sections 5.3.1, 5.3.2, and 5.3.3. It will then need to multiply those costs by the corresponding *"expected probability that idle-loop time will or will not be available"* (probabilityOfIdleTime), discussed in 5.4.5, and sum the results to get the expected cost per read for this alternative.

$$\left(\frac{\text{smoothed(monitorCost)}}{\text{smoothed(readsBeforeMutation)}} * \text{smoothed(probabilityOfIdleTime)}\right)$$
$$+ \quad (\text{lazyCostPerRead} * (1 - \text{smoothed(probabilityOfIdleTime)})) \tag{5.5}$$

5.4 Monitoring

Section 5.3 gave formulas for estimating the costs of different "modes." Those formulas included the terms:

- smoothed(previousAttempts)

- smoothed(previousInsideAttempts)
- smoothed(readsBeforeMutation)
- smoothed(readsBeforeMutationOrCommit)
- smoothed(probabilityOfIdleTime)
- smoothed(monitorCost)

Section 5.2 explained how values may be "smoothed." This section will explore what my system must do to collect the raw data; it focuses on how the system will measure costs and monitor usage so that reasonable estimates can be made using the formulas from section 5.3. Because semantic reasoning is generally impractical, observation of past behavior, or monitoring, will need to provide the primary source of information for my system's optimizations.

5.4.1 previousAttempts

Some of the formulas in section 5.3 use "smoothed(previousAttempts)." This represents the cost of previous attempts to compute a derived value from scratch. To calculate that value, each time the derived value is computed, the system must smooth the old previousAttempts value with the actual cost of the current attempt, to get a new value for previousAttempts. Fortunately measuring the current cost is simple. The system merely notes the wall clock time, calculates the derived value, then subtracts the saved wall clock time from the current wall clock time. The result is the duration of the calculation, which my system treats as the cost.

$$
\begin{array}{l}
A := Clock\ now. \\
\text{compute derived} \\
Time := Clock\ now - A.
\end{array}
$$

Many different things might contribute to the elapsed time the system observes. Obviously all calculations require some CPU time, and thus some wall clock time. Some calculations will also require disk or even remote network I/O delays. Under a multiuser timesharing system, a calculation might be delayed by resource contention, such as waiting for CPU time. An ACID database system might also delay the calculation because it is waiting for other transactions to release their locks. Some of these costs are intrinsic to the calculation, while some depend heavily on the total computer system's usage. Fortunately I want the costs the system calculates to reflect all delays, not merely the delays that would occur if the calculation had the system to itself. Thus my system can use a very simple formula to calculate the duration, and may ignore what caused it.

Derived of Derived?

What if the "compute derived" step triggers the computation of some other derived value; does that affect how my system should count the time? My system does not treat this case specially. Sometimes computing from scratch may include the time required to compute another derived value. However, presumably sometimes the other derived value will have been precomputed. So my system uses the simplest implementation, and does not treat this as a special case.

Because derived values immediately create a bidirectional link when they become derivers (see 2.1.1 for details), it would be possible to implement the other alternative, and to "stop the clock" during the computation of derived of derived values. If this were done, then only some portion of the cost of computing the derived value would be charged against the outside derived value computation. See 5.3 for examples of estimating the costs per access of computing derived values. However, that same cost should then be charged even when the derived value referenced was previously computed.

5.4.2 previousInsideAttempts

The "smoothed(previousInsideAttempts)" value is calculated the same way as the "smoothed(previousAttempts)" value just described. The difference is that only calculations of the derived value which occur **inside** of transactions which previously changed a deriver are considered. The reason, as described in section 5.3.4, is that inside of a transaction which changed a deriver, there is a fairly good chance that many of the derivers needed to calculate the derived value are already in memory. Thus there will tend to be fewer I/O delays involved in recalculating the value inside such a transaction.

5.4.3 readsBeforeMutation

The "smoothed(readsBeforeMutation)" value represents the number of reads expected before the next mutation of the derived value. It is rather simple to count the number of reads which occur. My system must include code which upon each access checks that the derived value is valid, and triggers a lazy evaluation if needed. This code can easily also increment a counter upon every access, and when a a valid *IveChanged* message is received update readsBeforeMutation with the current value of the number of reads counter (possibly zero), and reset the counter.

To save storage space, this variable is used in formula 5.2 for *lazy with a "committed" cache,* and also in the eager evaluation formulas 5.4, and 5.5. To accommodate this multiple usage, the readsBeforeMutation variable does not count the immediate read associated with a "lazy" evaluation. It only counts the reads which access the cached value.

5.4.4 readsBeforeMutationOrCommit

The "smoothed(readsBeforeMutationOrCommit)" value strongly resembles the readsBeforeMutation" version, but only counts the number of reads expected before the next mutation **or commit** of a derived value. This value can be handled the same way as readsBeforeMutation described in section 5.4.3 above, except that commit also causes an update of readsBeforeMutationOrCommit and resets the counter. For simplicity, this variable like the readsBeforeMutation" version only counts reads of the cached value, thus its value might be less than one.

5.4.5 probabilityOfIdleTime

The "probabilityOfIdleTime" value represents the chance that idle-loop time will be available to recompute a particular derived value before it is next accessed. My system obviously knows when it has scheduled a derived value for computation during the idle loop. If that scheduled computation occurs before the value is accessed, than the smoothing formula is applied to update probabilityOfIdleTime using 1 as the actual value. If instead the derived value is accessed before it is computed, though it was scheduled for computation in the idle loop, then the smoothing formula is applied using 0 as the actual value. I consider this solution somewhat crude, but it does have the redeeming virtues of being simple and sufficient.

5.4.6 The complete cost of monitoring

Probably the biggest cost difference between eager and lazy evaluation modes is the additional monitoring cost associated with a caching approach. Thus the expected cost to monitor a derived value's derivers for mutations is very important. Unfortunately, it is also extremely difficult to measure; unquestionably, the hardest of the costs to measure in this section is the "smoothed(monitorCost)" described in 5.4.8 below. In that section I will focus on the additional costs of monitoring which are incurred when my system selects eager evaluation mode instead of lazy evaluation mode for some derived value. However this section is devoted to the broader question of what monitoring costs my system incurs to support eager evaluation at all.

The fundamental cost of monitoring is detecting when it is necessary to send *IveChanged* messages (see section 2.1) and then actually sending the *IveChanged* messages. Solving these problems more than doubles the user's code size.

First, derived values must be matched to derivers. The system does this while the derived value is being computed. As section 2.1.1 said

> "Each object accessed is sent the message `youAreDeriverFor: derivedValueOOP at: timestampVersionNumber`."

To do this, my system actually creates a second copy of the user's code which sends this message to every object accessed. This second copy is used when and only when a derived value is being computed. The relevant `derivedValueOOP` and `timestampVersionNumber` values are simply kept in global variables during the computation.

Both the second copy of the user's code just described, and the "normal" user's code must also be altered to check for deriver mutations. All code which might alter a deriver must contain additional overhead code such as that in table 5.1. In an strictly encapsulated object-oriented system, these

```
code alters a deriver
if (self has dependents)
        send all dependents IveChanged messages
```

Table 5.1: Code needed every time a deriver mutates.

checks are at least nicely localized inside of the object's associated code. However, the cost can still be significant.

Basically my system must insert something resembling the table 5.1 code after every assignment statement which might mutate a deriver. This may be optimized somewhat, as I will describe later in this section, so that not every one of the user's assignment statements will directly require such a check. However, the number of such insertions is still definitely very significant. Such insertions have five basic costs:

1. The code expansion cost.

2. The if (*self has dependents*) check runtime cost.

3. The additional storage costs.

4. The cost of sending *IveChanged* messages when needed.

5. The cost of adding up all of the other costs!

Under my system, an object may become a deriver at any time. An especially painful case is that the system might begin executing a dynamic method for an object A which is initially not a deriver. That method might then call some other method, which in turn might call other methods, ..., eventually resulting in a call which accessed a derived value B. The computation of that derived value might access the original object A, and thus create a deriver-derived dependency between A and B. When eventually all of those calls returned to A's method, A would now be a deriver, and thus should monitor itself for changes. However, my system supports such monitoring by altering the code. Thus upon returning from a call, there could be a need to change what code is run! It is certainly theoretically possible to handle such cases. However, as it is not central to my dissertation, I only briefly describe this possibility in future work section 7.3.5. Instead I handle such problems by assuming that all the code associated with potential derivers is always instrumented for monitoring.

The downside of this is that some of the costs:

1. The code expansion cost.

2. The if $(self\ has\ dependents)$ check runtime cost.

and some of:

3. The additional storage costs.

are always incurred even if none of the derived values are ever actually maintained eagerly.

Code expansion cost

Creating two separate copies of the user's code, and adding the additional code described above will obviously result in larger code. There is a cost associated with loading larger code. However, that cost is very difficult to quantify; hopefully it is relatively trivial. That hope is based on the idea that usually disk access costs totally swamp all other costs, and it is relatively cheap to load an extra block provided this does not require an additional seek operation. It is important to note that this additional cost should theoretically be amortized across ALL of the derived values which force this object to be an deriver. Under my current system, that actually consists of every derived value which might use a caching strategy. The code expansion cost is the same if there is one derived value dependent, or a million. Unfortunately under my current system, this code and thus this cost always exists. Thus my system does not consider this cost when deciding between different evaluation modes, because this cost will be unavoidable regardless of the evaluation mode chosen.

Runtime *if* (*self has dependents*) check cost

The next cost is the runtime cost of performing the *if* (*self has dependents*) check. Since this check will obviously only be run when the *self* object is already loaded into memory, and merely involves checking a flag, this check operation should be quite fast when compared, for example, with a disk access. Unfortunately, almost every assignment to an instance or global variable will result in such a check. The check is inexpensive, but there are a very large number of them. Fortunately we can optimize away at least some of them.

