Generating Language-Based Editors: A Relationally-Attributed Approach

Susan B. Horwitz

TR 85-696
August 1985
(Ph. D. Thesis)

Department of Computer Science
Cornell University
Ithaca, NY 14853
GENERATING LANGUAGE-BASED EDITORS:
A RELATIONALLY-ATTRIBUTED APPROACH

A Thesis
Presented to the Faculty of the Graduate School
of Cornell University
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

by
Susan Beth Horwitz
August, 1985
Susan Beth Horwitz was born on January 6, 1955, in Berkeley, California, but spent most of her life in the more pleasant climate of Syracuse, New York. She received a Bachelor of Arts degree in Ethnomusicology from Wesleyan University in 1977, and a Master of Science degree in Computer Science from Cornell University in 1982. She has accepted a position as Assistant Professor with the Department of Computer Sciences at the University of Wisconsin - Madison, and is looking forward to experiencing life in the mid-west.
GENERATING LANGUAGE-BASED EDITORS:
A RELATIONALLY-ATTRIBUTED APPROACH

Susan Beth Horwitz, Ph.D.
Cornell University 1985

The basic premise of this thesis is that an editing environment should include a database of information about a program, and a facility for performing computations on that information. Many features traditionally provided in an ad hoc fashion by a collection of loosely-connected tools can be supported in a uniform fashion as computations on a database of program information.

The goal of the research was to define a language-independent model of editing that includes the maintenance of a database of program information and supports database computations, thus allowing editing environments to be generated rather than built ad hoc. A second goal was to design the algorithms needed to support interactive editing under the defined model with an acceptable degree of efficiency.

In this model, a program is represented as a consistently attributed abstract-syntax tree with an associated relational database. Relations can be functionally dependent on the state of the attributed tree, or on the values in other relations. An editing operation is a subtree derivation or a subtree replacement; after an editing operation, attribute values and functionally dependent relations are made consistent with the new state of the tree.
The major contributions of the thesis are the following:

(1) The demonstration of the versatility of the simple concept of performing computations on a relational database of program information. It is shown how such computations can support static-semantic checking, anomaly detection, an interrogation facility, and the ability to define alternate views of a program.

(2) The definition of implicit relations, relations whose values are computed as needed rather than being stored explicitly in their entirety, and the design of a new method for query evaluation. The query-evaluation method, which is vital to the efficient evaluation of queries using implicit relations, can provide significant time and space savings in the evaluation of general relational queries.

(3) The design of an incremental update algorithm for relational views. Views can be updated after a single change to the database, or after a series of changes. The update algorithm can be tuned according to the relative importance of time and space consumption.
**Table of Contents**

Chapter 1: Introduction ........................................................................................................ 1

1.1 Specifying language-based editors: The procedural *versus* the declarative approach .................................................. 2

1.2 A language-independent model of editing environments ........................................... 3

1.3 Overview of the thesis ................................................................................................. 5

Chapter 2: Background ...................................................................................................... 7

2.1 Attribute grammars ................................................................................................... 7

2.2 The relational database model .................................................................................... 8

Chapter 3: The use of relations in an editing environment ............................................. 11

3.1 Locally-derived relations ......................................................................................... 11

3.2 Unconstrained relations .......................................................................................... 13

3.3 Dataflow relations .................................................................................................... 14

3.4 Queries and views ...................................................................................................... 15

Chapter 4: Combining relations and attributes ............................................................... 21

4.1 Limitations of the relational model .......................................................................... 21

4.2 Adding attribute grammars ..................................................................................... 23

4.3 Relations and attributes: a symbiosis ...................................................................... 28
Chapter 1

Introduction

Language-based editing environments can enhance the programming process by preventing and detecting errors during program entry and by helping the programmer to understand an existing, perhaps very large, program. This thesis proposes a model for language-based editing environments that includes a relational database of information about a program and facilities for performing computations on that information. There are two basic advantages to this model:

(1) Versatility: Many features traditionally provided in an ad hoc fashion by a collection of loosely-connected tools can be supported in a uniform fashion as computations on a database of program information.

(2) Language-independence: A language-independent model of editing environments permits the design of an editor generator. Given the large number of programming languages currently in use, generating language-based editing environments from language specifications is a far more appealing prospect than writing each such environment entirely by hand.

The particular editing environment model proposed in this thesis is described in more detail in section 1.2, following a discussion and comparison of the two basic approaches to the design of an editor generator's specification language.
1.1. Specifying language-based editors: The procedural versus the declarative approach

There are two basic approaches to the specification of context conditions in language-based editors: procedural and declarative. Using the procedural approach, the editor designer writes a collection of *action routines*, procedures that are called in response to user input. These procedures examine the program, and may modify global data structures. GANDALF [Medina-Mora and Notkin 1981] is an example of a generator that uses the procedural approach; an editor designer using this system writes procedures to be called after cursor motion, subtree insertion, subtree deletion, etc.

Using the declarative approach, the editor designer writes equations that define the state of the editing environment as a function of the state* of the program. The Synthesizer Generator [Reps and Teitelbaum 1984] is an example of a generator that uses the declarative approach; the editing environment is specified using an attribute grammar. The basic difference between the two approaches is that the declarative approach only defines what the state of the environment must be, without defining how it should be computed.

The uniformity of declarative specifications makes them, in general, easier to understand and to modify than procedural specifications. While it may seem that the procedural approach is better-suited to handling special cases

*Remember that we are discussing editing environments; 'state of the program' refers to the current form of the program, rather than to its execution state.
efficiently, proponents of the declarative approach regard this as a challenge rather than as a defeat, hoping to achieve the same level of efficiency by applying optimization techniques. In addition, there is an important problem that can arise using the procedural approach but not using the declarative approach: an editor defined by an erroneous procedural specification can exhibit order-dependent errors. Using such an editor, it is possible to arrive at environment state $S$ by constructing a program using one sequence of operations, and to arrive at a different environment state $S'$ by constructing the same program using a different sequence of operations. For example, after building a program top-down, the programmer might be told that the program contains an error, while after building the same program bottom-up, he is told that the program is error-free! Such order-dependent errors in the editor specification can be very difficult to track down.

The considerations discussed above have led us to the use of a declarative rather than a procedural approach to specifying editing environments. This choice has influenced our design of a model of editing environments, as discussed in the next section.

1.2. A language-independent model of editing environments

The model of editing environments proposed in this thesis includes a relational database of information about a program and facilities for performing computations on that information. The versatility of this simple model is demonstrated with respect to the goals of language-based editors given above: detecting errors
during program entry and helping the programmer understand existing programs. Unfortunately, the limited computational power of the relational operators restricts the power of the model as a whole. Furthermore, the disappointing results of [Linton 1984] indicate that a purely relational approach in which the program itself is represented as a collection of relations, is impractical, at least using current database technology.

While adding new operators would solve some problems, it would simultaneously introduce new ones: termination of queries might no longer be guaranteed, and the efficiency of query evaluation and view updating would undoubtedly decrease. Further, relying on the presence of non-standard operators would mean giving up the possibility of using most if not all existing relational database systems.

In view of these limitations of the pure relational model, and because of previous positive experience with attribute grammars [Reps 1984] [Reps and Teitelbaum 1984], we have chosen to combine the two formalisms, using attribute grammars to support and augment the relational model. We have found that relations and attribute grammars form a symbiotic relationship as a basis for editing environments, the strengths of one filling in for the weaknesses of the other. Programs are represented as attributed abstract-syntax trees with an associated relational database. The tuples of relations, like attributes, are defined declaratively on a per-production basis; thus, a relation is an aggregation
of information that would otherwise be scattered throughout the program tree. Tuples can contain tokens, program points, and attribute values. Computations that are outside the scope of the relational operators can be performed in the attributes, and the results incorporated in tuples. The relations so defined are kept up to date by virtue of the incremental attribute updating algorithm of [Reps 1984] and are available as arguments to relational queries and views. Whereas in the pure attribute grammar formalism an attribute can depend only on its neighbors in the tree, a hybrid system introduces the possibility of allowing attributes to depend on information extracted from global relations, provided this introduces no circular definitions.

1.3. Overview of the thesis

Brief reviews of attribute grammars and the relational database model are given in chapter 2. The remainder of the thesis is organized into three parts: the first part, chapters 3 and 4, describes our language-independent model of editing. We begin in chapter 3 with a purely relational model, showing how relations can be used to provide static-semantic checking, anomaly detection, an interrogation facility, and alternative views of a program. Chapter 4 gives examples of important information that cannot be computed in a purely relational model and discusses the advantages of combining relations and attributes. A simple example program is introduced in chapter 3 and is used as a running example throughout the thesis.
The second part, chapters 5, 6, and 7, presents new algorithms for query evaluation and incremental view updating motivated by efficiency requirements for relationally-attributed-grammar-based editors.

The third part, chapter 8, describes our prototype implementation of an editor generator and discusses problems that might arise in future, more complete implementations. Possible solutions are suggested for some of these problems; others are left as areas for further research.
Chapter 2

Background

2.1. Attribute grammars

An attribute grammar [Knuth 1968] is a context-free grammar extended by attaching zero or more attributes to each nonterminal. There are two kinds of attributes: synthesized attributes, which pass information up the abstract-syntactic tree, and inherited attributes, which pass information down the abstract-syntactic tree or between siblings in the tree. For each production of the grammar, semantic equations are given defining the synthesized attributes of the left-hand-side nonterminal and the inherited attributes of the right-hand-side nonterminal(s). An attribute is defined as a function of the other attributes of the production.

A non-circular attribute grammar is one from which it is impossible to derive an abstract-syntactic tree in which the value of an attribute depends on itself. All uses of the term ‘attribute grammar’ in this thesis should be understood to mean ‘non-circular attribute grammar’.

The usefulness of attribute grammars as the basis for editing environments is demonstrated in [Reps 1984] and [Reps and Teitelbaum 1984]. The notation used in these two works is followed here: ‘X.a’ is used to denote attribute ‘a’ of nonterminal ‘X’; when there are multiple occurrences of nonterminal ‘X’ in a
production, ‘$X$.a’ denotes attribute ‘a’ of the leftmost such occurrence, ‘$X$.a’
denotes attribute ‘a’ of the second-to-leftmost occurrence of nonterminal ‘$X$’,
and so on.

2.2. The relational database model

The relational database model was first introduced in [Codd 1970]. A relation is
defined as follows: given a collection of sets $S_1, S_2, \ldots, S_n$, not necessarily dis-
tinct, $R$ is a relation on these sets if it is a set of ordered $n$-tuples $<s_1, s_2, \ldots,$
$s_n>$, such that $s_1 \in S_1, s_2 \in S_2, \ldots, s_n \in S_n$. Sets $S_1, S_2, \ldots, S_n$ are the
domains of $R$; $n$ is the arity of $R$.

It is convenient to think of a relation as a table. Each row represents one
tuple; a relation with arity $n$ has $n$ columns. The columns are sometimes called
attributes, however, for obvious reasons, we will not use that terminology and
will refer to the columns of a relation as fields. A relation is denoted by its
name and the names of its fields, which are suggestive of its domains; for exam-
ple, a relation containing information about the collection of programs in an
editing environment might be denoted by:

PROGRAM_INFORMATION(program name, author, last modification date).

The individual fields of a relation can be denoted either by position or by name;
for example, PROGRAM_INFORMATION.field #2 denotes the same field as
PROGRAM_INFORMATION.author.
A *query* is an expression in the relational algebra in which relational operators are applied to argument relations to produce a relational result. The result of a query is entered into the database as a new relation for future reference.

The operators considered in this thesis include the usual set operators: UNION, INTERSECT, MINUS, and CROSS PRODUCT, as well as the special relational operators: SELECT, PROJECT, JOIN, and IS IN. SELECT is a unary operator that selects the subset of its operand relation's tuples that satisfy a given condition. The condition can have any of the following forms:

\[
\text{condition} \rightarrow \text{condition} \text{ "AND" condition} \\
\quad \mid \text{condition} \text{ "OR" condition} \\
\quad \mid \text{single-clause}
\]

\[
\text{single-clause} \rightarrow \text{field relop constant} \\
\quad \mid \text{field relop field}
\]

\[
\text{relop} \rightarrow = | \neq | < | \text{etc. } ;
\]

PROJECT is a unary operator that selects only those columns of its operand relation that are in a given list of fields. The PROJECT operator is sometimes denoted by the Greek letter 'pi' subscripted with the given list of fields, for example:

\[
\Pi_{\#1,\#4}(R)
\]

denotes the projection of the first and fourth fields of relation R.

JOIN is a binary operator that is a shorthand for CROSS PRODUCT followed by SELECT according to a given condition. The condition of a JOIN can
have any of the following forms:

\[
\text{condition} \rightarrow \text{condition} \text{ "AND" condition} \\
\quad \text{condition} \text{ "OR" condition} \\
\quad \text{field-of-left-operand relop field-of-right-operand} \\
\]

\[
\text{relop} \rightarrow = | \neq | < | \text{etc.} \\n\]

When the only \text{relop} in a condition is \text{"=",} the JOIN is called an EQUI-JOIN. For the purposes of the algorithms presented in chapters 6 and 7, EQUI-JOIN is the most interesting form of JOIN. EQUI-JOIN is treated in depth in these chapters; the treatment of the other forms of JOIN follows from the treatment of CROSS PRODUCT and SELECT.

IS IN is a binary operator that selects those tuples of its left-hand operand relation in which the value of a given field is an element of its right-hand operand relation. The right-hand operand must be a relation of arity 1. A weaker version of IS IN is sometimes seen in the literature as a possible form of a SELECT operator's condition:

\[
\text{condition} \rightarrow \text{field IS IN list-of-values};
\]

The more general IS IN used in this thesis does not increase the power of the relational operators, but simplifies the expression of some queries.

A \text{view} is a query result that is, conceptually, recomputed after every change to an operand of the query. In fact, in most systems, views are only computed as needed.
Chapter 3

The use of relations in an editing environment

In our model, an editing environment includes a relational database of information about the program being edited and facilities for performing computations on that information. The database includes two kinds of "base" relations, *locally-derived relations* and *unconstrained relations*, and two computational facilities, *dataflow analysis* and *relational operations*.

Dataflow analysis makes use of the information in locally-derived relations to produce dataflow relations. The relational operators can be applied in the form of queries or view definitions to any existing relations, producing new unconstrained or view relations.

This chapter discusses the relations and computational facilities of our model and illustrates how they can be used to provide static-semantic checking, anomaly detection, an interrogation facility, and the ability to define alternative views of a program.

3.1. Locally-derived relations

The contents of a locally-derived relation is an immediate function of the current state of the program being edited. As the program is modified, locally-derived relations are changed accordingly. Locally-derived relations are so named because their tuples can only contain information local to an editing unit.
Some examples of locally-derived relations that might be defined in an editing environment are:

\texttt{PROC\_DECL( program point, proc name, \# of params )}

\texttt{PROC\_CALL( program point, called proc, \# of args )}

\texttt{DEFINITION( program point, variable name )}

\texttt{USE( program point, variable name )}

The presence of the tuple \langle p, \text{foo}, 1 \rangle in the \texttt{PROC\_DECL} relation means that procedure \text{foo} is declared at program point \texttt{p} and has one parameter. Figure 3.1a shows a simple example program (with deliberate errors); figure 3.1b gives the corresponding contents of the \texttt{PROC\_DECL}, \texttt{PROC\_CALL}, \texttt{DEFINITION}, and \texttt{USE} relations. It is convenient in these examples to represent program points as line numbers; in an actual system, program points would be values that are unaffected by editing operations elsewhere in the program, for example, tree-node addresses.

(1) \texttt{PROCEDURE main( )}
(2) \texttt{	PRINT( sum( ) )}
(3) \texttt{}
(4) \texttt{PROCEDURE sum(x, y, z) }
(5) \texttt{	a := x + y}
(6) \texttt{	RETURN( a )}
(7) \texttt{}

\textbf{Figure 3.1a}

Example program
Locally-derived relations can contain information about a program's structure as well as its contents. For example, a PARENT relation can be defined to represent the structure of the program tree, and a CONTROL_FLOW relation can be defined to represent the program's control-flow graph.

A potentially useful relation is the binary CALLS relation:

CALLS( procedure name, procedure name )

The presence of the tuple \(<P1, P2>\) in the CALLS relation means that procedure \(P1\) contains a call to procedure \(P2\). Defining the CALLS relation as a locally-derived relation requires making the name of the calling procedure available in the editing unit that contains a procedure call.

### 3.2. Unconstrained relations

Unconstrained relations, unlike locally-derived relations, are not functionally dependent on the program state. Unconstrained relations can be used for
documentation, describing the organization and history of system components:

PROC_INFO( proc name, author, last modification date, use)

They can also be used to record the history of a program execution, for debugging or profiling [Snodgrass 1984]:

EXECUTION( proc name, start time, stop time)

as a notepad facility:

TO_DO( program point, comment )

and to store the results of query executions.

3.3. Dataflow relations

Global dataflow analysis assigns to each program point a set of program facts, information that will be true at that point in every possible execution. The nature of the program facts depends on what kind of dataflow analysis is done. After live variable analysis, for example, the facts at a program point are the names of variables for which there is a control-flow path from that point to the end of the program on which the variable is used before being defined.

Global dataflow analysis can be performed using the information available in locally-derived relations. Live variable analysis, for example, would use the information in the CONTROL_FLOW, DEFINITION, and USE relations. Dataflow relations reflect the results of global dataflow analysis. The dataflow relation for live variable analysis is: LIVE( program point, variable name ).
Figure 3.2 gives the contents of the LIVE relation for the program of figure 3.1a:

\[
\text{LIVE: } \langle (5), a \rangle \\
\langle (4), x \rangle \\
\langle (4), y \rangle
\]

Figure 3.2
LIVE relation associated with example program

3.4. Queries and views

The facilities for defining and executing queries, which are provided as part of a relational database management system, can be used to support an interrogation facility similar to that provided by MASTERSCOPE [Masinter 1980]. Queries can be written to pose questions such as:

(1) What procedures call procedure P?

(2) Where is variable x defined?

(3) Where is variable y used?

The answers to queries are themselves relations; thus, the user is able to compute with query answers. For example, the intersection of the answers to queries (2) and (3) above is the set of program points where variable x is defined as a function of variable y.

Because relations can include information about the definitions and uses of variables, queries might be formulated to assist with tasks like replacing multiple similar blocks of code with procedure calls and the appropriate procedure body, deciding whether the order of consecutive statements can be reversed without
changing the semantics of the program, and determining the scope of a proposed program modification.

A view definition is a query that is, conceptually, reexecuted after every change to the underlying database; a view relation, or simply, a view, is the result of the most recent such execution. A view is thus a function of the relations used in the view definition. Locally-derived, dataflow, unconstrained, and view relations can all be used to define a view provided there is no circularity in the definition; a view cannot directly or transitively depend on itself.

View relations can be used to define alternative views of a program. The user may wish to limit his view of a program to include only procedures for which some predicate is satisfied, for example procedures that call procedure P, procedures written by Jones, or procedures used to implement hash tables. The appropriate view definition is easily written using the sample relation CALLS or PROC_INFO to select the names of the relevant procedures, and the sample relation PROC_DECL to match names to program points.

Graphical views of a program, as in [Reiss 1984], can be provided by alternative schemes for displaying relations. For example, graphical display of the CALLS relation provides a call-graph view of a program.

Given the appropriate locally-derived and dataflow relations, it is possible to define views that indicate the presence of anomalies [Osterweil and Fosdick 1976] [Gillette 1977] [Freudenberger 1984] in a program. An anomaly is something that, although correct in terms of syntax and static semantics, indicates a
possible logical error. Anomalies that can be defined as views include:

- definitions of values that are never used
- uses of variables that may be unitialized
- declared variables that are never used
- procedures that are never called
- jumps to jumps
- modification of an input parameter

For example, the view relation indicating the presence of the first anomaly:

```
UNUSED_DEFINITION(program point, variable name)
```

can be defined using the locally-derived relation DEFINITION and the dataflow relation LIVE. The set of used definitions, that is, the set of definitions of variables that are live at the point of definition, is subtracted from the set of all definitions: DEFINITION − LIVE. Figure 3.3 shows this computation for the DEFINITION and LIVE relations given in figures 3.1b and 3.2; the resulting tuple, \(<(4), z>\), shows that, at line 4 of the example program, variable \(z\) is given a value that is never used.
Views can also indicate the presence of static-semantic errors such as:

- uses of undeclared variables
- calls to undeclared procedures
- multiply declared variables
- calls to procedures with the wrong number of arguments

For example:

`WRONG_#_ARGS(program point, called proc, correct # args)`

can be defined using the locally-derived relations `PROC_DECL` and `PROC_CALL` by taking the cross product of these relations, selecting those tuples in which the name of the declared procedure matches the name of the called procedure but the number of parameters does not match the number of arguments, and then projecting out the point of the call, the name of the called procedure, and the number of parameters. Figure 3.4 shows this computation.
for the PROC_DECL and PROC_CALL relations given in figure 3.1b; the resulting tuple, < (2), sum, 3 >, shows that, at line 2 of the example program, there is a call to procedure 'sum' that should have three arguments.

![Diagram](image)

Figure 3.4
Definition tree for the WRONG_#_ARGS view relation

Figure 3.5 gives a snapshot of an editing session where our example program and the two views UNUSED_DEFINITION and WRONG_#_ARGS are displayed in windows on the screen.
By defining anomaly and static-semantic error detection as views, the user is given complete control over feedback about anomalies and static-semantic errors. The user can keep views for which he wants immediate feedback on the screen while he is editing his program and can display other views when he is ready to see them.

Of course, any view displayed on the screen during program editing must be updated whenever an editing operation causes a change to a relation used in the view definition. Recomputing views from scratch every time would certainly be too slow; incremental view updating is needed. This problem is addressed in chapter 7.
4.1. Limitations of the relational model

Chapter 3 illustrated the versatility of the concept of computations on program information under the relational model. Unfortunately, the limited computational power of the relational operators precludes performing some useful computations. Three kinds of queries whose definitions are outside the scope of the relational operators are:

- queries that require transitive closure
- queries that require order-dependent processing
- queries that require arithmetic processing

It is shown in [Aho and Ullman 1979] that transitive closure of arbitrary binary relations cannot be defined without the addition of a non-standard relational operator. Transitive closure is needed to determine whether one program point is the ancestor of another in terms of the program’s hierarchical structure. This ANCESTOR relation is a very useful one. The CALLS relation introduced in chapter 3, for example, can only be defined as a locally-derived relation if the name of the calling procedure is somehow made available at the point of the call. By contrast, given a transitive closure operator, the CALLS relation could be defined as a view using the PROC_DECL, PROC_CALL, and ANCESTOR relations.
Order-dependent processing is needed for type-checking. The type of a node in an expression tree cannot be determined until the types of its children are known. Although for every expression tree $T$ it is possible to write a query $Q_T$ such that $Q_T(T) = type(T)$, without order-dependent processing it is not possible to write a single query $Q$ such that for all expression trees $T$ (of arbitrary height), $Q(T) = type(T)$.

Order-dependent processing is also required for computations on variables when the programming language in use is scoped. Identifiers alone are not sufficient to name variables in a scoped language, and because the information in inner scopes takes precedence over the information in outer scopes, transitive closure is insufficient; order-dependent processing must be used.

Arithmetic operators are usually excluded from relational query languages in accordance with the philosophy that information retrieval should be kept separate from arithmetic computation. This precludes the detection of some interesting static-semantic errors and anomalies. Arithmetic processing is needed to exploit the results of constant propagation to evaluate expressions that are used as loop or IF statement conditions, array subscripts, and denominators. Evaluating these expressions would permit the detection of conditions that are always true or always false, subscripts out of bounds, and division by zero.
4.2. Adding attribute grammars

New relational operators could be added to provide transitive closure and order-dependent and arithmetic processing [Chandra 1981]. This solution would require handling problems including possible non-termination of queries and decreased efficiency of query evaluation and view updating.

Another possibility would be to use Prolog [Closkin and Mellish 1981] in place of a relational database. Although this is an intriguing idea, the absence of any incremental recomputation in current versions of Prolog make this approach impractical.

A third possibility is to augment the relational approach with some other formalism. In our case, the availability of the Synthesizer Generator [Reps and Teitelbaum 1984], a working generator of attribute-grammar-based editors, made the second possibility the more attractive one. By building on top of the Synthesizer Generator we were spared many low-level design decisions and a great deal of coding, thus allowing us to pursue the more interesting aspects of the research. Further, as discussed in section 4.3, we were aware of some shortcomings of the pure attribute grammar approach and were interested in seeing how combining attributes and relations might solve some of these problems.

We have chosen to adopt the editing paradigm of the Synthesizer Generator as well as its use of attributes: programs are represented as consistently-attributed abstract-syntax trees with a current cursor position; an editing
operation either moves the cursor or performs a subtree replacement at the point of the cursor. Subtree replacement can cause attributes at the root of the subtree to become inconsistent; the incremental attribute-updating algorithm of [Reps 1984] is used to reestablish consistency throughout the program tree.

Representing programs as trees rather than as relations has helped us to overcome some of the efficiency problems suffered in pure relational systems such as in [Linton 1984]. In Linton’s system, using the standard INGRES relational database management system [Stonebraker et al 1976], the display of a ten-line procedure body requires forty seconds of elapsed time. Even using a version of INGRES specially tuned for this task, the display requires seven seconds. By contrast, display of procedures in editors generated by the Synthesizer Generator is virtually instantaneous. Further advantages of representing programs as trees are discussed in chapter 5.

We call the combination of relations and attribute grammars a relationally attributed grammar, defined as follows:

A relationally attributed grammar is an attribute grammar augmented in the following ways:

(1) A grammar production may have attached to it zero or more tuple membership assertions. A tuple membership assertion is of the form:

$$\text{tuple} \in \text{relation}$$

“tuple” is specified by giving expressions for the values of each of its fields.
An expression can use the attribute values, program points, and tokens available at the production. "relation" is specified by giving a relation name. The meaning of attaching a tuple membership assertion to production \( p \) is that for every instance \( p_i \) of production \( p \) in the program tree there is a tuple in the named relation whose field values are the values of the given expressions evaluated in the context of \( p_i \).

Tuple membership assertions can also be given conditionally: \textbf{if} condition \textbf{then} tuple membership assertion. In this case, instance \( p_i \) contributes a tuple to the named relation only if \textit{condition} is true in the context of \( p_i \).

(2) The semantic equations for an attribute may include relational operations on named relations, provided this introduces no circular dependencies.

Tuple membership assertions provide a mechanism both for specifying the contents of locally-derived relations and for keeping them up-to-date at edit time. The value defined by a tuple membership assertion is, from the internal system's point of view, an attribute, since it depends only on values local to the production. This means that the incremental attribute-updating algorithm of [Reps 1984] can be used to keep locally-derived relations up-to-date as a program is edited; individual tuples are modified to reflect changes to the attributes, program points, or tokens on which they depend. (A notational device similar to tuple membership assertions is used in [Knuth 1968] to specify the
insertion and extraction of elements from a set-valued attribute of the root of the abstract-syntax tree.)

The amount of information that can be included in a tuple of a locally-derived relation is increased by allowing the tuple's defining expressions to include attribute values. Recall that a tuple of a locally-derived relation can only contain information local to an editing unit. Attributes can be used to bring information from arbitrarily far away in the program tree to the editing unit in which a locally-derived relation's tuple is defined. For example, attributes can be used to extract the name of each procedure from its declaration and to pass the procedure name through the procedure body. In this way, the name of the calling procedure is made available at each call statement; the procedure-name attribute is used as the first field of the locally-derived CALLS relation. Figures 4.1a - 4.1c illustrate this: figure 4.1a gives a context-free grammar fragment; figure 4.1b adds the declaration of the procedure name attribute and its defining semantic equations; figure 4.1c adds the declaration of the CALLS relation and its tuple membership assertion.

\[
\text{procedure} \rightarrow \text{IDENT} \ "(" \text{parameter_list} \ "\)" \ "\{" \text{stmt} \ "\}
\]
\[
\text{stmt} \rightarrow \text{stmt} \ "\;\;\;\" \text{stmt} \\
| \ "\text{CALL}" \ \text{IDENT} \ "\(" \text{argument_list} \ "\)"
\]

Figure 4.1a
Context-free grammar fragment
proc_name: inherited STRING attribute of stmt;

procedure -> IDENT "(" parameter_list ")" "{" stmt "}"  
{ stmt.proc_name = IDENT; }

stmt -> stmt ";" stmt  
{ stmt$2.proc_name = stmt$1.proc_name;  
 stmt$3.proc_name = stmt$1.proc_name; }

| "CALL" IDENT "(" argument_list ")"

Figure 4.1b
Attributed grammar fragment

locally-derived relation CALLS("call from" STRING, "call to" STRING );
proc_name: inherited STRING attribute of stmt;

procedure -> IDENT "(" parameter_list ")" "{" stmt "}"  
{ stmt.proc_name = IDENT; }

stmt -> stmt ";" stmt  
{ stmt$2.proc_name = stmt$1.proc_name;  
 stmt$3.proc_name = stmt$1.proc_name; }

| "CALL" IDENT "(" argument_list ")"  
{ <stmt.proc_name, IDENT> ∈ CALLS; }

Figure 4.1c
Relationally attributed grammar fragment

Just as attribute values can be used to construct relations, it is reasonable to allow values from relations as inputs to semantic equations, provided there is no possibility of circularity. For example, to compute the type of an expression one must know the types of the variables that appear in the expression. This information can be extracted from the relation: VAR_DECL(var name, type). Figure 4.3 gives a grammar fragment in which the "type" attribute of the "exp"
nonterminal is defined using the VARDECL relation.

\[
\text{exp} \rightarrow \text{IDENT} \\
\quad \{ \text{exp.type} = \text{Project(type)} \\
\quad \quad \text{from Select(var name = IDENT)} \\
\quad \quad \text{from VARDECL;}
\}
\]

Figure 4.3
Using relations to define attributes

Chapter 8, section 8.2.1, discusses the practical implications of allowing attribute values to depend on relations.

4.3. Relations and attributes: a symbiosis

While the use of attribute grammars augments the relational model, the use of relations also enhances the editing environment provided by a pure attribute-grammar-based system. One weakness of attribute grammars is that information cannot jump from one point in the program tree to another; it must be threaded through the tree structure. In the absence of this potentially costly threading, information remains scattered throughout the program tree. This can have an adverse effect on user feedback. For example, a program modification such as changing a declaration can cause statements throughout the program to become erroneous. If this information is not available at the declaration, the user may be unaware that he has introduced errors into his program. Further, because there is no way to group related attributes, the user cannot decide to receive immediate notification about some kinds of errors and not others.
Under the relational model, by contrast, a separate view can be defined for each kind of error. The user can decide which views to display, and because all displayed views are updated in response to program modifications, the user is aware of the consequences of each editing operation.

This question of grouping versus passing information through the tree has implications in terms of the cost of an editing operation as well as in terms of user feedback. Under the pure attribute model, a symbol table can be built up in the declarations part of a program and then passed down throughout the statements part of the program. Each instance of a variable use involves a symbol table lookup to insure, for example, that the variable is declared, or to determine its type. Changing the declaration of any variable changes the symbol table attribute, and all variable lookups must be redone. The cost of changing a declaration is thus proportional to the number of variable uses in the scope of the declaration.

Under the relational model, the symbol table can be maintained as a relation, with views defined to check for undeclared variables or to extract the types of variables. A change to a variable declaration causes a change to the symbol table relation, which triggers incremental updating of the appropriate views. Whereas an attribute value is recomputed whenever an argument attribute changes, without taking into account the exact nature of the change, view updating is driven by the particular change to the changed defining relation. The cost of updating a view is proportional to the sum of the sizes of the
changes to the defining relations, the changes to the intermediate relations defined in the view-definition tree, and the change to the view itself. This may well prove to be a smaller cost than the cost of attribute updating.

Solutions to the problem of updating aggregate-valued attributes are also being sought within the pure attribute-grammar framework. Some examples of this effort are: [Johnson 1983] [Demers et al 1985] [Reps et al 1985] [Beshers and Campbell 1985]. The goal of these proposals is to update only those attributes that depend on the changed elements of an aggregate-valued attribute, ignoring attributes that depend on unchanged elements of the aggregate. This is similar to incremental updating of views in which the only operation is selection.
Chapter 5

Implicit relations and a new method for query evaluation

Chapter 4 mentioned some of the advantages of storing programs as trees rather than as tuples of relations: operations like displaying a procedure body or moving the editing cursor are more straightforward, and thus faster, given a tree representation. This chapter presents a further advantage of storing programs as trees: structural information implicit in the tree can be used in relational computations. This is not possible, however, given standard query-evaluation techniques. A new query-evaluation method is introduced and outlined in this chapter; an in-depth presentation is given in chapter 6. Although the new query-evaluation method was inspired by the needs of relationally-attributed-grammar-based editors, it is applicable to general relational queries. The advantage of this new method is that it avoids computing intermediate values during query evaluation, and may thus provide substantial time and space savings over more traditional methods.