My system is in part based on the idea that the system will take the user's code, and modify it by inserting additional code such as that shown in table 5.1. A straightforward implementation would simply naively add the *if* (*self has dependents*) check after every assignment statement that mutates a deriver object. As the check merely involves testing a boolean, that could even be tolerable. However, it is possible to be somewhat wiser, though complete optimization is really a compiler problem, and beyond the scope of my thesis.

The basis for this improvement is that, after a mutation, *IveChanged* messages do not need to be sent immediately, but only before any derived values are accessed. So long as the system is certain that no derived values are being accessed, sending *IveChanged* messages may be deferred. This is safe because the object's set of dependents can only change when a derived value is computed, and if the set of dependents is not altered, *IveChanged* messages must merely be received by their associated derived values in time to invalidate any cached values before they are accessed. This is obviously only a problem when a derived value is accessed, thus it is safe to defer the *if* (*self has dependents*)... check until just before a derived value might be accessed. Obviously, two consecutive *if* (*self has dependents*)... checks are redundant, and may be combined into a single check. Thus it is possible to combine and defer multiple *if* (*self has dependents*)... checks until right before the code might next access a derived value. Unless the system has extensive inter-procedural analysis, that will probably mean deferring the check until right before the next function call or message send. This should, at the very least, generally permit moving the check code out of tight loops.

I just described a way to decrease the number of times the system must perform the *if (self has dependents)...* check. However this check can never be entirely eliminated. Again it should be noted that this cost should theoretically be spread across ALL of the derived values forcing the existence of the check. Under my current system, that consists of all derived values which might ever use a caching strategy. It is tempting to think it only consists of the derived value dependents for the object containing the check code, however that is incorrect. Assume that at some point none of the derived values in the entire system were currently using a caching strategy. The cost would still exist because one of them might unexpectedly be switched to a caching strategy. The *if (self has dependents)* check cost is essentially the same whether the deriver has zero or a million dependents. (The cost of actually sending the *IveChanged* messages is considered below.) Under my current system, this cost is unavoidable even if an object has no derived value dependents; so again my system does not consider this cost when deciding between eager and lazy evaluation strategies.

5.4.7 Load, store, and contention costs

A number of additional variables are required to upgrade a system from a database which only supports the lazy evaluation of derived values without any caching support, to my system which supports the caching and eager evaluation of derived values. The variables required were described in chapter 4. Adding variables to a database system incurs new storage costs to load and store the variables, it also creates new possibilities for contention between transactions. Those costs are the subject of this section.

As I described in section 4.1.2, my system adds additional instance variables to all potential derivers. There is obviously some additional cost involved with loading larger objects than would otherwise be required. However, all potential derivers incur this cost regardless of their number of derived value dependents. My system also specifies some instance variables for all derived values with similar additional costs. In both cases, the costs are unavoidable if any derived value might use a caching strategy. Thus my system again need not consider this cost when selecting an evaluation strategy.

Some of those additional instance variables will refer to other support objects. Especially significant is the collection of deriver-derived dependency links. Obviously there is additional cost involved in loading these objects, compared to a database which did not incorporate my system. These support objects must be loaded when a derived value is recomputed. However, that cost is already lumped into the recalculation cost described in section 5.4.1 above, thus I do not need to consider it further here. The deriver-derived link collection (DDLC) must also be loaded whenever a deriver is mutated in order to send *IveChanged* messages. That cost is considered in the next section, where I consider the cost of sending the *IveChanged* messages. The additional cost of loading the derived value itself when sending *IveChanged* can be accurately measured, and will also be described later.

Storing at commit

Objects may be loaded at almost any time during a transaction, but I assume they are only written as part of a commit operation.[2] I have assumed that the underlying existing database system handles locking, and determines which objects have been written by a transaction. When a user's transaction commits, this underlying system must also obviously store any objects which have been altered. My system will as previously described have created, altered, or enlarged some of the objects which must be stored. Important examples include the dependency link collections, any derived values computed during the transaction, and the monitoring variables. While having my system make very minimal assumptions about the underlying database system is clearly a desirable thing, here it creates a problem. My assumptions have not even specified whether the underlying database system is implemented as a "page-server" or as an "object-server." Obviously, without low level system knowledge, my system can not directly determine how much of the time spent storing variables at commit was spent on the user's variables, and how much was spent storing variables which would not exist in a purely lazy, uncached system. Therefore, as my system can not directly determine the costs involved, I assume that the underlying system provides a reporting function my system can call to measure these store costs. So before committing, to measure store costs, my system will call the underlying database with lists of the objects needing cost data, and a location to store that cost information. Then at the end of the commit pro-

[2]This assumption has the side affect that when a transaction is aborted, none of the monitoring variables are written. I do not consider this a major defect, because if the database system is aborting a significant number of transactions, it has larger problems than merely neglecting some history when my subsystem selects materialization strategies.

cedure, the underlying database will compute and store the requested cost information. Given that assumption, the system can rather directly sum the store costs associated with a given derived value.

If this assumption that the underlying database will provide a cost function proves impractical, there is a crude alternative. Though my system may not directly measure the portion of commit time spent on system variables, it can if necessary make a fairly good estimate of how many and what size variables are involved. The implementors of my system could create and run a number of calibration tests, which caused the underlying system to commit transactions with known combinations of dirty values. As a result, some admittedly crude constants could be developed. These constants would allow my system to estimate commitment time store costs based on the number and size of the system variables involved.

Contention

In section 4.5, I described probably the hardest-to-quantify cost for my system. There are cases where two transactions which without my system might peacefully share a read lock must instead acquire write locks on a shared deriver's dependency link collection. Obviously, the actual contention cost is also very sensitive to the underlying database implementation. A "page-server" database will probably have a different locking granularity, and thus different contention characteristics, than an "object-server" database.

This is definitely a cost that my system can not measure directly. The whole point of transactions is that only the underlying database will ever notice when one transaction causes another transaction to block. My system does manage to indirectly measure some of this cost when in section 5.4.1 it measures the total wall clock time required to compute a derived value. A similar approach to measuring the cost of sending *IveChanged* messages in section 5.4.8 below will also capture some of this contention cost. However, it definitely seems impossible for my system to measure all of this cost without low-level access to the underlying database.

Even with low-level access, it is unclear how to allocate such costs to individual derived values, though the costs are definitely dependent upon the existence of cached derived values. My not entirely satisfactory solution is to assume that my system can inform the underlying database of all the extra objects that my system creates, and that the underlying database can return some time cost based on how long transactions spend blocked waiting for those objects. This cost is then divided by the number of cached derived values, and that number is then added to the monitorCost estimate for all derived values. Clearly a better solution would be very desirable.

5.4.8 monitorCost

The "code expansion," and "*if* (*self has dependents*) check" costs are unavoidable overheads required to support the possibility that a derived value might use eager evaluation. As just described, some of the "additional storage costs" are unavoidable overheads while some do depend upon the derived values which are eagerly computed. This section addresses the other costs I listed which depend upon how many derived value dependents an object has. Thus this section considers:

4. The cost of sending *IveChanged* messages when needed.

5. The cost of adding up all of the other costs!

Unfortunately, some of these costs are distributed instead of being nicely localized. Sometimes these costs also need to be amortized over many different derived values, instead of a single derived value.

Sending *IveChanged*

Happily, the time cost of sending the *IveChanged* messages can be reasonably measured. This is especially important because this cost may be significant. This cost should be trivial if the derived value is already in memory. However, if the derived value is not in memory, the cost will obviously be significant. My system currently assumes that sending an *IveChanged* message will block the sender until the receiver (a derived value dependent) is loaded and processes the message. A future implementation might enqueue the loading of such derived values into memory, because my system will generally not need them until trying to commit. Such an implementation would not block until the derived value was "normally" accessed, or needed at commit. That possible optimization is beyond the scope of my dissertation. Fortunately, to measure the cost, we merely need to add an additional argument to the *IveChanged* message itself, and to create a new object which we finalize after completing the message send. The basic code to measure the cost is:

$$
\begin{aligned}
&A := Clock\ now.\\
&\text{send } IveChanged\\
&Time := Clock\ now - A.
\end{aligned}
$$

We can enhance that by writing:

$$A := new\ TimeCostObject(Clock\ now)$$
$$\text{send } IveChanged(A)$$
$$A.secondTime(Clock\ now)$$

The *TimeCostObjects* allow us to get around a variable scoping problem. The *IveChanged* messages are sent by derivers, which do not know the cost of sending the message until the call returns. However, it is the derived values, not the derivers, which will later need to know the cost. The solution is simply to create a tiny mutable *TimeCostObject* object, pass its reference to the derived value, and finalize the *TimeCostObject* after the time required to send *IveChanged* is known, but before the derived value needs it. An alternative implementation would first send *IveChanged*, which might result in disk access, then immediately send a second message containing the cost of sending the first message. The reasoning is that the second message could not be very expensive, since it would not result in a disk access under any reasonable system. Under either approach, later the total message send costs can be added up when the system recomputes derived objects. Notice, the algorithm described in 2.1 will need to save the *TimeCostObjects*, even if it is otherwise totally ignoring the associated *IveChanged* messages. It should also be noted that this cost can be directly allocated to individual derived value dependents.

As I discussed in section 5.4.6 above, the deriver-derived link collection (DDLC) must be loaded whenever a deriver is mutated. The actual cost of that load obviously depends upon the size of the collection. We would also like to allocate the cost across all of the derived values in the collection. This can be done with a rather simple extension to the *TimeCostObject* solution described above. First we determine the cost to load the DDLC per derived value:

$$B := Clock\ now.$$
$$\text{load the DDLC}$$
$$C := size\ of\ collection$$
$$Time := Clock\ now - B.$$
$$D := Time \div C$$

Then we add that prorated cost to the *TimeCostObjects* when they are created, so that code becomes:

$$A := new\ TimeCostObject(D, Clock\ now)$$
$$\text{send } IveChanged(A)$$
$$A.secondTime(Clock\ now)$$

Thus, when eventually the derived values add up their *TimeCostObjects* they will also be adding up their share of the DDLC load costs.