5.1. Implicit relations

Some information about a program’s structure can be captured in locally-derived relations, for example, the PARENT relation introduced in section 3.1. The transitive closure of the PARENT relation, however, can neither be defined as a locally-derived relation nor can it be computed from the PARENT relation using the standard relational operators. Fortunately, given the representation of
programs as abstract-syntax trees, it is unnecessary to explicitly build either the PARENT or the ANCESTOR relations; this information is implicitly available in the structure of the program tree. Another example of a relation implicitly available in the program tree is the relation of program points and the language construct at that point. *Implicit relations* such as these can be predefined:

\[
\begin{align*}
PARENT(&parent, child) \\
ANCESTOR(&ancestor, descendant) \\
CONSTRUCT(&program point, language construct)
\end{align*}
\]

and are available for use in queries and views. A query to extract all definitions of variable \( x \) made inside procedure \( P \) illustrates the use of implicit relations. This query can be expressed using the implicit relation ANCESTOR and the locally-derived relation DEFINITION: the set of program points at which \( x \) is defined is intersected with the set of program points in procedure \( P \); the program point of the root of procedure \( P \) is denoted by ‘@\( P \)’. A query tree is given in figure 5.1.

An interesting technical problem is how to evaluate queries that contain implicit relations, since explicitly building such relations would obviously be too costly. Our solution relies on the use of two operations, membership testing and selective retrieval. While conceptually performed on the relations, these operations can actually be performed on the program tree.
5.2. Membership testing and selective retrieval

Definition:

1. A membership-test procedure for relation R is a procedure that, given a tuple t, returns 'true' if t is in R, and otherwise returns 'false'.

2. A selective-retrieval procedure for relation R is a procedure that, given a list of fields f₁, f₂, ..., fᵌ, and a list of values v₁, v₂, ..., vᵌ, returns the set of tuples in R that have value vᵢ in field fᵢ for all i.

In the following sections we show how membership-test and selective-retrieval procedures can be implemented for the relations named at the leaves of a query tree, both implicit and explicit relations. Examples are then given showing how
membership-test and selective-retrieval procedures can be used during query evaluation and how the use of these procedures can provide substantial time and space savings over more traditional query-evaluation methods. These examples motivate the need to provide membership-test and selective-retrieval procedures for the relations represented by the internal nodes of a query tree as well as for the relations named at the leaves of the query tree.

Section 5.6 gives a few representative examples of the membership-test and selective-retrieval procedures that are built for the internal nodes of a query tree; some of the problems and solutions associated with this process are presented.

The purpose of the following sections is to give the reader an overview of query evaluation using membership-test and selective-retrieval procedures, without getting bogged down in details. Having read this chapter, the reader can either go on to chapter 6, which gives an in-depth presentation of the method, or skip ahead to chapter 7.

5.3. Membership-test and selective-retrieval procedures for leaf relations

We have said that our new query-evaluation method relies on the use of membership testing and selective retrieval. In this section we show how procedures for these operations are implemented for both implicit and explicit relations, and we discuss the costs associated with the procedures.
Appropriate cost measures as well as the costs of the membership-test and selective-retrieval procedures themselves will vary from one implementation to another. The query evaluation method presented here is best suited to systems in which the use of CPU time and of temporary storage space are of greatest concern; this is the case, for example, with in-core database systems. It is these factors, therefore, that we emphasize below and in chapter 6 where we discuss the costs of membership-test and selective-retrieval procedures for the internal nodes of a query tree.

5.3.1. Membership-test and selective-retrieval procedures for implicit relations

Membership-test and selective-retrieval procedures for implicit relations operate on the program tree. For example, consider the two implicit relations ANCESTOR and CONSTRUCT, defined above in section 5.1. The most straightforward way to implement membership testing for the ANCESTOR relation is the following: given a potential ancestor-descendant pair \(<a, d>\), start at node \(d\) and walk up the program tree to the root; if node \(a\) is found on this path return 'true', otherwise return 'false'. The time cost of a membership-test procedure implemented in this way is, in the worst case, proportional to the height of the program tree. Amortized cost \(O(\log(\text{size of program tree}))\) can be achieved using the method of [Tarjan 1983]. This method requires that a self-balancing tree representation of the program be maintained; unless this representation can be used by the editing environment in place of the usual
program tree, this improved time cost is gained at the expense of the space needed to store the self-balancing tree.

The membership-test procedure for the CONSTRUCT relation simply verifies whether the given language construct is indeed used at the given program point; thus, an invocation of this membership-test procedure requires a constant amount of time and has no space overhead.

The selective-retrieval procedure for the ANCESTOR relation, given a value \( d \) for the descendant field, walks up the program tree to the root, building a tuple \(<n, d>\) for each node \( n \) along the path. Given a value \( a \) for the ancestor field, the procedure traverses the subtree rooted at \( a \), building a tuple \(<a, n>\) for each node \( n \) in the subtree. Both the time and space costs of selective retrieval for the ANCESTOR relation are thus proportional to the size of the answer set. The selective-retrieval procedure for the CONSTRUCT relation, given a value \( p \) for the program point field, returns the tuple \(<p, \text{language construct used at } p>\), with constant time and space costs. Given a value \( c \) for the language construct field, the procedure will, in the worst case, traverse the entire program tree looking for occurrences of construct \( c \). This process can be improved by maintaining auxiliary information, for example, for each language construct \( c \), a list of the language constructs that can enclose \( c \). Thus, knowing that a statement can occur inside a while loop, but not inside a declaration, the selective-retrieval procedure looking for occurrences of the construct 'statement' will traverse only potentially productive subtrees [Donzeau-Gouge et al 1980]. In
the worst case, the time cost of selective-retrieval for the CONSTRUCT relation, given a value for the construct field, is $O(\text{size of program})$; the space cost is $O(\text{size of answer set})$, plus any overhead required to maintain auxiliary information.

5.3.2. Membership-test and selective-retrieval procedures for explicit relations

Membership-test and selective-retrieval procedures can also be written for explicit relations, relations stored as sets of tuples. The efficiency of the membership-test and selective-retrieval procedures associated with explicit relations depends on the storage and access methods provided for the relations. In this section we consider three sample methods:

1. the relation has an associated hash table for field $f$
2. the relation is stored sorted on field $f$
3. the relation is stored unsorted with no auxiliary access method.

Method 1 provides the fastest procedures for membership testing and for selective retrieval when the list of fields includes field $f$, at the expense of the space needed for the hash table. If field $f$ is the key field, a membership test requires a constant amount of time for hashing and to compare the non-key fields with the given values. Selective retrieval requires a constant amount of time for hashing, and, if values are given for fields in addition to field $f$, for comparing the given values to the values actually in the tuple.
If field $f$ is not the key field, a membership test requires a constant amount of time for hashing, plus time proportional to the number of tuples with the given value in field $f$ to search for the given tuple. Selective retrieval requires a constant amount of time for hashing, plus time proportional to the number of tuples with the given value in field $f$ to find all tuples whose field values match the given list of values.

Method 2 is the same as method 1 once the first tuple with the given value in field $f$ is found; assuming that the relation is stored so that an efficient search technique can be used, this method adds $O(\log(\text{size of relation}))$ time to membership testing and selective retrieval.

Method 3 illustrates the worst case, requiring a complete scan of the relation for a membership test or selective retrieval; thus, the cost of one of these operations is $O(\text{size of relation})$.

5.4. Using membership-test and selective-retrieval procedures

Some relational operators can use an operand relation's membership-test or selective-retrieval procedure in place of an explicit representation of the operand relation. INTERSECTION is one such operator. Given a membership-test procedure for either relation R1 or relation R2 and an explicit representation of the other relation, one can implement $(R1 \cap R2)$ as follows: a single scan is made through the explicit operand relation considering each tuple $t$ in turn; $t$ is in the result relation if it is a member of the other operand relation.
let
    R1 and ANSWER be relations
    R2.membership be a membership-test procedure
    for relation R2
in
begin
    ANSWER := {};
    for each tuple t in R1 do
        if (R2.membership(t))
            then ANSWER := ANSWER ∪ {t};
    return(ANSWER);
end

Figure 5.2
Implementing INTERSECTION using a membership-test procedure

Similarly, EQUI-JOIN can be implemented using one explicit operand relation and the other operand relation's selective-retrieval procedure. A single scan is made through the explicit operand relation; for each tuple t, the selective-retrieval procedure of the other operand relation is called with the name of the join field and the appropriate value from t. All returned tuples are joined with t and added to the result relation. The code given below in figure 5.3 is for the operation: JOIN (R1, R2) WHEN (field #3 of R1) = (field #1 of R2).
let
R1 and ANSWER be relations
R2.selective-retrieval be a selective-retrieval procedure
for relation R2

in
begin
  ANSWER := {};
  for each tuple t in R1 do
    for each tuple u in R2.selective-retrieval((1), (t.field #3)) do
      ANSWER := ANSWER \ JOIN(t, u);
  return(ANSWER);
end

Figure 5.3
Implementing EQUI-JOIN
using a selective-retrieval procedure

While using a relation’s membership-test or selective-retrieval procedure in place of an explicit representation of the relation is necessary for efficient evaluation of queries containing implicit relations, this evaluation technique is also applicable to queries without implicit relations. In the examples of figures 5.2 and 5.3, relation R2 is never used directly. When R2 is an intermediate relation in a query tree, R2.membership and R2.selective-retrieval can often be implemented so that neither R2 nor any of the intermediate relations involved in the computation of R2 need to be built; instead, the membership-test or selective-retrieval procedures associated with the nodes of the subtree rooted at R2 are called. Since the computation of an intermediate relation can be arbitrarily complex, it is clear that the use of efficient membership-test and selective-retrieval procedures can significantly reduce both the time and space requirements of query evaluation.
5.5. Algorithm outline

Given a query tree, and given membership-test and selective-retrieval procedures for the relations named at the leaves of the tree, the algorithm for query evaluation using membership-test and selective-retrieval procedures is the following:

(1) Make one post-order traversal of the query tree. At each node \( n \) build procedures \( n.\text{membership} \), \( n.\text{selective-retrieval} \), and \( n.\text{relation} \); \( n.\text{relation} \) is a relation-producing procedure, a procedure that explicitly builds the relation represented by node \( n \). Also, compute an estimate of the cost of invoking each of these procedures.

(2) Call the relation-producing procedure at the root node of the query tree.

Procedures at node \( n \) can call any procedure at a child of \( n \) in the query tree. For example, the relation-producing procedure at the root of the query tree of figure 5.1 might call its left child’s relation-producing procedure and its right child’s membership-test procedure.

The cost estimate for a procedure at node \( n \) is based on the cost estimates for the procedures it calls. When there are several possible ways to implement a procedure, cost estimates are used to choose the most efficient implementation.

The implementation of membership-test and selective-retrieval procedures for leaf relations was discussed in section 5.3; the next section gives several representative examples of the membership-test and selective-retrieval procedures built for internal nodes of a query tree. Some of the potential problems and solutions associated with this process are discussed.
5.6. Membership-test and selective-retrieval procedures for internal nodes

One could implement the membership-test and selective-retrieval procedures for an internal node of a query tree by building the intermediate relation represented by the node and searching for the appropriate tuples. Since the point of using membership-test and selective-retrieval procedures is to avoid building intermediate relations, this strategy will rarely be cost-effective. In general, membership-test and selective-retrieval procedures make use of similar procedures found at the next level down in the query tree, which in turn make use of procedures at the next level down, and so on, until we reach the leaves of the tree and the procedures associated with the leaf relations.

The particular membership-test or selective-retrieval procedure built for an internal node of a query tree depends on which relational operator is found at that node. Sometimes there are several ways to implement the membership-test or selective-retrieval procedure. The selective-retrieval procedure for an INTERSECTION node \( n \), with operands \( R_1 \) and \( R_2 \), is a good example; one could implement \( n.\text{selective-retrieval} \) in any of the following ways:

1. Given parameters \( f \) and \( v \), \( n.\text{selective-retrieval} \) returns the intersection of \( R_1.\text{selective-retrieval}(f, v) \) and \( R_2.\text{selective-retrieval}(f, v) \).

2. For every tuple \( t \) returned by \( R_1.\text{selective-retrieval}(f, v) \), \( n.\text{selective-retrieval} \) calls \( R_2.\text{membership}(t) \); \( t \) is in the set returned by \( n.\text{selective-retrieval} \) iff \( R_2.\text{membership} \) returns 'true'.

(3) Same as (2), reversing the roles of R1 and R2.

For each instance of an INTERSECTION node, the implementation of its selective-retrieval procedure depends on the costs of its children’s membership-test and selective-retrieval procedures.

In the absence of PROJECT nodes, the design of membership-test procedures for internal nodes of a query tree is quite straightforward. For example, the membership-test procedure for an INTERSECTION node \( n \) with operands R1 and R2, given parameter \( t \), returns the logical \textbf{and} of \( \text{R1.membership}(t) \) and \( \text{R2.membership}(t) \):

\[
\text{n.membership}(t) \{
\text{if} (\text{R1.membership}(t))
\text{then return}(\text{R2.membership}(t));
\text{else return}(\text{false});
\}
\]

PROJECT nodes complicate membership-test procedures because the arity of the relation represented by a PROJECT node is smaller than the arity of its operand relation. We want to build a membership-test procedure \( p \) for the PROJECT node that invokes its child’s membership-test procedure \( p' \); however, \( p \) cannot pass to \( p' \) values for fields that are not included in the PROJECT node’s list of projected fields. We allow \( p \) to pass a special “wildcard” value, denoted by ‘\( * \)’, to \( p' \) for every field not included in the list of projected fields. Membership-test procedures must thus be prepared to handle tuples that contain wildcard values.
In some cases, SELECT nodes can replace wildcard values with constant values. This situation can be illustrated by considering the example query of figure 5.1. The root node of this query tree is an INTERSECTION node. The relation-producing procedure for this node can be implemented by calling the relation-producing procedure of its left child, and the membership-test procedure of its right child.

The membership-test procedure of the right-hand-side PROJECT node is called by the relation-producing procedure of the INTERSECTION node with a tuple of the form \(<d>\), because the arity of the relation represented by both the INTERSECTION node and the PROJECT node is 1.

The PROJECT node’s membership-test procedure calls the membership-test procedure of the right-hand-side SELECT node with a tuple of the form \(<*, d>\), because the arity of the relation represented by the SELECT node is 2. The wildcard value is in the first field of the tuple because the projected field, ‘descendant’, is the second field of the relation represented by the SELECT node.

The SELECT node’s membership-test procedure calls the membership-test procedure of the ANCESTOR relation with a tuple of the form \(<@P, d>\). The SELECT node’s membership-test procedure is able to replace the wildcard value in the first field with a constant value because a tuple cannot be in the relation represented by the SELECT node unless it has the value @P in the first field.
The above example illustrates the least perturbance that can be caused by the presence of wildcard values; the SELECT node simply replaces the wildcard value with a constant and calls its child's membership-test procedure. In some cases, more drastic action is required. For example, in the presence of wildcard values, the membership-test procedure for an INTERSECTION node cannot simply return the conjunction of its children's membership-test procedures. Figure 5.4 illustrates a situation in which doing so would lead to erroneous results.

If the INTERSECTION node's membership-test procedure is called with a tuple of the form <*, 1> and calls its children's membership-test procedures with the same tuple, they will each return 'true'. R1.membership returns 'true' because it includes tuple <0, 1>, and R2.membership returns true because it includes tuple <1, 1>, yet there is no tuple in the relation represented by the INTERSECTION node with a 1 in the second field.

When there are wildcard values in the given tuple $t$, the INTERSECTION node's membership-test procedure must be implemented in one of the following ways:

**Implementation method 1**

Step 1:
Call one child's selective-retrieval procedure, passing as arguments the list of fields in $t$ that do not have wildcard values and the corresponding list of values.
Step 2:
For every tuple \( u \) returned by Step 1, call the other child’s membership-test procedure with argument \( u \). As soon as the membership-test procedure returns ‘true’, the membership-test procedure for the INTERSECTION node returns ‘true’. If the membership-test procedure never returns ‘true’, the membership-test procedure for the INTERSECTION node returns ‘false’.

**Implementation method 2**

Step 1:
Call both children’s selective-retrieval procedures, passing as arguments the list of fields in \( t \) that do not have wildcard values and the corresponding list of values.

Step 2:
Compute the intersection of the two sets returned by step 1. If the intersection is non-empty, the INTERSECTION node’s membership-test procedure returns ‘true’, otherwise it returns ‘false’.
5.7. Relation to previous work

The space and time savings of the query-evaluation method introduced in this chapter are due to the fact that some intermediate relations are never built. A similar technique, avoiding the computation of intermediate values, is used in [Liu 1979] to optimize the execution of set-oriented programming languages. However, because Liu was concerned with set-oriented languages and not with relational databases, he considered only the set operators, and not selection, projection, or join. It is the presence of these operators in queries that complicates the query-evaluation method presented here. While it is rather trivial to build membership-test procedures for all nodes of a query tree that contains only set operators, it is not obvious how to do so for a query tree that includes selection, projection, and join. Some of the subtleties that arise in the latter case were mentioned briefly in section 5.6; a complete treatment appears in chapter 6.