Cost of adding the costs

Finally, there is the cost of adding up all the costs. Hopefully that cost will always remain relatively insignificant. To the extent that this measurement cost is incurred inside of a user's transaction, the cost of adding the costs may also largely be measured using a simple:

$$\begin{array}{l} A := Clock\ now. \\ \text{add the costs} \\ Time := Clock\ now - A. \end{array}$$

calculation. Ideally, this cost should also be amortized over the relevant objects.

Only an estimate is needed

Some of the cost calculations described above are clearly imperfect. It might theoretically be possible to "perfectly" measure all of the costs above. However, clearly that would entail an even more significant measurement cost, and thus is not desirable when all that is really required is a cost estimate.

5.5 When to choose

When an unmaterialized derived value is accessed, the "eager" alternatives are by definition no longer available. The system must use a "lazy" alternative, and must immediately select between maintaining and not maintaining dependency information while calculating the required value.

If my system is going to select the *"Eager inside of the deriver modifying transaction"* choice, it must obviously make that choice before the *"deriver-modifying transaction"* commits. It is easy to see how the system might see a clear advantage to some other choice at that time, and thus immediately make another choice. For example, the system might at that time schedule the derived value for recomputation during the idle loop.

If the system does not immediately select an alternative before the *"deriver-modifying transaction"* commits, then the choice may be deferred, at the cost of eliminating the *"Eager inside of the deriver-modifying transaction"* alternative. If the system eventually has surplus idle-loop time, and finishes recomputing all the derived values previously scheduled for *Eager* *in the idle loop* recomputation, then the system might profitably reconsider eagerly computing other unmaterialized derived values.

The decision deadline	**for mode**	(sub-mode)
might be forced when accessed	**Lazy**	Once Temporary cache Committed cache
before commit	**Eager**	inside deriver-modifying transaction
before accessed	**Eager**	inside the idle loop

Figure 5.1: Summary of available materialization choices.

5.6 The actual algorithm

My system must select among five possible materialization choices.

1. *Lazy* as described in section 5.3.1.

2. *Lazy with a "committed" cache* as described in section 5.3.2.

3. *Lazy with a "temporary" cache* as described in section 5.3.3.

4. *Eager inside of the deriver-modifying transaction* as described in section 5.3.4.

5. *Eager in the idle loop,* as described in section 5.3.5.

As was discussed in section 4.3, until an efficient means is developed to "temporarily" write a deriver, the *Lazy with a "temporary" cache* alternative will never be a good choice. However, that still leaves four potentially reasonable choices to consider.

My system considers these four choices at three different times.

1. Inside of a user transaction when an uncached value is accessed.

2. Inside of a user transaction before the transaction is committed, but after the user's code has indicated that it is ready to commit.

3. During the idle loop when all derived values previously scheduled for *Eager in the idle loop* have already been computed.

5.6.1 The derived value is needed NOW!

When an uncached derived value is accessed, its value is by definition needed immediately. The time that the "Lazy" modes were waiting for has come. The system must calculate the derived value immediately. The first step of that calculation is deciding whether this time the system will cache this derived value. To make this decision, the system calculates the two lazy formulas, formula 5.1 and formula 5.2. The system obviously uses whichever choice corresponds with the lower cost estimate.[3] If plain lazy evaluation is selected, the system computes the derived value without creating dependency links, and does not cache the result. If *Lazy with a "committed" cache* is selected, the system does create dependency links, and does cache the result. In either case, once the derived value is computed, the user's code resumes execution. When triggered by an access of an unmaterialized derived value, my system will only select a materialization mode for that one derived value.

5.6.2 At commit

In section 2.1.2, I wrote that when a derived value had to invalidate its cache it informed a global *operations to be performed at commit object* that it needed to be updated. During the user's portion of a transaction, the global commit object merely kept a collection of these derived values. All other work was deferred, for the simple reason that the user's transaction might have aborted on its own, and there was no reason to delay the user of such transactions. However, now that the user is ready to commit, some calculations are desirable.

In section 3.4, I expanded upon the operations performed at the end of a transaction by discussing how derived of derived dependency trees should be evaluated from the leaves up to minimize the system memory required, where the leaves are derived values with no other derived value dependents. Building such a tree also assists in enforcing the "must be lazy" constraints. If a leaf gets forced into "must be lazy" mode, all of its parents can also be immediately forced into lazy mode.

Thus at commit the commit object first builds derived of derived dependency trees from its collection of derived values mutated during this transaction.

[3]The first time through some of the costs needed by formulas 5.1 and 5.2 may still be unknown. By initially setting all unknown costs to 0, I can cause the system to explore most of the available options, thus collecting the data which allows it to make informed choices later.

Two possible implementations are reasonable here. A simple local cost evaluation, and a more global evaluation. I describe a local cost evaluation strategy here, and defer global strategies to future work in 7.3.3.

After building the tree, the commit object begins processing the leaves. For each leaf, it evaluates the four reasonable cost formulas.

If *Eager inside of the deriver-modifying transaction,* described in section 5.3.4 and formula 5.4 provides the lowest cost estimate, the derived value is computed immediately. Then the next leaf is processed.

If *Eager in the idle loop,* described in section 5.3.5 and formula 5.5 provides the lowest cost estimate, the derived value is scheduled for recomputation in the idle loop. Then the next leaf is processed.

If one of the lazy choices provides the lowest cost estimate, the system is done processing this leaf. Either *Lazy* as described in section 5.3.1 and formula 5.1, or *Lazy with a "committed" cache* as described in section 5.3.2 and formula 5.2 is an acceptable lazy choice. While if a lazy mode is selected for a leaf no other processing needs to be done on this derived value, one other action might be desirable. At this point two reasonable system implementations seem practical. One possible implementation would immediately assume that all of the parents of this derived value should also automatically be placed in lazy evaluation mode, and be immediately removed from the trees being processed. The other reasonable implementation would note that while the system happened to choose lazy evaluation for this value, it was not forced to use lazy evaluation. Thus if a parent's cost formula indicates that it should use eager evaluation, then the parent's computation may later change the decision for this leaf. At this time, I have no reason to prefer one of these approaches over the other. However, which choice is selected will obviously affect how the "previousAttempts" cost should incorporate the costs of supporting derived of derived derivers, as discussed in section 5.4.1.

5.6.3 In the idle loop

The one other time my system might consider what evaluation strategy is appropriate is when the system finds itself with unused idle-loop time. At that time it might recompute the cost estimates for some of the unmaterialized derived values. The two lazy choices would still need to be considered as before. However the other available option, *Eager in the idle loop,* would now be slightly different. Formula 5.5 had two parts, depending upon the probability of idle-loop time being available. When the choice is being reconsidered inside of the idle-loop because of surplus idle-loop time, we assume that enough idle-loop time will remain available to compute one more derived value. Thus the cost estimate simplifies to that shown in formula 5.6.

$$\frac{\text{smoothed(monitorCost)}}{\text{smoothed(readsBeforeMutation)}} \tag{5.6}$$

That formula change may result in a different choice this time. If *Eager in the idle loop* does have the lowest cost estimate, obviously the system will immediately compute this derived value. It will then select another derived value to reconsider. I currently assume the derived values selected will be randomly selected. Other than avoiding reconsidering the same derived value multiple times, I presently have no reason to prefer one derived value over any other.

5.7 The idle-loop scheduler

My general system assumptions in section 1.3 included the assumption *"that the database system is not* always *busy (that I can use the idle loop)."* The details of how that will work are obviously very dependent upon the underlying database system. Minimally, my system will include an idle-loop scheduler object as previously mentioned in section 4.1.5. This object will keep a queue of the derived values needing to be evaluated in the idle loop. Presumably the underlying database system will provide my system with a low priority thread which is run whenever the system has idle-loop time available. In essence, that is my system's definition of idle-loop time. Idle-loop time is the time when the underlying database system schedules my system's low priority support thread. The support thread will, as its highest priority task, spawn transactions to compute the derived values in its queue. If the queue becomes empty, it will reconsider eagerly evaluating other derived values as just described in section 5.6.3. That is how my system can function using just the minimum support required from the underlying database system. Obviously some underlying database systems may have much more elaborate scheduling systems, under which my system's idle-loop tasks must coexist with other idle-loop tasks already used by the underlying database system. I anticipate no unusual difficulties in such an integration, but it would obviously be very specific to the particular underlying database system.

Chapter 6

Incremental updates

Up to this point, all of my work has assumed that a derived value is computed from scratch, and then possibly cached. If the cache is invalidated, the derived value is again recomputed from scratch. Obviously it would sometimes be desirable to use a more incremental approach. If a derived value with many derivers was often invalidated because only one of its derivers tended to change, while its other derivers remained unchanged, updating the derived value without entirely recomputing it from scratch might be both possible and very desirable. Of course, supporting such incremental updates would also require more overhead, and thus would not always be desirable.

6.1 The user's granularity choice

Though it might not have been obvious, my system as previously described already supports an incremental approach. Any time that the programmer believes some derived value is so large that it should be incrementally maintained, the programmer is free to redefine the large derived value in terms of other derived values. For example, any time the user has:

```
derived = f() + g() + h()
```

they may rewrite it as:

```
derived1 = f()
derived2 = g()
derived3 = h()
derived = derived1 + derived2 + derived3
```

Then hopefully the change of a single deriver will only cause one of the new derived values to be recomputed, while reusing the work previously done computing the other two new derived values.

Using this approach does require additional storage to cache the state information for the intermediate derived values, and their associated monitoring variables. However, as described in section 2.1.1, the "derived" value above would only need dependency links with "derived1," "derived2," and "derived3." It would not need dependency links with all of their derivers. Thus the total monitoring cost would not significantly increase. This is the main way I expect my system to support incremental updates.

6.2 Running-total-style derived values

The solution described in section 6.1 is fine for handling objects defined in terms of instance variables that contain either references to similar objects or primitive values. However, it does not handle large composite objects[1] well. One classic example it does not support is the idea of a "running total."