Other approaches to query optimization that seek to avoid building intermediate relations are the use of pipelining [Smith and Chang 1975] [Yao 1979] and of tree transformations [Smith and Chang 1975] [Hall 1976]. When pipelining is used, tuples are passed up the tree as they are computed rather than waiting until an entire intermediate relation is formed. Although pipelining decreases the space requirements of query evaluation, all tuples of all intermediate relations are computed; thus, in a single-processor environment there is no decrease in the time requirements of query evaluation.
Tree transformations such as combining sequences of projections into a single projection and combining sequences of selections into a single selection, can reduce the number of intermediate relations in the query tree. Moving selection operators ahead of construction operators can reduce the sizes of the intermediate relations represented by the internal nodes of the query tree. These techniques do not, however, address the question of whether one can avoid building some of the intermediate relations of the transformed tree.

Both pipelining and tree transformations can be profitably combined with our approach. Selective-retrieval and relation-producing procedures can use pipelining rather than building and returning entire relations; relation-producing procedures built for the nodes of a transformed tree are sometimes more efficient than procedures built for the nodes of the original tree.

Similarly, other optimization techniques could be used to produce more efficient membership-test, selective-retrieval, and relation-producing procedures. Exploiting idempotency and unsatisfiability to simplify conditions [Hall 1976] [Eswaran et al 1976] can lead to more efficient membership-test and relation-producing procedures; using the methods of [Astrahan and Chamberlin 1975] [Blasgen and Eswaran 1977] [Selinger et al 1979] [Yao 1979] for access path selection, the methods of [Smith and Chang 1975] for choosing sort orders, and the methods of [Gotlieb 1975] [Wong and Youssefi 1976] for calculating join orders can all lead to more efficient relation-producing procedures.
Chapter 6

Query evaluation using membership-test and selective-retrieval procedures: In-depth presentation

The preceding chapter gave an overview of query evaluation using membership-test and selective-retrieval procedures. Examples were given of relational operators that can use an operand relation's membership-test or selective-retrieval procedure in place of an explicit representation of the operand relation. The operand relation can be either a relation named at a leaf of the query tree or a relation represented by an internal node of the query tree. Section 5.3 covered membership-test and selective-retrieval procedures for leaf relations; section 5.6 gave a few examples of membership-test and selective-retrieval procedures for internal nodes.

In sections 6.1 and 6.2 below we discuss building membership-test and selective-retrieval procedures for each relational operator, and we consider the costs of invoking these procedures. Section 6.3 covers all the relational operators whose relation-producing procedures can be implemented using an operand's membership-test or selective-retrieval procedure rather than its relation-producing procedure. Section 6.4 discusses the problem of choosing among several possible implementations of a node's membership-test, selective-retrieval, or relation-producing procedure.
6.1. Selective-retrieval procedures at internal nodes

We begin by describing how selective-retrieval procedures are built for the internal nodes of a query tree. The design of selective-retrieval procedures is much less complicated than the design of membership-test procedures. The complicating factor for membership-test procedures is the possibility of wildcard values; there is no such problem for selective-retrieval procedures. The only selective-retrieval procedure that presents some difficulties is the one for EQUI-JOIN nodes, which can, if written carelessly, be unnecessarily expensive. The efficient version of the EQUI-JOIN's selective-retrieval procedure requires a slight modification to the definition of selective retrieval; this modification proves useful in the design of other nodes' selective-retrieval and membership-test procedures as well. We thus begin with the EQUI-JOIN node below; the other relational operators are covered in the following sub-sections.

6.1.1. EQUI-JOIN nodes

The selective-retrieval procedure for a node of the form:

\[
( \text{JOIN} \ (R_1, R_2) \ \text{WHEN} \ (R_1.\text{field } i = R_2.\text{field } j))
\]

is straightforward when at least one of the join fields (fields \(i\) and \(j\)) is on the given list of fields \(f\):

1. If both join fields are on list \(f\) and the corresponding values on list \(v\) are not equal, EQUI-JOIN.selective-retrieval returns the empty set.

2. If both join fields are on list \(f\) and the corresponding values on list \(v\) are equal, EQUI-JOIN.selective-retrieval splits \(f\) and \(v\) into \(f_1, v_1\) and \(f_2, v_2\), according to the arities of the operand relations, and returns the cross
product of the sets returned by R1.selective-retrieval( f_1, v_1) and R2.selective-retrieval( f_2, v_2).

(3) If just one join field is on list f, with corresponding value val, EQUI-JOIN.selective-retrieval adds the other join field to f, adds val as the corresponding value, and proceeds as in (2) above.

The associated costs of calling EQUI-JOIN.selective-retrieval are:

(1) constant

(2), (3) \text{cost}(R1.selective-retrieval) + \text{cost}(R2.selective-retrieval) + \\
\text{size(set returned by } R1.selective-retrieval) \times \\
\text{size(set returned by } R2.selective-retrieval) \)

When neither join field is on the given list of fields f, one can reason as follows: f cannot be empty, it must contain at least one field number, which corresponds to a field in R1 or to a field in R2. Assume, without loss of generality, that it corresponds to a field in R1. EQUI-JOIN.selective-retrieval could thus call R1.selective-retrieval, passing as arguments the sublists of f and v that correspond to fields in R1. For every tuple t returned by R1.selective-retrieval, EQUI-JOIN.selective-retrieval could call R2.selective-retrieval. The argument f_2 passed to R2.selective-retrieval would include:

(1) All fields on list f with number greater than the arity of R1, reduced by the arity of R1.

(2) R2’s join field, field \#j.
The argument $v_2$ passed to R2.selective-retrieval would include:

1. The values from $v$ that correspond to the fields of (1) above.

2. The value of $t$'s field $\#i$, where $i$ is R1's join field.

EQUI-JOIN.selective-retrieval would include the cross product of tuple $t$ and the set of tuples returned by R2.selective-retrieval in its answer set. A code fragment implementing this strategy is given in figure 6.1.

$$\text{(JOIN (R1,R2) WHEN (R1.field } \#i = \text{ R2.field } \#j))}$$

\[
\begin{align*}
\text{if ( neither field } \#i \text{ nor field } \#(\text{arity(R1)}+j) \text{ is on list } f \text{ AND } \\
\text{ f includes a field with number } \leq \text{ arity(R1)) } \\
\text{ then } f_1 := \text{ all fields from } f \text{ with number } \leq \text{ arity(R1); } \\
& v_1 := \text{ values from } v \text{ corresponding to the fields of } f_1; \\
& f_2 := \text{ all fields from } f \text{ with number } > \text{ arity(R1), } \\
& \quad \text{ reduced by arity(R1); } \\
& v_2 := \text{ values from } v \text{ corresponding to the fields of } f_2; \\
& f_2 := f_2 \text{ concat field } \#j; \\
& \text{ ANSWER := \{}; } \\
& \text{ forall } t \text{ in R1.selective-retrieval}(f_1, v_1) \text{ do } \\
& \quad v_2 := v_2 \text{ with } t \text{'s field } \#i \text{ corresponding to field } \#j \text{ on list } f_2; \\
& \quad \text{ ANSWER := ANSWER } \cup \\
& \quad (\{t\} \times \text{R2.selective-retrieval}(f_2, v_2));
\end{align*}
\]

Figure 6.1

Selective-retrieval procedure fragment for EQUI-JOIN
when neither join field is on list $f$
and $f$ includes a field number corresponding to a field in R1:
First attempt

There is one obvious problem with the approach of figure 6.1: several calls can be made to R2.selective-retrieval with the same arguments. This happens when several tuples in R1 that have the values of list $v_1$ in the corresponding fields of list $f_1$, have the same value in field $\#i$ (R1's join field). An example of
this objectionable behavior is given in figure 6.2. In this example, the first and third calls to R2.selective-retrieval both have arguments \((f_2 = \#_j, v_2 = 0)\).

![Diagram](image)

arguments passed to R2.selective-retrieval:
- first call: \((f_2 = \#_j, v_2 = 0)\)
- second call: \((f_2 = \#_j, v_2 = 1)\)
- third call: \((f_2 = \#_j, v_2 = 0)\)

Figure 6.2
Example of multiple calls made to R2.selective-retrieval with the same arguments

The problem illustrated in figure 6.2 is not difficult to overcome. Rather than calling R2.selective-retrieval once for each tuple returned by R1.selective-retrieval, EQUI-JOIN.selective-retrieval should call R2.selective-retrieval once for each unique value of field \(#_i\) (R1's join field) in the set returned by R1.selective-retrieval. For each such unique value \(val\), the cross product of the set returned by R2.selective-retrieval is formed with each tuple \(t\) returned by R1.selective-retrieval such that \(t.field \#_i = val\). R1.selective-retrieval may be able to return its answer set ordered according to the values of field \(#_i\) with no extra cost; if not, EQUI-JOIN.selective-retrieval can sort the set returned by
R1.selective-retrieval before calling R2.selective-retrieval. A new version of the code fragment of figure 6.1 is given in figure 6.3.

\[ (\text{JOIN} (R1, R2) \text{ WHEN} (R1.\text{field} \#i = R2.\text{field} \#j)) \]

\[
\text{if} (\text{neither field } \#i \text{ nor field } \#(\text{arity}(R1)+j) \text{ is on list } f \text{ AND} \\
\text{f includes a field with number } \leq \text{arity}(R1)) \\
\text{then } f_1 := \text{all fields from } f \text{ with number } \leq \text{arity}(R1); \\
v_1 := \text{values from } v \text{ corresponding to the fields of } f_1; \\
f_2 := \text{all fields from } f \text{ with number } > \text{arity}(R1), \\
\text{reduced by arity}(R1); \\
v_2 := \text{values from } v \text{ corresponding to the fields of } f_2; \\
f_2 := f_2 \text{ concat field } \#j; \\
\text{ANSWER} := \{\}; \\
\text{TEMP1} := \text{R1.selective-retrieval}(f_1, v_1); \\
\forall \text{values } val \text{ in field } \#i \text{ of TEMP1 do} \\
v_2 := v_2 \text{ with } val \text{ corresponding to field } \#j \text{ on list } f_2; \\
\text{TEMP2} := \text{R2.selective-retrieval}(f_2, v_2); \\
\forall \text{t in TEMP1 such that } t.\text{field } \#i = val \text{ do} \\
\text{ANSWER} := \text{ANSWER } \cup (\{t\} \times \text{TEMP2});
\]

Figure 6.3
Selective-retrieval procedure fragment for EQUI-JOIN
when neither join field is on list f
and f includes a field number corresponding to a field in R1:
Improved version

Unfortunately, there is a more subtle problem associated even with this improved approach. If there is an EQUI-JOIN node n in the subtree rooted at R2, and neither of n's join fields is R2's join field, then several calls with the same arguments can be made to one of n's operands' selective-retrieval procedures. This problem is illustrated in the example of figure 6.4.
Figure 6.4a
Example query-tree fragment
Figure 6.4b
Example series of calls associated with query tree of figure 6.4a

In figure 6.4b, a single call to J1.selective-retrieval causes several calls to be made to J2.selective-retrieval. Although each of these calls has unique arguments, the resulting calls to R3.selective-retrieval do not always have unique arguments.
The solution to this problem is to change the definition of selective retrieval to allow parameter \( v \) to include a set of values for each field in list \( f \). For example, the call: \( \text{R.selective-retrieval}(f = \#1, v = \{0, 1\}) \), means, “Return all tuples in relation \( R \) that have either a zero or a one in their first field”.

Given this possibility, procedure \( \text{J1.selective-retrieval} \) of figure 6.4 calls \( \text{J2.selective-retrieval} \) only once, passing as arguments: \( (f = \#1, v = \{v_1, v_2\}) \). \( \text{J2.selective-retrieval} \) calls \( \text{R2.selective-retrieval} \) passing the same arguments; \( \text{R2.selective-retrieval} \) returns the set: \( \{<v_1, x, a>, <v_1, x, b>, <v_1, y, c>, <v_2, x, d>\} \), and \( \text{J2.selective-retrieval} \) calls \( \text{R3.selective-retrieval} \) once with the arguments: \( (f = \#1, v = \{x, y\}) \). \( \text{J2.selective-retrieval} \) returns the join of the sets returned by \( \text{R2.selective-retrieval} \) and \( \text{R3.selective-retrieval} \), and \( \text{J1.selective-retrieval} \) returns the join of the sets returned by \( \text{R1.selective-retrieval} \) and \( \text{J2.selective-retrieval} \). Figure 6.5 gives the final version of the code fragment introduced in figure 6.1.

The cost of calling an EQUI-JOIN node’s selective-retrieval procedure with argument \( f \) that includes neither of the node’s join fields is:

\[
\text{cost}(\text{R1.selective-retrieval}) + \text{cost}(\text{R2.selective-retrieval}) + \text{cost}(\text{JOIN operation})
\]

Remember that the set returned by \( \text{R1.selective-retrieval} \) must be sorted in order to extract the set of values that appear in \( R1 \)'s join field; similarly, the set returned by \( \text{R2.selective-retrieval} \) must be sorted as part of the JOIN operation.
(JOIN(R1,R2) WHEN (R1.field #i = R2.field #j))

if (neither field #i nor field #(arity(R1)+j) is on list f AND
    f includes a field with number \( \leq \) arity(R1))
then \( f_1 := \) all fields from f with number \( \leq \) arity(R1);
    \( v_1 := \) values from v corresponding to the fields of \( f_1 \);
    \( f_2 := \) all fields from f with number \( > \) arity(R1),
    reduced by arity(R1);
    \( v_2 := \) values from v corresponding to the fields of \( f_2 \);
    \( f_2 := f_2 \) concat field #j;
    TEMP1 := R1.selective-retrieval(\( f_1, v_1 \));
    S := \{ \};
for all values val in field # i of TEMP1 do
    S := S \cup val;
    \( v_2 := v_2 \) concat S;
return(JOIN(TEMP1, R2.selective-retrieval(\( f_2, v_2 \))));

Figure 6.5
Selective-retrieval procedure fragment for EQUI-JOIN
when neither join field is on list f
and f includes a field number corresponding to a field in R1:
Final version

If these two sets are not returned in sorted order, the cost of sorting must be
added to the cost given above.

Allowing parameter v to include sets of values for the associated fields of
parameter f increases the cost of selective-retrieval procedures at leaf nodes in
proportion to the number of values in each set. As we will see in the following
sections, this new possibility is invisible to the selective-retrieval procedures of
PROJECT, UNION, CROSS-PRODUCT, INTERSECTION, and MINUS nodes,
and has only a minimal effect on the selective-retrieval procedures of SELECT
and IS IN nodes.
6.1.2. PROJECT nodes

The selective-retrieval procedure for a PROJECT node, called with arguments \((f, v)\), calls the PROJECT node's child's selective-retrieval procedure, renumbering the fields in list \(f\) as necessary. The set of tuples returned by the PROJECT node's selective-retrieval procedure is the appropriate projection of the set of tuples returned by the PROJECT node's child's selective-retrieval procedure. An example is given in figure 6.6: figure 6.6a shows how PROJECT.selective-retrieval renumbers the fields in list \(f\) in the call to R.selective-retrieval; figure 6.6b shows how PROJECT.selective-retrieval returns a projection of the fields from the set returned by R.selective-retrieval. The cost of calling PROJECT.selective-retrieval is:

\[
\text{cost(renumbering the fields of } f) + \text{cost(R.selective-retrieval)} + \text{cost(projection operation)}
\]
6.1.3. SELECT nodes

In the worst case, a SELECT node's selective-retrieval procedure called with arguments \((f, v)\), calls its child's selective-retrieval procedure with the same
arguments. Tuple $t$ returned by $R$.selective-retrieval, is in the set of tuples returned by $SELECT$.selective-retrieval iff the predicate, $\text{satisfies}(t, condition)$ evaluates to 'true':

```
SELECT.selective-retrieval($f, v$) {
    ANSWER := $\{$;
    forall $t$ in $R$.selective-retrieval($f, v$) do
        if ($\text{satisfies}(t, condition$))
            then ANSWER := ANSWER $\cup$ $\{$$t$$\}$;
    return(ANSWER);
}
```

Figure 6.7
Selective-retrieval procedure for a SELECT node:
Version 1

The call to $R$.selective-retrieval can be avoided when no tuple with the given values in the given fields can possibly be in the relation represented by the $SELECT$ node; in this case, $SELECT$.selective-retrieval returns 'false' without calling $R$.selective-retrieval. To check for this case, a $check$ procedure is written for the $SELECT$ node's $condition$; the $SELECT$ node's selective-retrieval procedure calls its $condition$'s $check$ procedure before calling its child's selective-retrieval procedure. If the $condition$'s $check$ procedure return 'false', $SELECT$.selective-retrieval returns the empty set without calling $R$.selective-retrieval; if the $condition$'s $check$ procedure returns 'true', $SELECT$.selective-retrieval proceeds as in figure 6.7.