Almost everyone seems to understand that when you have a collection of numbers, you may either compute the total from scratch when you need it, or you may keep a running total as you add and delete numbers from the collection. If total is a derived value, and the collection is in fact a collection of immutable primitive values such as integers, then the only mutable deriver for total would be the collection itself. The rest of this chapter will focus on extending my system to support running-total-style derived values.

6.3 Delder values

The first thing that is needed is a different kind of derived value specification. Up until now, the user has declared a derived value by providing a method which computes the derived value from scratch. My system then did the rest of the work of deciding when to call the user's method.

Now, the user will need to declare their derived value in terms of a function that applies deltas to the old value. To reinforce the idea that we are dealing with a different form of derived value, I will refer to such values as "delder values.[2]" Initially let us have the user provide the following methods to declare a delder value:

reset() Which predictably resets the delder value to an initial state corresponding to an empty collection.

[1]Composite objects such as arrays, collections, tables, and vectors.

[2]I coined the term "delder" from **del**ta **der**ived value to distinguish between derived values which are computed based on repeatedly adding deltas and derived values which are computed from scratch.

add(anItem) Which is called when anItem is added to the collection. It updates the delder value appropriately.

remove(anItem) Which is called when anItem is removed from the collection. Again it must update the delder value appropriately.

The user will also need to specify the underlying collection. Therefore, my system may compute a delder value from scratch by first calling reset() and then iterating over the entire collection calling add(anItem) for each element of the collection.

Thus, for example, the user could declare a running-total delder value with code something like that shown in table 6.1.

```
delder RunningTotal {
        myValue=0;
        reset() { myValue = 0; }
        add(anObject) { myValue = myValue + anObject; }
        remove(anObject) {myValue = myValue - anObject; }
}
```

Table 6.1: Simple pseudo-code RunningTotal example.

The system would automatically provide a default "constructor" which associated the delder with its collection, and called the delder's reset() method. So to borrow more Java syntax, the user would write something like `new RunningTotal(aCollection)` to create an instance of the delder value RunningTotal.

6.4 Collections of immutable objects

If a delder value's collection is composed entirely of "effectively immutable objects," then the only mutable deriver is the collection itself. In this context, an object is "effectively immutable" *if and only if all of the objects accessed by any messages sent to the object by the delder (and recursively any other messages that are sent while processing the original message sends) are immutable.* For example, assume a collection of employees, where an employee has a name, address, and salary. Then further assume that we have a delder which totals all employee salaries. That delder would not care if the employee changed their name or address, it only needs the salary to remain immutable.[3]

[3]For employees to be "effectively immutable" in this example, an instance of employee's salary must not change. Thus to implement a raise would require making a new instance of employee, with the new salary, and removing the old instance.

At first it might instead seem that merely the object itself must be immutable; however, it is simple to see that if the primary object contained a reference to some mutable object, the delder might experience the primary object as mutable. To solve this, we might say the objects must be "recursively immutable." We could define "recursively immutable" objects as objects which are either primitive-immutable values, or objects which both are themselves immutable and also contain only recursively immutable objects in their instance variables. However, this is both too strict, and not strict enough. It is not strict enough, because with that definition a recursively immutable object's method might still reference a mutable global variable. However, it is also too strict, in the sense that it forces instance variables we don't care about, such as name and address in the example above, to be immutable. The algorithm described in this section will be expanded in section 6.5 to handle collections which contain objects that are not "effectively immutable."

I will also assume that a delder's collection can only be mutated by resetting itself to empty, adding, or removing an item from itself. I will assume that all other operations on a collection will be defined as a sequence of these primitive operations. Given these assumptions, to support a delder value, my system must simply treat the collection's reset(), add(), and remove() methods specially. Where such methods would normally send an *IveChanged* message to all derived value dependents, when the dependent is a delder value, such methods will instead send a message causing the delder value to call its corresponding reset(), add(), or remove() method. I will call these "delta messages." The delder might not process the delta messages immediately, but will process them before providing its "cached" value to an accessor.

6.4.1 Summary of this algorithm

In summary, the objects stored in the delder's collection may not mutate. Thus the delder's only mutable deriver is the collection itself. The collection may only mutate via its own reset(), add() and remove() methods. These methods will be treated specially. Instead of the previously described *IveChanged* messages, when they are called they will send any delder dependents *delta messages* which the delder objects will enqueue. Before responding to an access request, the delder object will process any enqueued delta messages by calling its own corresponding reset(), add(), or remove() method.

6.5 Collections of mostly immutable objects

In section 1.3.1, I assumed that the simple case was the common case, and that thus most objects my system dealt with would be primitive-immutable values. However, it remains essential that my system produce "correct" results even when it encounters complex mutable objects. So we need to extend the algorithm just given in section 6.4.1 to handle the possibility that the collection will not be entirely composed of effectively immutable objects.

Suppose instead we have a collection containing instances of the immutable class FixedCost and the mutable class Professor; assume that both classes return an integer when sent the message yourCost(). The revised RunningTotal example code fragment is shown in table 6.2.

```
class FixedCost {
        myCost;
        FixedCost(int theCost) { myCost = theCost;}
        int yourCost() { return(myCost); }
}
class Professor {
        salary;
        int yourCost() { return(salary); }
        void newSalary(int amount) { salary = amount; }
}
delder RunningTotal {
        myValue=0;
        reset() { myValue = 0; }
        add(anObject) { myValue = myValue + anObject.yourCost(); }
        remove(anObject) { myValue = myValue - anObject.yourCost(); }
}
```

Table 6.2: Pseudo-code for RunningTotal with mutable objects.

The solution in section 6.4.1 still correctly handles mutation of the collection itself. The system can add() and remove() instances of FixedCost and Professor to its heart's content without causing any problems. However, the previous solution did not allow the possibility that the objects stored in the collection might themselves mutate.

Suppose, for example, a Professor instance was "terminated" which involved both removing the instance from the collection, and setting the instance's salary to zero. The call to the collection's remove() method would cause RunningTotal to enqueue a delta message which will eventually cause it to run its own remove() method using a reference to the Professor instance as the argument. While that message is sitting in RunningTotal's queue, the Professor instance's salary variable is set to zero. When RunningTotal's remove() method is eventually called, it is going to subtract zero from myValue. Presumably the Professor was paid some positive amount when added to the collection, so the remove() call does not reverse the add() as it should.

A solution for this set of assumptions is rather simple, and is based on the other thing that happens when the Professor instance is mutated. Setting salary to zero also causes *IveChanged* messages to be sent to all its dependents. This instance of RunningTotal was added as a dependent of the Professor instance when Professor's yourCost() method was called by the RunningTotal add() method. The *IveChanged* message generated by setting salary to zero did not originate in a delder-collection's reset(), add(), or remove() method, so it was not treated specially. Instead the *IveChanged* message can be used to invalidate this delder value's cache. Thus when this delder is next accessed, it can be recomputed "from scratch" by first calling its reset() method and flushing its cache of any old delta messages, and then by calling add() for each element then stored in the collection.

The algorithm given in chapter 2 can be extended rather easily to support delder values. The delder collection's reset(), add() and remove() methods are handled specially as described in section 6.4.1. For all other derivers, the chapter 2 algorithm is followed with the following modifications. A new unique time-stamp-version-number (described in section 2.1.1) is now created whenever the delder's reset() method is called. Upon receipt of a current *IveChanged* message as described in section 2.1.2, the delder is marked as invalid, and when it is next computed its reset() method will be called, any old enqueued delta messages will be flushed without being processed, and the delder will be recomputed from scratch.

6.5.1 Summary of this algorithm

In summary, now the objects stored in the delder's collection **may** mutate. Mutations of the delder's collection itself are still handled the same way as in section 6.4.1. Thus the delder's collection may only mutate via its own reset(), add() and remove() methods. These methods will be treated specially. They will send any delder dependents *delta messages* which the delder objects will enqueue. Before responding to an access request, the delder object will process any enqueued delta messages by calling its own corresponding reset(), add(), or remove() method. Other mutations of the delder's derivers **will** generate *IveChanged* messages. The delder will handle a valid *IveChanged* message by invalidating its cached value. When the delder is next computed, it will flush its queue of any delta messages without processing them, it will call its own reset() method, it will generate a new unique time-stamp-version-number, and it will recompute its value from scratch by calling its add() method with each element of the collection.

This section extended the simple algorithm presented in 6.4.1 by incorporating my system's existing mutation monitoring system, and thus incurred minimal additional design and runtime overhead. However, whenever any object stored in the collection mutates, this solution does unfortunately discard all previous work, and recompute the delder from scratch. Fortunately, when objects in the underlying collection rarely mutate, this solution nicely enforces correctness with minimal overhead. Any implementation of my base system should seriously consider implementing this (section 6.5.1) algorithm. Some of the following sections discuss much more aggressive solutions which unfortunately incur considerably more overhead.

6.6 Many mutating objects

For relatively small collections, which must only rarely handle mutations, section 6.5.1 made a reasonable trade-off between constant overhead and the occasional additional overhead of recomputing from scratch. This section will attempt to handle a more extreme case. Suppose my section 1.3.1 assumption that the simple case is the common case remains true, but that for some collection mutations are also common. An example of that would be a very large collection. A collection so large that it will not fit into main memory might have a very small percentage of mutating elements. However, for such a large collection, even a low per element mutation rate would translate into a very high collective mutation rate. With such a large collection, recomputing from scratch is obviously prohibitive. The following sections will develop algorithms which never or only rarely require recomputing from scratch once the delder has been initialized.

6.6.1 Proxies

It is often more convenient to deal with only one element of the collection at a time. So just as section 6.1 broke up a large derived value into smaller derived values, let us begin by breaking up the delder's computation into many smaller derived values. The support objects described in this subsection will be used by all of the running-total algorithms developed in later sections. For this task we will use another special form of derived value; we will call this value a proxy. The "Proxy" design pattern is described in [GHJV95] in far more detail, though not directly applied to derived values. Where previously we would add objects directly to the delder's collection, now we will instead encapsulate the objects in a new instance of a proxy, and add the proxy to the collection. The basic behavior of a proxy is to receive a message, pass that message unchanged to the object it encapsulates, then return to its caller any value the encapsulated object returns. That basic behavior will be enhanced in a number of different ways to help implement the algorithms which follow.

```
class Proxy {
        myObject;
        delders[];

        Proxy(objectToEncapsulate, delderDependents) {
                myObject = objectToEncapsulate;
                delders = delderDependents;
                SendDeltaAddMessagesTo(delders);
        }
        reset() {
                delders = null;
        }
        remove() {
                SendDeltaRemoveMessagesTo(delders);
        }
        Object perform(methodName, args) {
                return(myObject.perform(methodName, args));
        }
}
```

Table 6.3: Pseudo-code for simple Proxy object.

The first enhancement is that the proxy objects will keep track of their delder dependents as the proxies are added and removed from the collection. Where previously upon a reset(), add(), or remove() operation, the collection sent a delta message to the delder, now the collection will call the proxy. For add() and remove(), the proxy will send the delta messages to the delder. For reset(), the collection must now call reset() for each proxy element, then call the delder's reset() method, and cause the delder to flush its queue of delta messages without processing them. This indirection through the proxies is because delta messages are not processed immediately. However, these direct method calls by the collection to the proxies are processed immediately.

Without this immediate processing, some race conditions would exist in the following algorithms. The pseudo-code listed in table 6.3 may make some of this somewhat clearer.

When an object is added to a collection, the collection creates a new proxy instance, and adds that new instance to the collection instead of the object nominally being added. As part of that creation process, the proxy learns what object is must encapsulate, and what delder values it will be serving. Once the proxy has recorded its delder dependents, it sends them the delta message to add, with the proxy itself as the argument to add. Upon reset() being called, the proxy changes its internal state so that in the future it will not propagate change messages to any delder dependents. Upon removal of this proxy from the delder collection, the remove() method is called, and the proxy sends a remove() delta message to its delder dependents.

Finally, the proxy's "perform" message is listed.[4] In Smalltalk, the "perform" message is implemented in Object, thus all objects in a Smalltalk system understand the "perform" message. Not all languages provide such flexibility. An alternative implementation is to know what messages will be sent to the encapsulated object, and write an explicit forwarder for each message. Thus, in place of "perform" my system might from the delder declarations generate code similar to the pseudo-code shown in table 6.4.

```
ReturnTypeA MethodNameA(a1, a2, ..., an) {
     return(myObject.MethodNameA(a1, a2, ..., an));
}
ReturnTypeB MethodNameB(a1, a2, ..., an) {
     return(myObject.MethodNameB(a1, a2, ..., an));
}
ReturnTypeC MethodNameC(a1, a2, ..., an) {
     return(myObject.MethodNameC(a1, a2, ..., an));
}
```

Table 6.4: Pseudo-code alternative to the "perform" method.

The algorithms which follow will further enhance the functionality of proxies beyond these basic traits.

[4] A proxy needs to accept an arbitrary message, and pass that message on to the object it encapsulates. In Smalltalk, the perform message takes as arguments, a method name and a list of arguments. It then executes that method name using the list of arguments as the argument list. The Java 1.1 equivalent in the invoke() method in class java.lang.reflect.Method.

6.7 Restricting the programmer to non-mutating methods

The algorithm described in section 6.5.1 placed very few constraints on the programmer. The programmer had to implement reset(), add(anObject), and remove(anObject), but that was about it. My system could handle any problems encountered by recomputing from scratch; it could even switch to an entirely uncached lazy evaluation strategy when necessary.

My system must still handle the possibility that "anObject" will mutate, but now it may not simply reset the delder and recompute from scratch. Such a mutation could be handled by first calling remove(anObject), then performing the mutation, and finally calling add(anObject). However, for my framework that is trickier than it at first appears. The problem is detecting where to place the remove() call, and ensuring that the remove() call is in fact "safe" in the sense that it does not alter the behavior of the user's other code.

Let S1 be the set of memory locations accessed in computing a function f(). Let S2 be the set S1 minus the location written, but not read, to hold the return value of f(), and minus any temporary storage which is allocated, written before being read, and also deallocated during the computation of f(). If all of the memory locations in S2 are unchanged after the computation of f(), then a second invocation of f() will return the same result. In fact, N invocations of f() will return the same result N times. Let us call a function f() with this property a non-mutating method. If we impose on the programmer the requirement that within their add(anObject) and remove(anObject) methods they only call the same non-mutating method with anObject, then both add() and remove() will share the same set S2 of derivers.[5] Thus, assuming remove() is the inverse of add(), N consecutive invocations of the sequence add(anObject), remove(anObject) would be a very complex operation which did not produce any visible results. Provided that none of the memory locations in the set S2 were changed, we could also insert otherwise basically arbitrary code g() between calls to remove(anObject) and add(anObject) in the sequence, still without having the sequence produce any visible results.

My system already annotates the user's code so that *IveChanged* messages are sent whenever an object mutates. Thanks to the atomicity rules described in section 3.1.7, if such messages were sent immediately before

[5] Assuming the add() and remove() methods themselves otherwise only access local variables, and the delder's "myValue" instance variable.

the mutation, instead of after the mutation, the system as presented to this point would still function correctly. We know that during the first call to add(anObject), dependency links will be created between all of the memory locations in the set S2, and the derived value being computed. If the *IveChanged* message is sent just before a mutation, the first such *IveChanged* message sent will be sent while the set S2 is still unchanged. If the set S2 is unchanged, and if the object was in a consistent state when add(anObject) was called, then the object was still in a consistent state after the call to add(). Since the object has not yet mutated since the call to add(), the object must still appear to be in a consistent state.[6] Thus the atomicity constraints are satisfied. If remove(anObject) is not going to change the set S2, then it is safe to insert the call to remove(anObject) immediately before the mutation, as it will not make any visible change to the behavior of the user's code.

Combining that, when we can impose the strict restrictions that the delder's add(anObject) and remove(anObject) methods must only call the same non-mutating method for anObject, and that remove() must also be the inverse of the add() method, then we may treat an object mutation as a remove() followed by an add() of that object. To treat a mutation as a remove() followed by an add(), the algorithm will need to determine which element in the collection is mutating. This can be done by using the proxy objects to isolate the delder from having direct derived-deriver dependency links with the mutating objects. Section 2.1.1 described how when a derived value (DV1) computation accesses another derived value (DV2), the links made while computing DV2 are made between the derivers and DV2, without any direct links back to DV1. Similarly, here the delder will have links to the proxies, but only indirectly through the proxies to the other derivers. The proxies themselves will have the direct links to the derivers. Proxy behavior must differ from normal derived value behavior in that, when a proxy receives its first valid *IveChanged* message (which was sent and is being processed just **before** the deriver mutates), the proxy will immediately invoke the delder's remove() method with the proxy itself as the argument, and will also send the delder a delta message which the delder will enqueue. At some later time, complying with the section 3.1.7 atomicity rules, the delder will process that delta message by calling its add() method. We needed to use the proxy objects in this case because we needed to use

[6]Technically, it is possible the object is not in a consistent state, because some instance variable our computation will not access has changed. However, our computation will in that case still perceive the object as being in its previous consistent state. As the whole point here is to reproduce the "old value" for the object, that is sufficient.

a different time-stamp-version-number for every item in the collection. The first time an item in the collection gets ready to mutate, the *IveChanged* message it sends will be valid because it matches the proxy's time-stamp-version-number. Later *IveChanged* messages with that same time-stamp-version-number will be ignored. Eventually the delder's add() method will cause the proxy to generate a new time-stamp-version-number which will be used to create new dependency links between the proxy and its derivers.[7] For this algorithm, the proxy's special call of the delder's remove() method does not need to create any dependency links. Through all of this, the delder may actually retain the same time-stamp-version-number, because the proxy itself will make certain that it only sends the delder messages when necessary.

How this algorithm handles mutation has just been described. The initial reset() call is handled the same as in the section 6.5.1 algorithm. The initial add() call when an object is added to the delder's collection will now be handled via the proxies, as described in section 6.6.1.

One remaining problem area exists. The proxy might be removed from the collection and then its encapsulated object might mutate, or a mutation might occur quickly followed by removal from the collection. These cases can be a problem if the second remove() occurs before the corresponding add() delta message is processed. I solve these potential race conditions in part by having the collection send add() and remove() via the proxies, instead of directly to the delder values.

To handle the case where an element is first removed from the collection, and then mutates, the proxy's remove() method will be enhanced as shown in table 6.5. For this algorithm, it is safe to call the delder's remove() method anytime after its add() method has been called, but before the set S2 of memory locations has mutated. For this case, our assumption is that S2 has not yet mutated. So the system must determine whether the delder's add() method has been called yet. At this point we know that the delder must

[7] One possibility that has been skipped over is that a collection might have multiple delder values associated with it. Their different add(anObject) methods, might send the same or different messages to anObject. For example, two might ask for yourSalary(), while another asks for yourRetirementContribution(). The proxy objects should correctly forward both yourSalary() and yourRetirementContribution(). Proxy objects may use the same time-stamp-version-number for both, in which case obviously a mutation relevant to only one of the two computations will force every dependent to recompute. Alternatively, the proxy object may use a separate time-stamp-version-number for each method name it forwards, thus one time-stamp-version-number for yourSalary() even though it is called twice, and a separate time-stamp-version-number for yourRetirementContribution(). This might lead to greater efficiency, or might incur more *IveChanged* message overhead. Either implementation choice should produce "correct" results.

```
remove() {
   for (int i = 0; i < delders.length; i++) {
      boolean msgPurged;
      msgPurged = delders[i].PurgeYourQueueOfAddMessageFor(this);
      if (! msgPurged) delders[i].remove(this);
   }
}
```

Table 6.5: Pseudo-code for enhanced Proxy object remove() method.

have been sent a add() delta message for this proxy, because this proxy's constructor sent it. We also know that there are only two ways that delta message could have left the delder's queue. It could have been processed, in which case we may safely call the delder's remove() method directly. Alternatively, the delder could have been reset(), but in that case this proxy would have also been reset(), and thus have "forgotten" the delder. So the enhanced remove will first search the delder's queue for an unprocessed add() delta message with itself as the argument. If it finds such a message, it will purge the message from the delder's queue, and is finished processing that delder. If such a message is not found, then it will immediately call the delder's remove() method directly.[8] This is known to be safe by the same argument as used previously for calling the delder's remove() method directly to handle a mutation. When the remove() call is done, the proxy will be finished handling this delder, and will remove the delder from its set of dependents. The proxy's remove() method will repeat this procedure for all of the delder's in its set of dependents. As a side efficiency note, during the processing of the delder's remove() method, the system does not need to establish new derived-deriver dependency links.