The grammar for a $SELECT$ node's $condition$, given in chapter 2, is repeated below.
condition → condition "AND" condition  
| condition "OR" condition  
| single-clause

single-clause → field relop constant  
| field relop field

relop → = | ≠ | < | etc.

The check procedure for an entire condition is composed of check procedures written for each sub-condition. Figure 6.8a gives check procedures for single clauses; figure 6.8b gives the check procedure for an ‘AND’ condition; figure 6.8c gives the check procedure for an ‘OR’ condition; figure 6.9 gives the new version of SELECT.selective-retrieval.

\[(\text{field } \#i = k)\]

clause.check\((f, v)\) {
    if ( \#i is on list f AND no corresponding value from list v* = k)  
        then return(false);
    else return(true);
}

*If field \#i is the join field of an EQUI-JOIN node above this SELECT node in the query tree, then there may be a set of values on list v associated with field \#i on list f.
(field \#i < k)

clause.check(f, v) {
    if ( \#i is on list f AND no corresponding value from list v < k)
        then return(false);
    else return(true);
}

(field \#i = field \#j)

clause.check(f, v) {
    if (both \#i and \#j are on list f AND
        no value from list v corresponding to field \#i
        is equal to a value from list v corresponding to field \#j)
        then return(false);
    else return(true);
}

(field \#i > field \#j)

clause.check(f, v) {
    if (both \#i and \#j are on list f AND
        no value from list v corresponding to field \#i
        is greater than a value from list v corresponding to field \#j)
        then return(false);
    else return(true);
}

Figure 6.8a
Check procedures for single clauses
(c_1 \text{ AND } c_2 \text{ AND } \ldots \text{ AND } c_n)

\text{AND-condition.check}(f, v) \{ \\
\text{if } (c_1.\text{check}(f, v)) \\
\text{then if } (c_2.\text{check}(f, v)) \\
\text{then if } \ldots \\
\text{then return}(c_n.\text{check}(f, v)); \\
\text{return}(\text{false}); \\
\}

\text{Figure 6.8b}
Check procedure for an 'AND' condition

(c_1 \text{ OR } c_2 \text{ OR } \ldots \text{ OR } c_n)

\text{OR-condition.check}(f, v) \{ \\
\text{if } (c_1.\text{check}(f, v)) \\
\text{then return}(\text{true}); \\
\text{else if } (c_2.\text{check}(f, v)) \\
\text{then return}(\text{true}); \\
\text{else if } \ldots \\
\text{else return}(c_n.\text{check}(f, v)); \\
\}

\text{Figure 6.8c}
Check procedure for an 'OR' condition

\text{SELECT.selective-retrieval}(f, v) \{ \\
\text{if } (\text{condition.check}(f, v)) \\
\text{then ANSWER := \{}; \\
\text{forall } t \text{ in R.selective-retrieval}(f, v) \text{ do} \\
\text{if } (\text{satisfies}(t, \text{condition})) \\
\text{then ANSWER := ANSWER \cup \{ t \};} \\
\text{return}(\text{ANSWER}); \\
\text{else return}(\{\}); \\
\}

\text{Figure 6.9}
Selective-retrieval procedure for a SELECT node:
Version 2
The worst-case cost of calling SELECT.selective-retrieval is:

\[
\text{cost(calling condition's check procedure) + cost(R.selective-retrieval) + size(set returned by R.selective-retrieval) * cost(evaluating satisfies predicate)}
\]

6.1.4. UNION and CROSS-PRODUCT nodes

The selective-retrieval procedure for a UNION node, called with arguments \((f, v)\), returns the union of the sets returned by the UNION node's children's selective-retrieval procedures called with the same arguments:

\[
\text{UNION.selective-retrieval}(f, v) \{
\text{return}(R1.selective-retrieval(f, v) \cup R2.selective-retrieval(f, v));
\}
\]

The cost of calling UNION.selective-retrieval is:

\[
\text{cost(R1.selective-retrieval) + cost(R2.selective-retrieval) + cost(union operation)}
\]

The selective-retrieval procedure for a CROSS-PRODUCT node proceeds differently depending on whether the list of fields passed as its first parameter includes field numbers that correspond to fields of both of the CROSS PRODUCT's operand relations, or to fields of just one of the CROSS PRODUCT's operand relations. In the first case, the CROSS-PRODUCT node's selective-retrieval procedure splits its parameters \(f\) and \(v\) into \(f_1, v_1\) and \(f_2, v_2\), according to the arities of its operand relations. The procedure then returns the cross product of the sets returned by its children's selective-retrieval procedures. In the second case, the CROSS-PRODUCT node's selective-retrieval
procedure calls the selective-retrieval procedure of the appropriate child, and the
relation-producing procedure of the other child. It returns the cross product of
the sets returned by its children's procedures:

CROSS-PRODUCT.selective-retrieval(f, v) {
    if (f includes at least one field number \leq\text{arity}(R1) AND
        f includes at least one field number >\text{arity}(R1))
    then f₁ := list of fields from f with numbers \leq\text{arity}(R1);
            v₁ := list of values from v corresponding to the fields of f₁;
            f₂ := list of fields from f with numbers >\text{arity}(R1),
                   reduced by \text{arity}(R1);
            v₂ := list of values from v corresponding to the fields of f₂;
        return(R₁.selective-retrieval(f₁, v₁) \times R₂.selective-retrieval(f₂, v₂));
    else if (f includes only fields numbers \leq\text{arity}(R₁))
    then return(R₁.selective-retrieval(f, v) \times R₂.relation());
    else f := f with all field numbers reduced by \text{arity}(R₁);
        return(R₁.relation() \times R₂.selective-retrieval(f, v));
}

The cost of calling CROSS-PRODUCT.selective-retrieval is one of the following,
depending on which fields are passed as its first parameter:

(1) \text{cost(splitting} f\text{ and} v) + \text{cost}(R₁.selective-retrieval) + \text{cost}(R₂.selective-
                                 retrieval) + \text{size(set returned by} R₁.selective-retrieval) \times \text{size(set returned by}
                                 R₂.selective-retrieval)

(2) \text{cost}(R₁.selective-retrieval) + \text{cost}(R₂.relation) + \text{size(set returned by}
                                 R₁.selective-retrieval) \times \text{size}(R₂)

(3) same as (2), reversing the roles of R₁ and R₂

6.1.5. INTERSECTION and MINUS nodes

There is nothing particularly subtle about the design of selective-retrieval pro-
cedures for INTERSECTION and MINUS nodes, however, as discussed in
chapter 5, there are several possibilities.
INTERSECTION.selective-retrieval can be implemented in any of the following ways:

(1) Given parameters \((f, v)\), INTERSECTION.selective-retrieval returns the intersection of \(R1.selective-retrieval(f, v)\), and \(R2.selective-retrieval(f, v)\).

(2) For every tuple \(t\) returned by \(R1.selective-retrieval(f, v)\), INTERSECTION.selective-retrieval calls \(R2.membership(t)\); \(t\) is in the set returned by INTERSECTION.selective-retrieval iff \(R2.membership\) returns 'true'.

(3) Same as (2), reversing the roles of \(R1\) and \(R2\).

The associated costs of calling INTERSECTION.selective-retrieval are:

(1) \(\text{cost}(R1.selective-retrieval) + \text{cost}(R2.selective-retrieval) + \text{cost (intersection operation)}\)

(2) \(\text{cost}(R1.selective-retrieval) + \text{size (set returned by } R1.selective-retrieval) * \text{cost}(R2.membership)\)

(3) \(\text{cost}(R2.selective-retrieval) + \text{size (set returned by } R2.selective-retrieval) * \text{cost}(R1.membership)\)

MINUS.selective-retrieval can be implemented in either of the following ways:

(1) Given parameters \((f, v)\), MINUS.selective-retrieval returns \(R1.selective-retrieval(f, v) - R2.selective-retrieval(f, v)\).

(2) For every tuple \(t\) returned by \(R1.selective-retrieval(f, v)\), MINUS.selective-retrieval calls \(R2.membership(t)\); \(t\) is in the set returned by MINUS.selective-retrieval iff \(R2.membership\) returns 'false'.

The associated costs of calling MINUS.selective-retrieval are:

(1) \( \text{cost}(\text{R1.selective-retrieval}) + \text{cost}(\text{R2.selective-retrieval}) + \text{cost( minus operation)} \)

(2) \( \text{cost}(\text{R1.selective-retrieval}) + \text{size(set returned by R1.selective-retrieval)} \ast \text{cost( R2.membership)} \)

The actual strategy adopted by a particular INTERSECTION or MINUS node's selective-retrieval procedure depends on the estimated costs of its operands' membership-test and selective-retrieval procedures, and the estimated sizes of the sets returned by its operands' selective-retrieval procedures. The problem of choosing among several possible implementations is discussed further in section 6.4.

6.1.6. IS IN nodes

A tuple \( t \) is in the set returned by the selective-retrieval procedure associated with a node of the form: (field \( i \) OF R1 IS IN R2) iff the following conditions all hold:

(1) \( t \) contains the given values in the given fields.
(2) \( t \in R1 \)
(3) \( t.\text{field } i \in R2 \)

In general, the selective-retrieval procedure for an IS IN node of the form: (field \( i \) OF R1 IS IN R2) can be implemented either by calling its left child's selective-retrieval procedure once and its right child's membership-test procedure a number of times, or by calling each of its children's selective-retrieval procedures once:
(1) \text{IS-IN.selective-retrieval}(f, v) \{
    \text{ANSWER} := \{};
    \textbf{for all } t \text{ in R1.selective-retrieval}(f, v) \textbf{ do}
        \textbf{if } (\text{R2membership}(t))
            \textbf{then } \text{ANSWER} := \text{ANSWER} \cup \{t\};
    \textbf{return} (\text{ANSWER});
\}

(2) \text{IS-IN.selective-retrieval}(f, v) \{
    \text{TEMP1} := \text{R1.selective-retrieval}(f, v);
    S := \text{the set of values in field } #i \text{ of TEMP1};
    \text{TEMP2} := \text{R2.selective-retrieval}((#i), (S));
    \text{ANSWER} := \{};
    \textbf{for all } \text{values } val \text{ in } S \textbf{ do}
        \textbf{if } (\text{val } \in \text{TEMP2})
            \textbf{then } \text{ANSWER} := \text{ANSWER} \cup
            \{\text{all tuples in TEMP1 with val in field } #i \};
    \textbf{return} (\text{ANSWER});
\}

The associated costs of calling \text{IS-IN.selective-retrieval} are:

(1) \text{cost(}R1.selective-retrieval) + \text{size(set returned by R1.selective-retrieval) } \ast
\text{cost(R2membership)}

(2) \text{cost(}R1.selective-retrieval) + \text{cost(extracting set } S \text{ from the set returned by R1.selective-retrieval)} + \text{cost(R2.selective-retrieval)} + \text{size}(S) \ast
\text{cost(membership test for the set returned by R2.selective-retrieval)}

When field $#i$ is on the list of fields $f$, there are two further possibilities:

\text{R2.membership} or \text{R2.selective-retrieval} can be called before calling \text{R1.selective-retrieval}. In the first case, \text{R2.membership} is called once for each value $val$ in parameter $v*$ corresponding to field $#i$ on list $f$, with $val$ as the

*Recall that if field $#i$ is the join field of an EQUI-JOIN node above the IS IN node in the query tree, then there may be a \textit{set} of values associated with field $#i$. 
argument. In the second case, R2.selective-retrieval is called once with field \#i as the first argument, and the corresponding value(s) from parameter \( v \) as the second argument. In both cases, R1.selective-retrieval is then called with \( f \) as the first argument, and \( v' \) as the second argument. \( v' \) matches \( v \) on all values other than those corresponding to field \#i; the values corresponding to field \#i are the values for which R2.membership returned 'true', or the values returned by R2.selective-retrieval. The two new possible versions of IS-IN.selective-retrieval are given below:

1. IS-IN.selective-retrieval(\( f, v \)) { 
   \textbf{if} (field \#i is on list \( f \))
   \textbf{then} \( S := \{\} \);
   \textbf{forall} values \( val \) on list \( v \) corresponding to field \#i on list \( f \) do
   \textbf{if} (R2.membership(\( val \)))
   \textbf{then} \( S := S \cup \{val\} \);
   \textbf{if} (\( S = \{\} \))
   \textbf{then} \textbf{return}(\{\});
   \textbf{else} \( v' := v \) but with \( S \) corresponding to field \#i;
   \textbf{return}(R1.selective-retrieval(\( f, v' \)));
}

2. IS-IN.selective-retrieval(\( f, v \)) { 
   \textbf{if} (field \#i is on list \( f \))
   \textbf{then} \( S := \) the set of values in \( v \) corresponding to field \#i;
   \( v' := R2.selective-retrieval((\#i), (S)) \);
   \textbf{if} (\( v' = \{\} \))
   \textbf{then} \textbf{return}(\{\});
   \textbf{else} \textbf{return}(R1.selective-retrieval(\( f, v' \)));
}
The corresponding costs of calling IS-IN.selective-retrieval are:

(1) \((\text{number of values on list } v \text{ corresponding to field } #i) \times \text{cost}(R2.\text{membership}) + \text{cost}(R1.\text{selective-retrieval})\)

(2) \(\text{cost}(R2.\text{selective-retrieval}) + \text{cost}(R1.\text{selective-retrieval})\)

### 6.2. Membership-test procedures at internal nodes

#### 6.2.1. PROJECT nodes

Building a membership-test procedure for a PROJECT node was discussed in section 5.6. The procedure simply calls its child's membership-test procedure, passing a wildcard value for every field not included in the PROJECT node's list of projected fields. The cost of invoking a PROJECT node's membership-test procedure is thus equal to the cost of invoking the PROJECT node's child's membership-test procedure.

#### 6.2.2. UNION and CROSS-PRODUCT nodes

Building membership-test procedures for UNION and CROSS-PRODUCT nodes is straightforward, even in the presence of wildcard values, given membership-test procedures and arities for the nodes' operand relations. The membership-test procedure for a UNION node with operands R1 and R2, given parameter \(t\), returns the logical \text{or} of R1.membership\((t)\) and R2.membership\((t)\):
UNION.membership(t) {
    if (R1.membership(t))
        then return(true);
    else return(R2.membership(t));
}

The membership-test procedure for a CROSS-PRODUCT node with operands R1 and R2, given parameter t, splits t into two tuples, \(t_1\) and \(t_2\), according to the arities of R1 and R2, and returns the logical and of R1.membership(\(t_1\)) and R2.membership(\(t_2\)):

CROSS-PRODUCT.membership(t) {
    if (R1.membership(\(<t.field \#1, \ldots, t.field \#(arity(R1))>)\))
        then return(R2.membership(\(<t.field \#(arity(R1) + 1), \ldots, t.field \#(arity(R1) + arity(R2))>)\));
    else return(false);
}

The cost of invoking a UNION or CROSS-PRODUCT node's membership-test procedure is, in the worst case, equal to the sum of the costs of invoking the node's children's membership-test procedures.

6.2.3. INTERSECTION, MINUS, EQUI-JOIN, and IS IN nodes

Building a membership-test procedure for an INTERSECTION, MINUS, EQUI-JOIN, or IS IN node is straightforward in the absence of wildcard values; procedures are given below in figure 6.10. The worst-case cost of calling an INTERSECTION, MINUS, EQUI-JOIN, or IS IN node, passing an argument with no wildcard values, is equal to the sum of the costs of calling the node's children's membership-test procedures.
INTERSECT.membership(t) {
    if (R1.membership(t))
        then return(R2.membership(t));
    else return(false);
}

Figure 6.10a
Membership-test procedure for INTERSECTION
in the absence of wildcard values

MINUS.membership(t) {
    if (R1.membership(t))
        then return(¬R2.membership(t));
    else return(false);
}

Figure 6.10b
Membership-test procedure for MINUS
in the absence of wildcard values

EQUI-JOIN.membership(t) {
    if (t.field #i = t.field #(arity(R1) + j))
        then if (R1.membership(<t.field #1, ..., t.field #(arity(R1))>))
            then return(R2.membership(<t.field #(arity(R1) + 1), ..., 
                                       t.field #(arity(R1) + arity(R2))>));
    return(false);
}

Figure 6.10c
Membership-test procedure for an EQUI-JOIN of the form:
JOIN (R1, R2) WHEN (R1.field #i = R2.field #j)
in the absence of wildcard values
IS-IN.membeship(t) {
    if (R1.membeship(t))
        then return(R2.membeship(t.field #i));
    else return(false);
}

Figure 6.10d
Membership-test procedure for an IS-IN of the form:

    field #i of R1 IS IN R2

in the absence of wildcard values

It is easy to construct examples like that of figure 5.4, in which the procedures of figures 6.10a - 6.10d return erroneous results in the presence of wildcard values. For INTERSECTION and MINUS nodes, the presence of any wildcard value in parameter t requires an alternative strategy for membership testing. Two possible strategies were outlined at the end of section 5.6 for INTERSECTION nodes; similar outlines are given below in figure 6.11 for MINUS nodes.
Strategy 1

Step 1:
Call R1.selective-retrieval, passing as arguments the list of fields in t that do not have wildcard values, and the corresponding list of values.