The case where the element first mutates, and then is removed from the collection is essentially identical. The only real difference is now the check for an unprocessed add() delta message in the delder's queue is much more likely to succeed. For this case, the proxy has called the delder's remove() method directly to process the mutation, and has also sent the delder an add() delta message to finish processing the mutation. The collection's call of the proxy's remove() still has the exact same constraints as before, and

[8]To preserve encapsulation, this combination of searching the delder's queue, and then either purging an item from the queue, or invoking the delder's remove() method, would normally be implemented as a method the delder inherits from its superclass. However, that detail obviously does not alter the algorithm, while including such details usually makes the algorithm's description harder to follow. Therefore, I will not generally mention such details again.

thus the proxy remove() version described in the previous paragraph will also work for this case without further modification.

This algorithm as described so far has assumed that every object added to the delder's collection is a complex object potentially subject to mutation. However, that violates my section 1.3.1 assumption that the simple case is the common case. The important enhancement that makes this algorithm much more reasonable, is that when the collection is adding an element, it can notice if the object being added is a primitive-literal type. If it is, the collection may simply add the value directly instead of encapsulating it inside of a new proxy object. Obviously, when the collection uses this change, the collection must also send add() and remove() delta messages to the delder directly instead of via the nonexistent proxy. However, that is simply the algorithm presented in section 6.5.1. The cost of this change is that if a method call on one of these primitive-literal objects accesses a global variable, and that global variable mutates, the delder will receive an *IveChanged* message directly, without a proxy to shield it. That will result in a very ugly reset. The delder will need first to get all the proxies to remove it from their dependents collection, then it will need to reset itself as called for by the section 6.5.1 algorithm. Finally, it will need to recompute itself from scratch, which will involve adding itself back to every one of the proxy's dependents collection, as well as executing the add() operations themselves.

6.7.1 Summary of this algorithm

This algorithm extends the section 6.5.1 algorithm. It achieves greater efficiency by restricting the programmer's freedom in declaring the delder. This algorithm insists that the delder's add(anObject) and remove(anObject) methods must only call the same non-mutating method for anObject, and that remove() must also be the inverse of the add() method. With these new restrictions, this algorithm may treat an object mutation as a remove() followed by an add() of that object.

This algorithm starts with the section 6.5.1 algorithm. However, instead of storing objects directly in the delder's collection, now they are encapsulated in proxy objects as described in section 6.6.1. The proxy objects catch *IveChanged* messages immediately before a mutation occurs. Thus, by immediately calling the delder's remove(anObject) method, they can effectively remove the "old value" of anObject before its mutation. The proxy objects also send a delta message to handle the add() portion of treating a mutation as a remove(oldValue) add(newValue) pair. Finally, removal of an element is handled via the proxy, which either removes the corresponding unprocessed add() delta message from the delder's queue, or if such a message does not exist, it calls the delder's remove() method directly.

There is an optional version of this algorithm with lower storage overhead. When an instance of a primitive-literal type is added to the collection, the collection does not encapsulate it in a proxy, and simply uses the add() and remove() delta messages specified in the section 6.5.1 algorithm. This enhancement unfortunately raises the possibility of having to recompute the delder from scratch.

This algorithm provides a relatively low overhead way to handle arbitrary mutations while rarely needing to reset() and recompute the delder value from scratch. The only times it might need to call reset() are when the collection itself is reset(), or a global variable accessed via a primitive-literal object mutates. The second case may even be prevented, by not using the enhancement to handle literals. Unfortunately, that greatly increases the overhead required. Both versions of the algorithm impose a rather severe restriction on the delder value's implementation, in that the delder value's add(anObject) and remove(anObject) methods must only call a non-mutating "anObject" method. If the database language does not specifically mark what methods are non-mutating, either some very advanced inter-procedural analysis is required, or the system must trust that the user's code is correct.

Happily, the class of non-mutating methods is a rather important one. For example, it includes all arithmetic functions. On page 32 of [GE95] Avigdor Gal and Opher Etzion quietly limit themselves to *"Computational invariants define the derived data item as an* **arithmetic function** *of other data items."* Thus I am very pleased this algorithm will handle all the cases they report their system will handle.

6.8 Keeping "old values" around

The hardest problem for a running-total-style incremental algorithm is its need for access to the "old value" of an element to reverse the add() operation. The algorithm in section 6.4.1 solved this by insisting that values not mutate, so the "old value" was always the same as the "current value." The section 6.5.1 algorithm "solved" this by giving up and starting over from scratch whenever it needed an "old value" which was not easily available. Section 6.7.1 used a computational approach, and restricted the programmer to functions which would repeatedly report the same "old value" if run again before a mutation actually occurred. The one obvious approach not yet described is to simply keep the "old value" around.

Unfortunately, keeping the "old value" around has some severe problems. Its most trivial problem is that it incurs additional storage overhead. Keeping an "old value" around for each element of a collection is likely to require as much space as the original collection! For collections of immutable objects, simply duplicating the collection itself would keep all the "old values" around; although the section 6.4.1 algorithm showed that in that case we do not need to store the "old values."

The difficult cases all involve mutable objects. We know that simply storing the same references to the mutable objects as the original collection does is useless. Our "old value" reference would refer to the same mutated object as the original collection! So to keep an "old value" around will require making a copy of the value, not just the reference. However, how deep a copy should we make? We previously saw that a recursively immutable object was insufficient for the section 6.4.1 algorithm. By the same logic, a direct recursive "deepCopy()" of each element in the collection would not necessarily be sufficient. The deepCopy() operation also has the severe problem that it can potentially require cloning the entire database. That could greatly increase our storage overhead requirements! The definition of "effectively immutable" suggests that we must copy every value referenced during the add() operation for each element in the collection. However, that is effectively the entire set S2 of memory locations described for the section 6.7.1 algorithm. Clearly that option is also too expensive. I conclude that storing copies of the delder collection's elements will not provide the basis of a useful algorithm.

6.9 The "Memento-Cookie" algorithm

What we really need to save is enough information to invert the add() operation itself, not the state of the universe when add() was previously invoked. Having the system automatically infer that information would be very difficult; however, for the programmer implementing the delder, it might be a simple task. Other designers have faced similar problems, and one of the resulting solutions is the "Memento" design pattern. [GHJV95] describes the purpose of a Memento object as follows:

> "Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later."

A closely related idea is a "Cookie" which can be decoded by the creator of the "Cookie," but is an opaque object for the rest of the system. Though the overhead is somewhat prohibitive, these ideas combine into a fairly straight forward algorithm.

Previously, the programmer specified a delder by giving three methods, reset(), add(), and remove(). For this algorithm, the delder's add() method will also need to create a cookie. The programmer will not need to implement a remove() method, but will be required to implement an eatCookie(aCookie) method.[9] The eatCookie(aCookie) method takes as an argument a cookie which was created by a previous invocation of the add() method. It uses the data stored in that cookie to reverse the add() method's effects. The eatCookie(aCookie) method is called when the corresponding element in the collection mutates, or is removed from the collection.

This algorithm requires some location to store all the cookies. Since the cookies will be used when an element in the collection changes, I have somewhat arbitrarily decided to store them in the corresponding proxy instances. So like the section 6.7.1 algorithm, this algorithm will store proxy instances in its collection, instead of directly storing the objects the proxies will encapsulate. Also like the section 6.7.1 algorithm, this algorithm will have the delder collection call the reset() method of every proxy instance in the collection, and then the reset() method of every delder dependent when the collection's own reset() method is called. This prevents obsolete mutations from corrupting the delder values.

The delder's add() method will need to inform the the proxy instance of its cookie. One possible design would resemble:

```
add(aProxy) {
        theCost = aProxy.yourCost();
        myValue = myValue + theCost;
        cookie = new ThisDeldersCookieClass(theCost);
        aProxy.registerCookie(this,cookie);
}
```

It is important to note that the delder and its CookieClass will be responsible for ensuring that the cookie's information is not dependent upon a mutable instance which might be externally referenced by some outside object. The cookie must not mutate!

This algorithm again has the delder collection send the delder add() and remove() delta messages via the proxies, as described in section 6.6.1.

[9] Obviously an alternative naming scheme could reuse the remove() method name. I selected a different name simply to avoid any possibility of confusion.

Thanks to the immutable cookies, unlike the section 6.7.1 algorithm, this algorithm does not rush to process the delder's remove() method. It may simply send a delta message which will eventually be processed. It is however, important that the proxy itself update its state so that after sending the remove() delta message, the proxy will not send the same cookie to the delder again. For this algorithm, it will do that by associating a "null" with the the delder reference, where it would otherwise store the cookie.

Normally, when a proxy receives a valid *IveChanged* message, it will send two delta messages. The first message will invoke the `eatCookie(aCookie)` method. The second message will invoke the delder's `add(thisProxy)` method using this proxy instance as the argument. However, that will not happen if this instance of the proxy has already been removed from the collection.