Step 2:
For every tuple u returned by Step 1, call R2.membership(u). As soon as R2.membership returns 'false', the MINUS node's membership-test procedure returns 'true'. If R2.membership never returns 'false', the MINUS node's membership-test procedure returns 'false'.

Strategy 2

Step 1:
Call both R1.selective-retrieval and R2.selective-retrieval, passing as arguments the list of fields in t that do not have wildcard values, and the corresponding list of values.

Step 2:
Subtract the set returned by R2.selective-retrieval from the set returned by R1.selective-retrieval. If the result is non-empty, the MINUS node's membership-test procedure returns 'true', otherwise it returns 'false'.

Figure 6.11
Possible strategies for MINUS.membership in the presence of wildcard values

For EQUI-JOIN and IS IN nodes, an alternative approach to membership testing is required only when certain fields of parameter t have wildcard values.

For an EQUI-JOIN node, the important fields are the join fields, field $i$ and field $j$ in the example of figure 6.10c. If only one of the join fields has a wildcard value, the membership-test procedure requires only a small modification: the non-wildcard join field value replaces the wildcard join field value in the child's membership-test procedure call. A code fragment is given below, illustrating the case where the first join field has a non-wildcard value and the
second join field has a wildcard value:

\[
\begin{align*}
\text{if } & (\neg \text{is-wildcard}(t.\text{field} \#i)) \text{ AND (is-wildcard}(t.\text{field} \#(\text{arity}(R1) + j))) \\
& \text{then if } (R1.\text{membership}( <t.\text{field} \#1, \ldots, t.\text{field} \#(\text{arity}(R1))>) ) \\
& \quad \text{then } t.\text{field} \#(\text{arity}(R1) + j) := t.\text{field} \#i; \\
& \quad \text{return} (R2.\text{membership}( <t.\text{field} \#(\text{arity}(R1) + 1), \ldots, \\
& \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \q
values from \( t \) that correspond to fields of \( R2 \), but using the join field value from \( u \) in place of the wildcard value from \( t \). As soon as \( R2.\text{membership} \) returns 'true', the EQUI-JOIN node's membership-test procedure returns 'true'. If \( R2.\text{membership} \) never returns 'true', the EQUI-JOIN node's membership-test procedure returns 'false'.

**Strategy 3**

Same as strategy 2, but reversing the roles of \( R1 \) and \( R2 \). If \( t \) includes non-wildcard values only in fields corresponding to fields of \( R1 \), the EQUI-JOIN node's membership-test procedure must use strategy 2 above; if \( t \) includes non-wildcard values only in fields corresponding to fields of \( R2 \), the EQUI-JOIN node's membership-test procedure must use strategy 3 above.

The procedure given in figure 6.10d for an IS IN node works unless the named field of \( t \) (\( t.\text{field} \#i \) in the example of figure 6.10d) has a wildcard value. In this case, the membership-test procedure for an IS IN node must pursue one of the following strategies:

**Strategy 1**

Step 1:
Call \( R1.\text{selective-retrieval} \), passing as arguments the list of fields in \( t \) that do not have wildcard values, and the corresponding list of values.

Step 2:
For each tuple \( u \) returned by Step 1, call \( R2.\text{membership} \), passing \( u.\text{field} \#i \) as the argument. As soon as \( R2.\text{membership} \) returns 'true', the IS IN node's membership-test procedure returns 'true'. If \( R2.\text{membership} \) never returns 'true', the IS IN node's membership-test procedure returns 'false'.

**Strategy 2**

Step 1:
Call \( R1.\text{selective-retrieval} \), passing as arguments the list of fields in \( t \) that do not have wildcard values, and the corresponding list of values.
Step 2:
Call R2.selective-retrieval; the first argument is ($#j$), the second argument is the list of values in field $#i$ of the set returned by step 1.

Step 3:
From the set returned by step 1, extract and return all tuples $t$ such that $t$ field $#i$ $\in$ the set returned by step 2.

The approaches to membership testing in the presence of wildcard values discussed above all involve calling a selective-retrieval procedure. The first argument passed to the selective-retrieval procedure is a list of fields from the given tuple $t$ that do not have wildcard values. This is possible only when there is at least one non-wildcard value in $t$. As discussed above in section 6.2.2, the membership-test procedure for a CROSS-PRODUCT node divides its parameter $t$ in two, passing one part to each child's membership-test procedure. It is thus possible for one of the children's membership-test procedures to receive an argument containing only wildcard values.

The membership-test procedures for UNION and CROSS-PRODUCT nodes need not check for this special case; however, the membership-test procedures for all other nodes must, given a pure-wildcard argument, call the node's relation-producing procedure. The reason for this should be clear: calling a node's membership-test procedure with a tuple containing only wildcard values is equivalent to asking "Is the relation represented by this node non-empty?"; the result of calling the node's children's membership-test procedures with pure-wildcard tuples answers the question, "Are the relations represented by the node's children non-empty?". This information is sufficient to answer the same
question about the node itself only in the case of UNION and CROSS-
PRODUCT.

It is reasonable to assume that the list of fields in tuple $t$ that have wild-
card values is available or easily obtainable; thus, one can write membership-test
procedures for INTERSECTION, MINUS, EQUI-JOIN, and IS IN nodes that
behave correctly under all circumstances. An example is given below for MINUS
nodes. In this example, MINUSmembership uses the first strategy given in
figure 6.11 for the case where parameter $t$ contains some wildcard values.

```c
MINUSmembership( t ) {
    if ( $t$ contains only wildcard values )
        then if ( MINUS.relation() is non-empty )
            then return( true );
        else return( false );
    else if ( $t$ contains some wildcard values )
        then $f$ := list of non-wildcard fields of $t$
            $v$ := list of non-wildcard values of $t$
            forall $u$ in R1.selective-retrieval($f$, $v$) do
                if ( $\neg$ R2.membership($u$) )
                    then return( true );
            return( false );
    else if ( R1.membership($t$) )
        then return( $\neg$ R2.membership($t$) );
    else return( false );
}
```

Figure 6.12
Membership-test procedure for MINUS
in the presence of wildcard values

The cost of calling an INTERSECTION, MINUS, EQUI-JOIN, or IS IN
node's membership-test procedure, passing a pure-wildcard argument is equal to
the cost of calling the node's relation-producing procedure. The possible costs of
calling an INTERSECTION node's membership-test procedure passing an argument with some wildcard values, or of calling an EQUI-JOIN node's membership-test procedure with wildcard values in the join fields are:

(1) \( \text{cost(R1.selective-retrieval)} + \text{size(\text{relation returned by R1.selective-retrieval})} \times \text{cost(R2.membership)} \)

(2) \( \text{cost(R1.selective-retrieval)} + \text{cost(R2.selective-retrieval)} + \text{cost(\text{combining the sets returned by R1.selective-retrieval and R2.selective-retrieval})} \)

(3) same as (1) but reversing the roles of R1 and R2

The possible costs of calling a MINUS node's membership-test procedure passing an argument with some wildcard values, or of calling an IS IN node's membership-test procedure with a wildcard value in the critical field, are costs (1) and (2) above.

There are, however, two important considerations regarding the membership-test procedure of figure 6.12, and the similar procedures that would be written for an INTERSECTION, EQUI-JOIN, or IS IN node. First, when a node's membership-test procedure is used in place of its relation-producing procedure during query evaluation, that membership-test procedure is usually called not once, but a number of times. Each call to the node's membership-test procedure results in calls to its children's membership-test procedures, and so on. It may be that the membership-test procedure of some node \( n \) in this series of calls is forced to call its relation-producing procedure because it has received a pure-wildcard argument. This may be unavoidable the first time, however it is not very practical to throw away the results of calling \( n\text{.relation}; \) these results
can be saved, and used to provide a very inexpensive version of \textit{n}.membership for future calls.

Second, the cost of calling an \textsc{INTERSECTION}, \textsc{MINUS}, \textsc{EQUI-JOIN}, or \textsc{IS IN} node's membership-test procedure is obviously affected by the presence of wildcard values in the given argument. Remember that cost estimates for a node's children's membership-test, selective-retrieval, and relation-producing procedures are used to decide how to implement the procedures for the node itself. The fact that a membership-test procedure has a different cost when certain fields of its parameter have wildcard values can be passed up the query tree and used in making this decision.

\textbf{6.2.4. SELECT nodes}

A tuple is in the intermediate relation represented by a \textsc{SELECT} node only if the values of its fields meet the \textit{condition} parameter of the \textsc{SELECT}. A first attempt at writing a membership-test procedure for a \textsc{SELECT} node might be to apply the \textsc{SELECT}'s \textit{condition} to the given tuple \textit{t}; if \textit{t} does not satisfy the \textit{condition} return 'false', else call the \textsc{SELECT} node's child's membership-test procedure with argument \textit{t}:
SELECT.membership(t) {
    if (satisfies(t, condition))
        then return(R.membership(t));
    else return(false);
}

Figure 6.13
Membership-test procedure for SELECT nodes:
First attempt

The possibility of wildcard values, however, rules out such a membership-
test procedure for two reasons:

(1) When parameter t has a wildcard value in field #i, it is not always possible
to evaluate (satisfies(t, condition)), for conditions that mention field #i.

(2) As discussed in chapter 5, a SELECT node's membership-test procedure can
sometimes replace wildcard values with constant values in calls to its child's
membership-test procedure. The procedure of figure 6.13 does not allow for
this useful possibility.

The solutions to these problems are discussed in sections 6.2.4.1 and 6.2.4.2
below. The solution to problem (1) is relatively straightforward, involving, in
the worst case, a call to the SELECT node's child's selective-retrieval or
relation-producing procedure. The solution to problem (2) is more complicated:
the membership-test procedure for the SELECT node must be built using
membership-test procedures written for each component of its condition. Sec-
tion 6.2.4.2 begins by assuming that the condition is in disjunctive normal form;
sections 6.2.4.2.1 through 6.2.4.2.3 describe how to build a membership-test pro-
cedure for each possible kind of disjunct, and how to put these procedures
together to form the membership-test procedure for the SELECT node itself.
Section 6.2.4.2.4 applies the results of the preceding sections to non-normalized conditions. Section 6.2.4.3 summarizes the possibilities for a SELECT node's membership-test procedure.

6.2.4.1. Problem (1)

Recall that the single clauses of a SELECT node's condition can be any of the following:

- \textit{field} = \textit{constant} \\
- \textit{field} <, >, \neq, \textit{etc. constant} \\
- \textit{field} = \textit{field} \\
- \textit{field} <, >, \neq \textit{etc. field}

Problem (1) arises when any one of the following is true:

1. The SELECT node's condition contains a clause of the form: field \#i (operator other than =) \textit{constant}, and t.field \#i has a wildcard value.

2. The SELECT node's condition contains a clause of the form: field \#i = field \#j, and both t.field \#i and t.field \#j have wildcard values.

3. The SELECT node's condition contains a clause of the form: field \#i (operator other than =) field \#j, and either or both of t.field \#i and t.field \#j has a wildcard value.

We call such clauses problem clauses.

When a SELECT node's condition contains only problem clauses, and parameter \( t \) contains at least one non-wildcard value, the SELECT node's membership-test procedure calls its child's selective-retrieval procedure, passing
as arguments the list of non-wildcard fields in $t$, and the corresponding list of values. For every tuple $u$ returned by R.selective-retrieval, the predicate satisfies($u$, $condition$) is evaluated. As soon as the predicate evaluates to 'true', the SELECT node's membership-test procedure returns 'true'. If the predicate never evaluates to 'true', the SELECT node's membership-test procedure returns 'false':

$$\text{SELECT.m}
\text{embership}(t) \{ \\
\quad f := \text{list of non-wildcard fields of } t ; \\
\quad v := \text{list of non-wildcard values of } t ; \\
\quad \text{forall } u \text{ in R.selective-retrieval}(f, v) \text{ do} \\
\quad \quad \text{if (satisfies($u$, $condition$))} \\
\quad \quad \quad \text{then return(true);} \\
\quad \text{return(false);} \\
\}$$

Figure 6.14a
Membership-test procedure for SELECT nodes whose $conditions$
contain only problem clauses
when parameter $t$ has at least one non-wildcard value

The worst-case cost of the procedure of figure 6.14a is:

$$\text{cost(R.selective-retrieval)} + \text{size(set returned by R.selective-retrieval)} \times \text{cost(evaluating satisfies predicate)}$$

When a SELECT node's $condition$ contains only problem clauses, and parameter $t$ contains only wildcard values, the SELECT node's membership-test procedure must call its child's relation-producing procedure. For every tuple $u$ returned by R.relation, the predicate satisfies($u$, $condition$) is evaluated. As soon as the predicate evaluates to 'true', the SELECT node's membership-test procedure returns 'true'. If the predicate never evaluates to 'true', the SELECT node's membership-test procedure returns 'false':
SELECT.membership(t) {
    forall u in R.relation() do
        if (satisfies(u, condition))
            then return(true);
    return(false);
}

Figure 6.14b
Membership-test procedure for SELECT nodes whose conditions contain only problem clauses when parameter t has only wildcard values

The worst-case cost of the procedure of figure 6.14b is:

\[
\text{cost}(R.\text{relation}) + \text{size}(R) \times \text{cost(\text{evaluating satisfies predicate})}
\]

When a SELECT node's condition contains some problem clauses and some non-problem clauses, its membership-test procedure must, in the worst case, call its child's selective-retrieval or relation-producing procedure. In practice, however, it may be cost-efficient to try to avoid calling these more expensive procedures by first dealing with the non-problem clauses. For example, when the SELECT node's condition is of the form: \(c_1 \text{ OR } c_2\), and only \(c_2\) contains problem clauses, the SELECT node's membership-test procedure can return 'true' if both of the following hold:

1. satisfies\((t, c_1)\)

2. R.membership\((t)\)*

Similarly, for a condition of the form: \(c_1 \text{ AND } c_2\), when only \(c_2\) contains problem clauses, the SELECT node's membership-test procedure can return 'false' if

*The argument passed to R.membership may have some wildcard values replaced with constant values; see section 6.2.4.2.
either of the following holds:

(1) \( \neg \text{satisfies}(t, c_1) \)

(2) \( \neg \text{R.membership}(t) \)

Of course, in both of these cases, if the question of membership cannot be solved by considering only condition \( c_1 \), R.selective-retrieval or R.relation must be called.

6.2.4.2. Problem (2)

The solution to problem (2) requires building the SELECT node's membership-test procedure using procedures written for the components of its condition. The presentation of this section is simplified by assuming that the SELECT node's condition is in disjunctive normal form. A condition in disjunctive normal form can consist of:

(1) A single clause.

(2) A number of single clauses joined by AND operators.

(3) Any mixture of (1) and (2) above joined by OR operators.

Sections 6.2.4.2.1 through 6.2.4.2.3 describe how to write membership-test procedures for the three possible forms of a condition in disjunctive normal form listed above. Section 6.2.4.2.4 discusses writing membership-test procedures for SELECT nodes with conditions not in disjunctive normal form.
Membership-test procedures for SELECT nodes with problem clauses in their \textit{conditions} were covered in the previous section; we thus assume below that the \textit{conditions} contain no problem clauses.

\textbf{6.2.4.2.1. Membership-test procedures for single clauses}

A single clause is of one of the following forms:

\begin{itemize}
  \item \textit{field} = \textit{constant}
  \item \textit{field} <, >, \neq, etc. \textit{constant}
  \item \textit{field} = \textit{field}
  \item \textit{field} <, >, \neq, etc. \textit{field}
\end{itemize}

Figure 6.15 gives membership-test procedures for a representative example clause of each of these possible forms.