However, there is still one problem case that must be specially handled. That is the case where an element mutates, triggering eatCookie() and add() delta messages, and then is removed from the collection before the add() message is processed. The proxy will detect when this occurs, because it will not have a cookie to send the delder. Instead it will call a delder support method my system will supply, which will find the add(aProxy) message in the delder's queue which corresponds with this proxy, and remove that message from the queue without letting the delder process it.

One optimization the programmer declaring the delder should note, if the object returned from the `aProxy.yourCost()` or equivalent call is in its own right an effectively immutable object, a reference to that object may be used directly as the cookie. The eatCookie() method may easily be defined so that such objects do not have to be further encapsulated in a special CookieClass.

6.9.1 Summary of this algorithm

Like the algorithm of section 6.7.1, this algorithm stores items in the delder collection encapsulated in proxy instances. Delders declared to use this algorithm, must create a "cookie" as part of their add() method, and during the add() method must register this cookie with the proxy instance. The delder collection handles reset() as it did for the previous algorithm. It also uses the proxy to send the delder's add() and eatCookie() delta messages, instead of sending them directly to the delder. When a proxy receives a message from the collection saying it is being removed, if it has a cookie to send, it sends the delder an eatCookie() delta message with that cookie, stores null where it stored the cookie, and changes its time-stamp-version-number. If it does not have a cookie to send, instead of sending the delta message, it removes the unprocessed add() delta message from the delder's queue. If the proxy is handling an *IveChanged* message, if follows the same procedure, but it also sends the delder a delta message which will cause it to invoke its `add(aProxy)` method later.

Because it needs to store the cookies, unlike the section 6.7.1 algorithm, this algorithm does always use proxies. While the programmer might sometimes decide to use primitive-literal types as the cookies, at this point, I, and thus my system, do not know when the programmer will make that choice. Thus, this algorithm must always be prepared to store the delder's cookies. Therefore, this algorithm will always encapsulate every item in its collection inside of an instance of a proxy.

6.10 Algorithm selection

This chapter has presented four different running-total-style incremental algorithms.

1. Section 6.4.1 presented an algorithm which permitted only effectively-immutable objects in the collection.

2. Section 6.5.1 presented an algorithm which started over from scratch whenever an object in the collection mutated.

3. The algorithm in section 6.7.1 restricted the programmer's freedom, and thus was able to handle mutations as a remove() followed by an add().

4. Finally, section 6.9.1 presented an algorithm which stored a Memento for each collection element, and thus again was able to handle mutations as a remove() followed by an add().

Unlike the work presented in the previous chapters, for these algorithms, it is possible to state some definite preferences among the choices now, before a user's application is even designed. As a result, I expect the programmer to provide delder code implementing exactly one of the algorithms. My system will not attempt to determine which choice is the best at runtime, because we can probably do a better job right now!

For example, the first algorithm should probably never be implemented. When the objects know they are immutable, the first algorithm is just a special case of the second algorithm, which should be used instead. Though effectively-immutable objects might in theory be sufficient, if the objects do not "know" they are immutable, my system does not have a compelling reason to trust that the programmer is correct in claiming that all the objects in a collection will be effectively immutable. The performance advantage simply does not in my judgment outweigh the increased risk of "incorrect" results.

The third algorithm has obvious advantages over the second algorithm. Unfortunately, the programmer would not always wish to, or even always be able to, comply with the third algorithm's restrictions. However, when the programmer can comply, it seems clear that the third algorithm is almost always the best choice among these four alternatives. My one caveat is that again the system must, absent extremely advanced inter-procedural analysis, trust the programmer. Permitting this option is certain to cause some very obscure bugs. On that basis, some systems should not permit it at all. However, if it is permitted at all, then it is clearly almost always the best performance choice of the four.

Finally, the fourth algorithm unfortunately incurs substantial additional overhead. However, I can easily imagine systems where the programmer would prefer to trade space and additional distributed cost, to avoid the possible worst case response time associated with recomputing from scratch. However, again this is a choice I think the programmer must make.

There is one alternative algorithm I have not really discussed. The system could use a purely lazy approach, and every time the delder is accessed, call reset(), and then add() for each element in the collection. If the rules from chapter 2 force a delder into "must be lazy" mode, then this alternative will in fact be mandatory. My system should still enforce the chapter 2 rules. However, when my system finds it necessary to force a delder into "must be lazy" mode, it should also generate an error message at its highest non-fatal priority. I expect that the system will almost always have detected an error in the programmer's application when "must be lazy" mode is forced.

There are certainly usage patterns where lazy mode will outperform all of these algorithms under the cost criteria from section 5.1. For example, if the delder is never accessed at all, lazy mode obviously wins! However, normally lazy mode loses because of disk access costs. Computing a delder from scratch requires bringing the entire collection, **including the objects it references,** into memory from disk. That is a very expensive proposition. Updating a delder because an object mutated, or was added or removed from the collection, will require bringing the delder itself into memory. However, for an object to mutate, it must already be in memory. For an object to be added or removed from a collection, at least that part of the collection must already be in memory, and frequently, the relevant object will also be in memory. Thus, comparing lazy evaluation with the algorithms described in this chapter will usually involve comparing many disk accesses with relatively few disk accesses. We do not need to instrument a working system to know which will usually be faster!

The final reason for not having my system dynamically consider switching to lazy evaluation mode is the difficulty and cost of monitoring a delder. Simply monitoring at the granularity of the delder would provide rather coarse results. However, monitoring at the collection element level requires significant storage overhead. It could easily require more storage to monitor at that level than the section 6.9.1 algorithm requires to store all the "Memento-Cookies!"

For these reasons, my system does not attempt to second guess the programmer, when the programmer selects one of the algorithms described in this chapter.

Chapter 7

Contributions, conclusions, and future work

7.1 Contributions

I developed an original algorithm for safely matching derived and deriver in behaviorally-object-oriented database systems. Using this algorithm as the foundation, I created a design that permits such databases to safely cache the results of computing derived values without relaxing their correctness criteria. The cache will always be invalidated before being accessed if any of the derived value's derivers have mutated. My system is the only work I know of addressing this problem for behaviorally-object-oriented systems. My system also supports a superset of the *"arithmetic functions"* which [GE95] Avigdor Gal and Opher Etzion report supporting for a structurally-object-oriented system.

Chapter 5 presents the *"intelligent system"* I designed, which uses my cost estimate formulas to dynamically make informed materialization choices for the derived data values in the system.

Chapter 6 presents algorithms I designed extending my basic system to support "delder values." "Delder values" permit the system to support derived values defined in terms of "deltas." A classic example of a "delder value" is the running-total of a collection.

7.2 Conclusions

We have seen that supporting caching alternatives for derived data involves significant overhead. With all that overhead, why should the system ever materialize derived data? Because sometimes materializing derived data is faster! The overhead only becomes horrible when the derivers are mutable objects. A number of cases exist where the system should win by materializing derived data.

When a derived value has a large set of derivers, the potential savings (partly a function of the number of derivers) is also large. If the vast majority of those derivers are simple immutable objects, then a small percentage of the derivers are mutable, and thus the dependency cost (partly a function of the number of mutable derivers) will be small relative to the possible savings. When the savings are large and the costs are low, the system wins by materializing.

If the derived value is computationally expensive to calculate, and thus deriver access makes up a minimal part of the expense, then materializing the derived value in the idle loop will usually be a good choice.

Many classic computer science techniques win because computer access patterns are not randomly distributed, but instead exhibit considerable locality of reference. It seems reasonable to expect that a derived value's set of derivers would also tend to exhibit locality-of-reference behavior. If most of the derivers tend to be mutated in the same transactions, then the overhead of using the dependency links is considerably lessened. The relevant data items would generally be loaded into memory only once, and stale-link garbage collection becomes almost a non-issue.

Closely related, though a derived value's derivers might be technically mutable objects, the actual database usage pattern might almost never mutate them. If the database usage pattern also frequently accessed the derived data item, then clearly materializing would be the correct choice.

I am not exploring "hard real-time systems" in this document. However, possible future work (see 7.3.4) includes the idea that pre-materializing a derived data item when system response constraints are flexible, might make it possible to later meet tighter critical response time constraints.

Clearly many cases remain where the system should not materialize a derived data item. In those cases, my system's goal is to inexpensively recognize that sticking with lazy evaluation is the correct choice. The primary goal of my design is to create a system which chooses among the available materialization strategies, and changes its choices when changes to the database state and usage warrant. Even if the system usually selects lazy evaluation, my design goal is still useful if the system's occasional optimizations save more user time than the system consumes.

Thus, while my system would definitely not be an appropriate extension for all databases, it should be useful:

- When there are many derivers, but only a small percentage are mutable.

- When the derived value is computationally very expensive to calculate.
- When derived and derivers exhibit considerable locality of reference.
- When the usage pattern frequently reads the derived value, but rarely mutates any derivers.

If the response time a user experiences is an important design criteria, and the application also exhibits some of the listed characteristics, then the system designers should consider incorporating my work in their system.

7.3 Future work

7.3.1 Building a prototype

This dissertation has presented a design for a database extension. The next logical step would be to build a *proof-of-concept* prototype. To be useful, this would probably require a fully operational object-oriented database as the underlying system, it would also require the ability to place the resulting system under realistic loads, and to measure the resulting performance.

7.3.2 Supporting "temporary" writes

In sections 4.3 and 5.3.3, to support the idea of caching a derived value in RAM without writing it to disk, I desired the ability to write an object, and then unwrite it before committing.

I think the generic solution is a procedure which expands and contracts an object upon being loaded and unloaded from memory. The system should be smart enough to only save objects which upon being contracted will have changed from their pre-expanded state.

Obviously this idea is still very preliminary, but the need to temporarily write objects was very real. Merely having separate non-persistent objects would not have solved my problem. This looks like a wonderful topic for further work.