By definition, wildcard values match any constant value, and are detectable using the predicate \texttt{is_wildcard}:

\begin{verbatim}
(* = 6)  = true
is_wildcard(*) = true
is_wildcard(6) = false
\end{verbatim}

In the examples below, \texttt{R} is the SELECT node’s operand relation, with membership-test procedure \texttt{Rmembership}. For the purposes of these examples, \texttt{R} is assumed to have arity 3.
(field #1 = 6)

clause.membership(t) {
    if (t.field #1 = 6)
        then return(R.membership(<6, t.field #2, t.field #3>));
    else return(false);
}

(field #1 < 0)

clause.membership(t) {
    if (t.field #1 < 0)
        then return(R.membership(t));
    else return(false);
}

(field #1 = field #2)

clause.membership(t) {
    if (is_wildcard(t.field #1))
        then return(R.membership(<t.field #2, t.field #2, t.field #3>));
    else if (is_wildcard(t.field #2))
        then return(R.membership(<t.field #1, t.field #1, t.field #3>));
    else if (t.field #1 = t.field #2)
        then return(R.membership(t));
    else return(false);
}
(field #1 > field #2)

clause.membership(t) {
    if (t.field #1 > t.field #2)
        then return(R.membership(t));
    else return(false);
}

Figure 6.15
Example membership-test procedures for single, non-problem clauses

The first and third examples above illustrate how SELECT nodes' membership-test procedures sometimes replace wildcard values with constant values in further membership-test procedure calls. A tuple is in the relation represented by a SELECT node with the condition (field #1 = 6), only if the first field of the tuple contains the value 6; thus, in the first example of figure 6.15, the value 6 is explicitly placed in the first field of the argument passed to R.membership. Similarly, a tuple is in the relation represented by a SELECT node with the condition (field #1 = field #2) only if the first and second fields of the tuple contain the same value. A parameter with wildcard values in both fields #1 and #2 would make this a problem clause, and we are assuming in this section that we are dealing only with non-problem clauses. It is possible for the parameter to have a wildcard value in one of field #1 or field #2; thus, the third example of figure 6.15 checks for this possibility and replaces such a wildcard value with the constant value from the non-wildcard field.
6.2.4.2.2. Membership-test procedures for ‘AND’ conditions

A disjunct can consist of a single clause, as discussed above, or it can consist of a number of conjuncts of single clauses: \( c_1 \text{ AND } c_2 \text{ AND } \ldots \text{ AND } c_n \). Building a membership-test procedure for such a series of conjuncts requires first building a membership-test procedure for each single clause \( c_i \) as described in the previous section. Given membership-test procedure \( c_i \).member for each clause \( c_i \), one might expect the procedure associated with condition \( c \) of the form: 

\[
( c_1 \text{ AND } c_2 \text{ AND } \ldots \text{ AND } c_n ),
\]
to be the following:

```c

c.member(t) {
    if (c1.member(t))
        then if (c2.member(t))
            then if ...
                then return(c.n.member(t));
        return(false);
}
```

Figure 6.16
Memebship-test procedure for an “AND” condition:
First attempt

Such a procedure can lead to erroneous results when \( t \) contains wildcard values, because each clause \( c_i \) can be satisfied by a different tuple. This is the case for the query fragment of figure 6.17 if the argument passed by the PROJECT node’s membership-test procedure to SELECT node S’s membership-test procedure is \(<*,*,0>\). The first clause of S’s condition is satisfied by the tuple \(<10,0,0>\), and the second clause by the tuple \(<0,20,0>\); thus, using the procedure of figure 6.16, S’s membership-test procedure would return ‘true’, even though the relation represented by S is in fact empty.
Figure 6.17
Example query for which the procedure of figure 6.16 is erroneous

Correct results are obtained by replacing the invocation of R.membership within c₁.membership with an invocation of c₂.membership, replacing the invocation of R.membership within c₂.membership with an invocation of c₃.membership, and so on, leaving an invocation of R.membership only within cₙ.membership. The procedure associated with the condition (c₁ AND c₂ AND ... AND cₙ) is simply c₁.membership. Example procedures are given below in figure 6.18 for the condition of figure 6.17.
\begin{verbatim}
c_{1}.membership(t) \{ 
  if (t.field #1 = 10) 
    then return(c_{2}.membership(<10, t.field #2, t.field #3>));
  else return(false);
\}

c_{2}.membership(t) \{ 
  if (t.field #2 = 20) 
    then return(R.membership(<t.field #1, 20, t.field #3>));
  else return(false);
\}
\end{verbatim}

Figure 6.18
Membership-test procedures for the condition of figure 6.17

\subsection*{6.2.4.2.3. Membership-test procedures for 'OR' conditions}

When a SELECT node's condition is in disjunctive normal form, OR conditions cannot appear as clauses of AND conditions; if there is an OR operator in the condition, it is at the top level. This means that the membership-test procedure written for an OR condition is also the SELECT node's membership-test procedure.

Building a membership-test procedure for a series of disjuncts: \( c_{1} \text{ OR } c_{2} \text{ OR } \ldots \text{ OR } c_{n} \), requires first building a membership-test procedure for each disjunct \( c_{i} \). Disjunct \( c_{i} \) can be a single clause or a number of conjuncts of single clauses; membership-test procedures for these two cases were covered in the previous two sections.

Given membership-test procedure \( c_{i}.membership \) for each disjunct \( c_{i} \), one might expect the membership-test procedure for a SELECT node with a condi-
tion of the form: (c₁ OR c₂ OR ... OR cₙ), to be the following:

```csharp
SELECT.membership(t) {
    if (c₁.membership(t))
        then return(true);
    else if (c₂.membership(t))
        then return(true);
    else ...
        else return(cₙ.membership(t));
}
```

Figure 6.19
Membership-test procedure for an "OR" condition:
First attempt

Although this cannot lead to erroneous results, as did the analogous first-attempt procedure for 'AND' conditions, this procedure can lead to unnecessarily poor behavior. To illustrate this, consider the query tree in figure 6.20a. Associated with relation R is a membership-test procedure R.mem; associated with each condition cᵢ,j is a membership-test procedure cᵢ,j.mem; associated with each SELECT node Sᵢ is a membership-test procedure Sᵢ.mem. The call graph for these procedures is shown in figure 6.20b. When S₁.mem is invoked with tuple t = <0, 0>, whose field values satisfy all conditions cᵢ,j, but which is not a member of R, the series of calls is shown in figure 6.20c (reading each line in turn from left to right), with all procedures returning 'false'.

In this example, the call to S₁.mem results in 2⁴ calls to R.mem; two is the number of disjuncts in each SELECT node's condition, and four is the number of SELECT nodes on the path from S₁ to a leaf of the query tree. The problem is that procedures are invoked again and again with the same arguments, and
Figure 6.20a
Example query for which the procedure of figure 6.19 is unnecessarily costly
Figure 6.20b
Call graph for procedures associated with query of figure 6.20a
thus no hope of returning a new value. We conclude from this example that a
SELECT node's membership-test procedure should not call its operand's
membership-test procedure more than once with the same argument.
The example of figure 6.21 below shows that a further restriction is needed to avoid unnecessarily bad behavior.

\[
\text{SELECT} \quad \begin{array}{c}
\text{field } \#1 = 0 \text{ OR field } \#2 = 0 \\
< 0, * > \\
\end{array}
\]

\[
\begin{array}{c}
R: \\
< 1, 1 > \\
< 1, 0 > \\
\end{array}
\]

\[
c_1.\text{membership}(t) \{
\text{if } (t.\text{field } \#1 = 0)
\quad \text{then return}(\text{R.}\text{membership}(<0, t.\text{field } \#2>)) ;
\quad \text{else return}(\text{false}) ;
\}
\]

\[
c_2.\text{membership}(t) \{
\text{if } (t.\text{field } \#2 = 0)
\quad \text{then return}(\text{R.}\text{membership}(<t.\text{field } \#1, 0>)) ;
\quad \text{else return}(\text{false}) ;
\}
\]

\[
\text{SELECT.}\text{membership}(t) \{
\text{if } (c_1.\text{membership}(t))
\quad \text{then return}(\text{true}) ;
\quad \text{else return}(c_2.\text{membership}(t)) ;
\}
\]

\[\text{Figure 6.21}\]
\[\text{Second example of bad behavior} \]
\[\text{by the membership-test procedure of figure 6.19}\]

In the example of figure 6.21, the argument \( t \) passed to the SELECT node's membership-test procedure has a zero in field \( \#1 \), and a wildcard in field \( \#2 \).
The procedure of figure 6.19 (repeated in figure 6.21) first calls \( c_1.\text{membership} \), which calls \( R.\text{membership} \), passing \( t_1 = <0, *> \) as the argument. \( R.\text{membership} \) returns 'false'. Next, the procedure of figure 6.19 calls \( c_2.\text{membership} \), which calls \( R.\text{membership} \) passing \( t_2 = <0, 0> \) as the argument. \( R.\text{membership} \) again returns 'false'.

In this case the problem is not that the same argument is passed to \( R.\text{membership} \), but that a more restrictive argument is passed to \( R.\text{membership} \).

**Definition:**

Tuple \( t \) is more restrictive than tuple \( u \) iff both of the following hold:

1. All non-wildcard fields of \( u \) have the same value in both \( t \) and \( u \).

2. There is at least one field \( i \) such that \( u.\text{field} \#i \) has a wildcard value, and \( t.\text{field} \#i \) has a non-wildcard value.

Just as there is no point in calling a membership-test procedure repeatedly with the same argument, there is also no point in calling a membership-test procedure repeatedly with more restrictive arguments. In both cases, if the procedure initially returned 'false' it will continue to return 'false'.

At this point, let us put ourselves back in context. We are trying to understand how to build membership-test procedure \( \text{SELECT.membership} \) for a SELECT node with a condition of the form: \(( c_1 \lor c_2 \lor \ldots \lor c_n )\), given membership-test procedure \( c_i.\text{membership} \) for each disjunct \( c_i \), and the membership-test procedure \( R.\text{membership} \) for the SELECT node's operand. The question is how to minimize the cost of \( \text{SELECT.membership} \) by avoiding calls.
to disjuncts' membership-test procedures for which the outcome is predetermined. The answer is the following: from sections 6.2.4.2.1 and 6.2.4.2.2, we know that each disjunct's membership-test procedure includes exactly one call to R.membership; thus, associated with each disjunct's membership-test procedure \( c_i \cdot \text{membership} \) is the tuple \( t_i \) that is passed to R.membership by \( c_i \cdot \text{membership} \). The value of \( \text{R.membership}(t_i) \) is known in advance to be 'false' if a previous call to R.membership with a tuple identical to or less restrictive than \( t_i \) has returned 'false'.

To avoid calling procedures guaranteed to return 'false', SELECT.membership keeps a look-aside set \( S \) of tuples for which R.membership has returned 'false'. Disjunct \( c_i \)'s membership-test procedure is called only if there is no tuple in \( S \) that is identical to or less restrictive than \( t_i \).

It is important to note that there is a difference between the set of tuples for which R.membership has returned 'false', and the set of tuples associated with disjuncts whose membership-test procedures have returned 'false'. Looking back at figure 6.15 we see that a single clause's membership-test procedure returns 'false' without calling R.membership when the parameter \( t \) violates its condition. Tuple \( t_i \) is inserted into look-aside set \( S \) only when \( c_i \cdot \text{membership} \) calls R.membership, and R.membership returns 'false'.

The problem of maintaining set \( S \) so that one can efficiently determine whether, for a given tuple \( t \), there is an element of \( S \) that is identical to or less restrictive than \( t \), is addressed in [Melhorn 1984]. For database applications in
which a SELECT node's condition has only a few disjuncts, or in which the arity of the relations represented by SELECT nodes is small, straightforward methods are adequate. If the SELECT node has only a few disjuncts, there will only be a few values in set S, and sequential search will be reasonable. If the arity of the relation represented by the SELECT node is small, the number of possible tuples less restrictive than or equal to the given tuple t will be small; set S can be represented using a hash table, and all tuples less restrictive than or equal to t can be looked up.

Let us look again at the behavior of the examples of figures 6.20 and 6.21, using the new strategy outlined above. When a call is made to S₁.mem of figure 6.20 with the argument <0, 0>, the following series of calls is made:

S₁.mem → c₁₁.mem → S₂.mem → c₂₁.mem → S₃.mem → c₃₁.mem → R.mem

R.mem returns 'false', and the tuple <0, 0> is put into S₃'s look-aside set. The tuple t₃₂ associated with c₃₂.mem is <0, 0>, which is in S₃'s look-aside set; thus, S₃.mem returns 'false' without calling c₃₂.mem. The tuple <0, 0> is now put into S₂'s look-aside set, and so on, until S₁.mem returns 'false'. Only a single call to R.mem is made using this new strategy.

Similarly, in the example of figure 6.21, a call to SELECT.membership with tuple <0, *> causes a call to c₁.membership, which calls R.membership with the same tuple. R.membership returns 'false', and the tuple <0, *> is put into
the SELECT node's look-aside set. The tuple \( t_2 \) associated with condition \( c_2 \) is \(<0, 0>\), which is more restrictive than the tuple \(<0, *>\) in the SELECT node's look-aside set. Thus, SELECTmembership returns 'false' without calling \( c_2\text{membership} \).

The alert reader might ask, "What if the SELECT node of figure 6.21 were to call \( c_2\text{membership} \) before calling \( c_1\text{membership} \)?" In that case, the tuple passed to \( R\text{membership} \) and subsequently put into the SELECT node's look-aside set would be \(<0, 0>\). The tuple \( t_1 \) associated with \( c_1\text{membership} \) is \(<0, *>\), which is neither equal to nor more restrictive than the tuple \(<0, 0>\) in the SELECT node's look-aside set, and so both \( c_2\text{membership} \) and \( c_1\text{membership} \) would be called. This observation leads to the question of ordering the calls made by a SELECT node to its disjuncts' membership-test procedures.

In the worst case, a SELECT node's membership-test procedure calls all its disjuncts' membership-test procedures, and all of them call its operand's membership-test procedure, which always returns false. An example is given in figure 6.22. This means that ordering calls to a SELECT node's disjuncts' membership-test procedures cannot improve worst-case cost. In practice, however, some orderings may be preferable to others.

The example of figure 6.21 suggests that if the tuple \( t \) passed to SELECTmembership contains wildcard values, a disjunct's membership-test procedure that replaces no wildcard values with constant values should be the
first called. If this call results in a call to R.membership (which happens when \( t \) satisfies the disjunct’s condition), the tuple passed to R.membership is \( t \) itself. If R.membership returns ‘true’, we are done; SELECT.membership returns ‘true’. If R.membership returns ‘false’ we are done as well; the tuples associated with all other disjuncts’ membership-test procedures are guaranteed to be equal to or more restrictive than \( t \), and so SELECT.membership returns ‘false’.

Another way to choose an ordering is to consider the cost of calling the membership-test procedure associated with the SELECT node’s child \( n \). It may be that the cost of this membership-test procedure is higher given a tuple with a wildcard value in a particular field \( f \), than given a tuple with a constant value in field \( f \). This might be the case, for example, when \( f \) is the key field of some relation found at a leaf of the subtree rooted at \( n \), or when \( f \) is the field named by an IS IN node in the subtree rooted at \( n \). When the cost of calling
n's membership-test procedure depends on whether field $f$ has a wildcard or a constant value, an ordering could be chosen according to the value of field $f$ in each disjunct's associated tuple.

Another possible consideration is based on the fact that while tuple $t_1$, in which five wildcard values are replaced with constant values, is, in general, no more restrictive than tuple $t_2$, in which only one wildcard value is replaced with a constant value, intuitively one feels that $\text{R.membership}(t_1)$ is less likely to return true than $\text{R.membership}(t_2)$. The number of wildcard values replaced by constant values can thus be used as a criterion for ordering.

6.2.4.2.4. Membership-test procedures for SELECT nodes with non-normalized conditions

The preceding sections showed how to write membership-test procedures for SELECT nodes, assuming that the SELECT node's condition was in disjunctive normal form. While this assumption simplified the presentation of the SELECT node's membership-test procedure, the fact that a condition put into disjunctive normal form can have exponentially more clauses than the original condition makes the assumption impractical. In this section, we show how to write membership-test procedures for SELECT nodes with non-normalized conditions.

Writing a membership-test procedure for a non-normalized condition is very similar to writing a membership-test procedure for a normalized condition; in both cases, the condition's membership-test procedure is built using
membership-test procedures written for the condition’s components. The membership-test procedures written for single clauses are either the same, or very slightly different; the difference, if any, comes from the way multiple calls to R.membership with the same or more restrictive arguments are avoided. Writing a membership-test procedure for a non-normalized AND condition is conceptually slightly more complicated than writing a membership-test procedure for a normalized AND condition, because the clauses of a non-normalized AND condition have more possible forms than the clauses of a normalized AND condition. Writing membership-test procedures for non-normalized AND conditions is the first topic covered below. In a non-normalized condition, OR conditions can appear as clauses of AND conditions. This means that the membership-test procedure written for an OR condition is not necessarily the SELECT node’s membership-test procedure. After discussing how to write membership-test procedures for non-normalized AND conditions, we show how to write membership-test procedures for non-normalized OR conditions, and how to write the membership-test procedure for the SELECT node as a whole.