7.3.3 Expand the cost evaluation algorithm

In 5.6.2 I described a local cost algorithm. However, I noted that a global cost estimate algorithm might be useful, and that I deferred that to future work. An entire dissertation could be written exploring enhancements to my cost evaluation algorithms. My existing algorithm, as previously stated, considers only local costs. One obvious enhancement would be to consider the combined costs when there are derived of derived trees. This approach would be considerably complicated by the possibility that separate trees might have some leaves in common. Doing the cost analysis quickly inside of a user transaction would also be complicated by the fact that often only some of the relevant derived values would have been invalidated by that transaction. Perhaps a solution would be to perform some of the cost analysis in the idle loop instead; I did not explore that possibility at all.

If a "global" cost evaluation strategy is desired, it becomes necessary to consider the costs both above and below particular derived values in the derived of derived trees. Though not an especially closely related topic, some of the problems appear to be similar to those presented and solved in Donald Knuth's classic paper [Knu68] on Attribute Grammars.

Another possible way to improve the system's cost estimates, would be to solicit more programmer input. Although my dissertation assumes that the programmer can not necessarily know in advance which materialization strategy is best, the programmer might still have some insights the system could use. Creating a way for the programmer to communicate those insights would make a nice project.

7.3.4 Integrate with "hard real-time constraints"

As I mentioned in 7.2, pre-materializing derived data items might make it possible to meet real-time constraints that would otherwise be impossible. This might be an interesting area for further work by someone interested in real-time systems.

7.3.5 Not always having everything deriver ready

I made the simplifying assumption in 5.4.8 that basically any object may become a deriver at any time. I further assumed that to handle this without huge headaches, all of the system's code was always enhanced with deriver-derived dependency checking code. Thus the code expansion costs and run-time checking costs were incurred regardless of the deriver/not-deriver status of the associated objects.

It might be interesting to explore a more dynamic technique of switching the code between deriver-supporting and lower-overhead, not-deriver-supporting versions. However, it is important to remember that we might be literally inside of an object's behavior code when the switch is made! The hardest case is probably when we have called out of a method, and not yet returned, but when we return the object's deriver status has changed!

The other factor likely to complicate this approach is that in most object-oriented systems, the code is normally associated with classes, not instances. Many classes will have both instances which are derivers, and instances which are not derivers. However, it might be useful to look into the possibilities in more depth.

What should be deriver ready?

A closely related question is what variables should be eligible to become derivers? Instead of having everything deriver ready, could the system have objects which are not deriver ready, and when accessed, instead of becoming derivers, such objects force their accessors into *"must be lazy"* mode?

7.3.6 Using "spare" disk space

As I mentioned in 4.5, it is potentially possible to store some of the derived data and associated dependency information to otherwise unused disk scratch space. The underlying idea is simply that in principle it is "always" possible to recompute the derived data. Thus only enough information to know if this recomputation is necessary must be stored to durable storage. The other information could be stored to unstable storage, such as disk space subject to unexpected reuse. I did not explore this idea in part because its benefits were very unclear. However, in theory, it obviously has some attractive aspects.

References

[Adi81] Michel Adiba. Derived relations: A unified mechanism for views, snapshots and distributed data. In *Proceedings of the International Conference on Very Large Data Bases*, volume seven, pages 293–305, September 1981.

[BDH+90] Jean-Pierre Briot, Norihisa Doi, Yasuaki Honda, Yuuji Ichisugi, Yasushi Kodama, Ichiro Ohsawa, Etsuya Shibayama, Toshihiro Takada, Takuo Watanabe, and Akinori Yonezawa. *ABCL: An Object-Oriented Concurrent System.* MIT Press series in computer systems. The MIT Press, 1990.

[BE96] David Botzer and Opher Etzion. Optimization of materialization strategies for derived data elements. *IEEE Transactions on Knowledge and Data Engineering*, 8(2):260–272, April 1996.

[BJ70] G. E. P. Box and G. M. Jenkins. *Time Series Analysis, Forecasting, and Control.* Holden-Day Inc., San Francisco, 1970.

[Cat93] R. G. G. Cattell. *Object Data Management: Object-Oriented and Extended Relational Database Systems.* Addison-Wesley, July 1993.

[Cha95] Sharma Chakravarthy. Early active database efforts: A capsule summary. *IEEE Transactions on Knowledge and Data Engineering*, 7(6):1008–1010, December 1995.

[Cho65] Wen M. Chow. Adaptive control of the exponential smoothing constant. *The Journal of Industrial Engineering*, XVI(5):314–317, September-October 1965.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* The MIT Press, 1990.

[DPG91] Oscar Díaz, Norman Paton, and Peter Gray. Rule management in object oriented databases: A uniform approach. In *Proceedings of* *the International Conference on Very Large Data Bases*, volume seventeen, pages 317–326, September 1991.

[Etz93a] Opher Etzion. Pardes-a data-driven oriented active database model. *ACM SIGMOD Record*, 22(1):7–14, March 1993.

[Etz93b] Opher Etzion. The reflective approach for data-driven rules. In M. Huhns, M. P. Papazoglou, and G. Schlageter, editors, *Proceedings of International Conference on Intelligent and Cooperative Information Systems (Cat. No.93TH0474-7)*, pages 350–358. IEEE, May 1993.

[Etz94] Opher Etzion. An alternative paradigm for active databases. In J. Widom and S. Chakravarthy, editors, *Proceedings Fourth International Workshop on Research Issues in Data Engineering. Active Database Systems (Cat. No.94TH0618-9)*, pages 39–45. IEEE, February 1994.

[For82] Charles L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982. ISSN 0004-3702.

[FRS93] Françoise Fabret, Mireille Regnier, and Eric Simon. An adaptive algorithm for incremental evaluation of production rules in databases. In *Proceedings of the International Conference on Very Large Data Bases*, volume nineteen, pages 455–466, Dublin, Ireland, August 1993.

[FT95] Piero Fraternali and Letizia Tanca. A structured approach to the definition of the semantics of active databases. *ACM Transactions on Database Systems*, 20(4):414–471, December 1995.

[GE95] Avigdor Gal and Opher Etzion. Maintaining data-driven rules in databases. *Computer*, 28(1):28–38, January 1995.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: elements of reusable object-oriented software.* Addison-Wesley, 1995.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* The Java Series. Addison-Wesley, 1996.

[GM95] Ashish Gupta and Inderpal Singh Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE*

Data Engineering Bulletin, Special Issue on Materialized Views and Data Warehousing, 18(2), June 1995.

[Gol84] Adele Goldberg. *Smalltalk-80: The Interactive Programming Environment.* Addison-Wesley, 1984.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[Gra87] Jim Gray. The 5 minute rule for trading memory for disk accesses and the 10 byte rule for trading memory for CPU time. In *Proceedings of ACM Special Interest Group on Management of Data*, volume 16(3) of *ACM SIGMOD Record*, pages 395–398, December 1987.

[Knu68] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, June 1968. Correction: *Mathematical Systems Theory,* 5(1):95-96, March 1971.

[KP81] Shaye Koenig and Robert Paige. A transformational framework for the automatic control of derived data. In *Proceedings of the International Conference on Very Large Data Bases*, volume seven, pages 306–318, September 1981.

[Luc87] John M. Lucassen. Types and effects: Towards the integration of functional and imperative programming. Technical Report MIT/LCS/TR-408, MIT, 545 Technology Square, Cambridge, Massachusetts 02139, August 1987. Tech report version of his PhD. thesis.

[Mir87] D.P. Miranker. Treat: A better match algorithm for AI production systems. In *Proceedings of the National Conference on Artificial Intelligence*, Seattle, Washington, 1987.

[MJ76] Douglas C. Montgomery and Lynwood A. Johnson. *Forecasting and Time Series Analysis.* McGraw-Hill Book Company, 1976.

[MP90] C. Bauzer Medeiros and P. Pfeffer. A mechanism for managing rules in an object-oriented database. Technical report, Altair, 1990.

[Oll78] T. William Olle. *The CODASYL Approach to Data Base Management.* John Wiley & Sons, 1978.

[QW91] Xiaolei Qian and Gio Wiederhold. Incremental recomputation of active relational expressions. *IEEE Transactions on Knowledge and Data Engineering*, 3(3):337–341, September 1991.

[RCK+95] Nick Roussopoulos, Chungmin M. Chen, Stephen Kelly, Alex Delis, and Yannis Papakonstantinou. The ADMS project: Views "R" us. *IEEE Data Engineering Bulletin, Special Issue on Materialized Views and Data Warehousing*, 18(2):19–28, June 1995.

[RSS96] Kenneth A. Ross, Divesh Srivastava, and A. Sudarshan. Materialized view maintenance and integrity constraint checking: Trading space for time. In *SIGMOD RECORD: Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, volume 25(2), pages 447–458, June 1996.

[RW81] Enders A. Robinson and Herman Wold. Prediction and forecasting. In Enders A. Robinson, editor, *Time Series Analysis and Applications*, chapter 3, pages 45–82. Goose Pond Press, Houston, 1981.

[SR88] Jaideep Srivastava and Doron Rotem. Analytical modeling of materialized view maintenance. In *Proceedings of the Seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 126–134, March 1988.

[SZ91] Arie Segev and J. Leon Zhao. Data management for large rule systems. In *Proceedings of the International Conference on Very Large Data Bases*, volume seventeen, pages 297–307, September 1991.

[Tec89] Xerox Advanced Information Technologies. Hipac: A research project in active, time-constrained database management. Final Technical Report XAIT-89-02, Xerox, July 1989.

[Tod76] S. Todd. The Peterlee relational test vehicle – a system overview. *IBM Systems Journal*, 15(4):285–307, 1976.

Vita

William Eric Voss is a high school drop-out. In 1988 he received a B.A. degree with high honors from the University of Tennessee – Knoxville, with a dual major in Computer Science, and College Scholars *"with emphasis in Computer Science research"* and a minor in Mathematics. In partial fulfillment of the requirements for a "College Scholars" major, he wrote a simulation in Smalltalk-80 to evaluate voting algorithms for distributed commit protocols in the presence of partitioning because of network failures.

In 1994 he received an M.S. degree from the University of Illinois at Urbana-Champaign in Computer Science. His M.S. thesis described a simple single user relational database he implemented to support an early WWW site.