In a normalized condition, AND operators can be applied only to single clauses; in a non-normalized condition, AND operators can be applied to arbitrary sub-conditions. For example, a non-normalized condition can contain a clause of the form: \( (c_1 \text{ AND } c_2 \text{ AND } \ldots \text{ AND } c_n ) \), where one or more of the \( c_i \) contain OR conditions. These OR conditions can, in turn, contain AND conditions, and so on.
To write the membership-test procedure for a normalized AND condition ($c_1 \text{ AND } c_2 \text{ AND } ... \text{ AND } c_n$), we write a membership-test procedure $c_i$.membership for each clause $c_i$, and then replace the call made to R.membership in each procedure $c_i$.membership other than $c_n$.membership, with a call to $c_{i+1}$.membership. We cannot do exactly the same thing for a non-normalized AND condition, because a clause $c_i$ can itself contain an AND condition. In this case, some of the calls to R.membership within procedure $c_i$.membership should be replaced with calls to $c_{i+1}$.membership, and some should be replaced with calls to the membership-test procedure written for the rightmost conjunct of the interior AND operator.

The attribute grammar fragment of figure 6.23 defines the membership-test procedure called by each single clause’s membership-test procedure in a non-normalized condition. In this attribute grammar fragment, each component of a condition has two attributes: membership-test-proc is the name of the membership-test procedure written for the component; proc-to-call is the name of the membership-test procedure the component calls if it is a single clause. If the component is not a single clause, it is either an OR condition or an AND condition. If the component is an OR condition, the value of its proc-to-call attribute is passed down as the value of both disjuncts’ proc-to-call attributes. If the component is an AND condition, the value of its proc-to-call attribute is passed down as the value of its right conjunct’s proc-to-call attribute; the value of its right conjunct’s membership-test-proc attribute is passed down as the
value of the left conjunct’s proc-to-call attribute.

Figure 6.24 shows an example condition tree; the leaves of the tree, which represent single clauses, are labelled with the membership-test procedures they call.

As stated at the beginning of this section, the membership-test procedure written for a non-normalized OR condition is not necessarily the membership-test procedure of the SELECT node as a whole. The problems of avoiding multiple calls to R.membership with the same or more restrictive arguments, and of ordering the calls made to R.membership according to the values of the arguments passed by each call, must still be dealt with; however, for SELECT nodes with non-normalized conditions, these problems are no longer the concern of the OR condition’s membership-test procedure. The membership-test procedure written for a non-normalized OR condition of the form:

\[(c_1 \text{ OR } c_2 \text{ OR } ... \text{ OR } c_n)\], is the one first suggested in section 6.2.4.2.3:

```java
condition.membership(t) {
    if (c_1.membership(t))
        then return(true);
    else if (c_2.membership(t))
        then return(true);
    else if ...
        else return(c_n.membership(t));
}
```

We have yet to say what constitutes the membership-test procedure for a SELECT node with a non-normalized condition, and how to solve the two problems mentioned above. There are two possibilities: if we are willing to forgo the
# attribute declarations
#

membership-test-proc: synthesized procedure-name attribute of query-tree-node;

proc-to-call: inherited procedure-name attribute of cond;
membership-test-proc: synthesized procedure-name attribute of cond;

proc-to-call: inherited procedure-name attribute of single-clause;
membership-test-proc: synthesized procedure-name attribute of single-clause;

# productions and semantic equations
#

query-tree-node -> "SELECT FROM" query-tree-node "WHEN" cond
{ query-tree-node\$1.membership-test-proc =
cond.membership-test-proc;
cond.proc-to-call = query-tree-node\$2.membership-test-proc; }

cond -> single-clause
{ cond.membership-test-proc = single-clause.membership-test-proc;
single-clause.proc-to-call = cond.proc-to-call; }

| cond "OR" cond
{ cond\$1.membership-test-proc =
  build-OR-cond( cond\$2.membership-test-proc,
                    cond\$3.membership-test-proc );
cond\$2.proc-to-call = cond\$1.proc-to-call;
cond\$3.proc-to-call = cond\$1.proc-to-call; }

| cond "AND" cond
{ cond\$1.membership-test-proc = cond\$2.membership-test-proc;
cond\$2.proc-to-call = cond\$3.membership-test-proc;
cond\$3.proc-to-call = cond\$1.proc-to-call; }

Figure 6.23
Attribute grammar fragment defining the procedures called by the membership-test procedures written for the components of a non-normalized condition.
ability to order the calls made to R.membership, then the membership-test procedure of the SELECT node is the membership-test procedure written for the highest-level component of its condition, and multiple calls to R.membership with the same or more restrictive arguments are avoided as follows. A look-aside set of tuples as described in section 6.2.4.1.3 is maintained; the membership-test procedure of each single clause that contains a call to R.membership looks up the argument that it is about to pass before actually making the call. If the argument itself, or a less restrictive tuple is already in the look-aside set, the single clause’s membership-test procedure returns ‘false’ without calling R.membership. If there is no relevant tuple already in the look-aside set, the
call to \texttt{Rmembership} is made. If \texttt{Rmembership} returns ‘true’, the single clause’s membership-test procedure returns ‘true’; if \texttt{Rmembership} returns ‘false’, the single clause’s membership-test procedure adds the argument to the look-aside set and returns ‘false’. This checking of the look-aside set has to be done at the single-clause level because, contrary to the situation for \textit{conditions} in disjunctive normal form, there is no one membership-test procedure making a series of calls to procedures each of which includes exactly one call to \texttt{Rmembership}. This is also the reason why, in this case, it is not possible to order the calls made to \texttt{Rmembership}.

It is possible to write a membership-test procedure for a \texttt{SELECT} node with a \textit{condition} that is not in disjunctive normal form that orders the calls made to \texttt{Rmembership}; this requires a slightly different approach than the one we have been discussing. To allow the ordering of calls made to \texttt{Rmembership}, the procedures built for the components of the \texttt{SELECT} node’s \textit{condition} do not actually call \texttt{Rmembership}; instead, each procedure returns the set of tuples that would normally have been passed as arguments to \texttt{Rmembership} as a result of its invocation. New procedures are given below for the representative example single clauses of figure 6.15.
(field #1 = 6)

clause.membership(t) {
    if (t.field #1 = 6)
        then return({<6, t.field #2, t.field #3>});
    else return({});
}

(field #1 < 0)

clause.membership(t) {
    if (t.field #1 < 0)
        then return({t});
    else return({});
}

(field #1 = field #2)

clause.membership(t) {
    if (is_wildcard(t.field #1))
        then return({<t.field #2, t.field #2, t.field #3>});
    else if (is_wildcard(t.field #2))
        then return({<t.field #1, t.field #1, t.field #3>});
    else if (t.field #1 = t.field #2)
        then return({t});
    else return({});
}
(field #1 > field #2)

clause.membership(t) {
    if (t.field #1 > t.field #2)
        then return({t});
    else return({});
}

Remember that when a single clause is the descendent of a conjunct of an AND condition, its membership-test procedure may call the AND condition’s rightmost conjunct c_n’s membership-test procedure in place of calling R_membership. In this case, the new version of the single clause’s membership-test procedure behaves as follows:

If the given tuple t satisfies its condition, the single clause’s membership-test procedure returns the result of calling c_n_membership with the appropriate argument.

If the given tuple t does not satisfy its condition, the single clause’s membership-test procedure returns the empty set.

An example is given below:

(f #1 = 6) AND (f #2 < 0)

c1.membership(t) {
    if (t.field #1 = 6)
        then return(c2.membership(<6, t.field #2, t.field #3>));
    else return({});
}

c2.membership(t) {
    if (t.field #2 < 0)
        then return({t});
    else return({});
}
The new version of the membership-test procedure written for an OR condition of the form: \((c_1 \text{ OR } c_2 \text{ OR } \ldots \text{ OR } c_n)\), is:

\[
\text{OR.member}\text{ship}(t) = \begin{cases} 
\text{return}(c_1\text{.membership}(t) \cup c_2\text{.membership}(t) \cup \ldots \cup c_n\text{.membership}(t)); \\
\end{cases}
\]

The membership-test procedure for the SELECT node calls the membership-test procedure of its condition's top-level component to get the set of tuples that are to be passed to R.member\text{ship}. This set is exactly the same as the set of tuples \(t_i\) associated with the disjuncts \(c_i\) of the condition were it put into disjunctive normal form. The SELECT node's membership-test procedure can now order the calls to R.member\text{ship}, make the calls, and manage the look-aside set just as is done by the membership-test procedure of a SELECT node with a condition in disjunctive normal form.

### 6.2.5. Membership-test procedures for SELECT nodes: summary

When a SELECT node's condition contains only problem clauses, the SELECT node's membership-test procedure may have to call the node's child's selective-retrieval or relation-producing procedure. The cost of SELECT.member\text{ship} in this case is one of the following:

1. \[
\text{cost}(\text{R.selective-retrieval}) + \text{size(set returned by R.selective-retrieval}) \ast 
\text{cost(evaluating satisfies predicate)}
\]
2. \[
\text{cost}(\text{R.relation}) + \text{size}(\text{R}) \ast \text{cost(evaluating satisfies predicate)}
\]

When the condition contains no problem clauses, the SELECT node's membership-test procedure can determine whether the given tuple \(t\) is in the
relation represented by the SELECT node by calling only the node's child's membership-test procedure. SELECT.membership can avoid making unproductive calls to R.membership by consulting a look-aside set of tuples for which R.membership has already returned 'false' before making a new call. Ordering the calls to R.membership can also cut down on the cost of SELECT.membership.

In the worst case, however, SELECT.membership must call R.membership once for each disjunct in the disjunctive-normal-form version of the SELECT node's condition. The cost of SELECT.membership in this case is:

\[(\text{number of disjuncts in the disjunctive-normal-form version of the condition}) \ast \text{cost}(\text{R.membership}).\]

In some cases, this cost may be higher than the cost of calling R.selective-retrieval or R.relation and checking whether any returned tuple satisfies the condition. The implementation of a particular SELECT node's membership-test procedure must thus depend on the node's condition and the costs of calling the node's child's membership-test, selective-retrieval, and relation-producing procedures, as well as depending on the presence or absence of problem clauses in the condition.

6.3. Using membership-test and selective-retrieval procedures

Chapter 5 described how membership-test and selective-retrieval procedures can be used in the implementation of INTERSECTION and EQUI-JOIN respectively. In this section, we cover all the relational operators that can use an
operand's membership-test or selective-retrieval procedure in place of an explicit representation of the operand relation. In other words, the relation-producing procedures attached to nodes with these operators can call one of the node's children's membership-test or selective-retrieval procedures instead of its relation-producing procedure. Of course, a node's relation-producing procedure can always be implemented using only the relation-producing procedures of its children. Choosing among all possible implementations of a node's relation-producing procedure is discussed in section 6.4.

6.3.1. Using membership-test procedures

The relation-producing procedures for INTERSECTION, MINUS, and IS IN nodes can be implemented using a child's membership-test procedure. The relation-producing procedure for an INTERSECTION operator can use either child's membership-test procedure, using the other child's relation-producing procedure. The procedure below uses the right child's membership-test procedure, and the left child's relation-producing procedure.

```
INTERSECTION.relation() {
    ANSWER := {};
    for each tuple t in R1.relation() do
        if (R2.membership(t))
            then ANSWER := ANSWER \cup \{ t \};
    return(ANSWER);
}
```

Figure 6.25
Relation-producing procedure for INTERSECTION using a membership-test procedure
The relation-producing procedures for MINUS and IS IN nodes can only use the right child's membership-test procedure in place of its relation-producing procedure.

```
MINUS.relational() {
  ANSWER := {};
  for each tuple t in R1.relational() do
    if (¬ R2.membership(t))
      then ANSWER := ANSWER ∪ { t }
  return(ANSWER);
}
```

**Figure 6.26**
Relation-producing procedure for MINUS
using a membership-test procedure

```
IS IN.relational() {
  ANSWER := {};
  TEMP1 := R1.relational();
  forall values val in field #i of TEMP1 do
    if (R2.membership(<val>))
      then ANSWER := ANSWER ∪
           (all tuples t in TEMP1 such that t.field #i = val);
  return(ANSWER);
}
```

**Figure 6.27**
Relation-producing procedure for IS IN of the form:
(field #i of R1 IS IN R2)
using a membership-test procedure

The procedure of figure 6.27 uses the technique introduced in figure 6.5 to avoid calling a procedure several times with the same arguments. In this case, multiple calls to R2.membership with the same argument are avoided by calling
R2.membership once for each unique value in relation R1, rather than once for each tuple in relation R1.

The cost of implementing INTERSECTION, MINUS or IS IN using a membership-test procedure as illustrated in figures 6.25, 6.26, and 6.27, is, in the worst case:

\[ \text{cost}(R_1.\text{relation}) + \text{size}(R_1) \times \text{cost}(R_2.\text{membership}) \]

### 6.3.2. Using selective-retrieval procedures

The relation-producing procedures for EQUI-JOIN, IS IN, and some SELECT nodes can be implemented using a child's selective-retrieval procedure. The relation-producing procedure for an EQUI-JOIN node can use either child's selective-retrieval procedure, using the other child's relation-producing procedure. The relation-producing procedure for an IS IN node can only use its left child's selective-retrieval procedure and its right child's relation-producing procedure.

The implementation of EQUI-JOIN using a selective-retrieval procedure given in chapter 5 is similar to the first attempt at a selective-retrieval procedure for EQUI-JOIN given in figure 6.1; both call R2.selective-retrieval once for every tuple in relation R1. As discussed in section 6.1.1, this can result in multiple calls with the same argument being made, a duplication of effort that we would like to avoid. The relation-producing procedures for EQUI-JOIN and IS IN nodes given below in figure 6.28 use the ideas developed in section 6.1.1: a single
call is made to R1.selective-retrieval, passing a set of values associated with a single field.

\[
\text{EQUI-JOIN.\text{relation}}() \{ \\
\quad \text{TEMP2} := \text{R2.\text{relation}}(); \\
\quad S := \text{the set of values in field } \#j \text{ of TEMP2}; \\
\quad \text{return} (\text{JOIN}(\text{TEMP2}, \text{R1.\text{selective-retrieval}}((\#i), (S)))); \\
\}
\]

Figure 6.28a
Relation-producing procedure for an EQUI-JOIN of the form:
JOIN (R1, R2) WHEN (R1.field \#i = R2.field \#j)
Implemented using a selective-retrieval procedure

\[
\text{IS-IN.\text{relation}}() \{ \\
\quad \text{TEMP2} := \text{R2.\text{relation}}(); \\
\quad S := \text{the set of values in TEMP2}; \\
\quad \text{return} (\text{R1.\text{selective-retrieval}}((\#i), (S)));
\]

Figure 6.28b
Relation-producing procedure for an IS IN node of the form:
field \#i of R1 IS IN R2
Implemented using a selective-retrieval procedure

The cost of implementing EQUI-JOIN using a selective-retrieval procedure as illustrated in figure 6.29a is:

\[
\text{cost(R2.\text{relation}) + cost(extracting set S) + cost(R1.\text{selective-retrieval}) + cost(JOIN operation)}
\]

The cost of implementing IS IN using a selective-retrieval procedure as illustrated in figure 6.29b is:

\[
\text{cost(R2.\text{relation}) + cost(extracting set S) + cost(R1.\text{selective-retrieval})}
\]

A SELECT node's relation-producing procedure can be implemented using its child's selective-retrieval procedure when there are one or more field, value-
set pairs \( <f, V> \) such that every tuple in the relation represented by the SELECT node has one of the values in set \( V \) in field \( f \). We call these pairs \textit{required pairs}. Whether a particular SELECT node has any associated required pairs, and if so what their values are, can be determined by making a single post-order traversal of the node’s \textit{condition} tree. Rules for computing a \textit{condition}’s set of required pairs are given below.

\[
\begin{align*}
\text{condition} & \rightarrow \text{field} = \text{constant} \\
& \quad \text{condition}.\text{required-pairs-set} = \{ <\text{field}, \{\text{constant}\}> \}; \\
\text{condition} & \rightarrow \text{field} <,>, \neq \text{etc. constant} \\
& \quad \text{condition}.\text{required-pairs-set} = \{ \}; \\
\text{condition} & \rightarrow \text{field relop field} \\
& \quad \text{condition}.\text{required-pairs-set} = \{ \}; \\
\text{condition} & \rightarrow \text{condition AND condition} \\
& \quad \text{condition}^\$1.\text{required-pairs-set} = \text{condition}^\$2.\text{required-pairs-set} \cup \\
& \quad \quad \text{condition}^\$3.\text{required-pairs-set}; \\
\text{condition} & \rightarrow \text{condition OR condition} \\
& \quad \text{condition}^\$1.\text{required-pairs-set} = \\
& \quad \quad \{ <f, V \cup V'> \mid <f, V> \in \text{condition}^\$2.\text{required-pairs-set} \text{ AND} \\
& \quad \quad \quad <f, V'> \in \text{condition}^\$3.\text{required-pairs-set} \};
\end{align*}
\]

If the required pairs set associated with a SELECT node’s \textit{condition} is non-empty, the relation-producing procedure for the SELECT node can be implemented by calling the node’s child’s selective-retrieval procedure. The list of fields passed to the selective-retrieval procedure includes all the fields from the \textit{condition}’s required pairs set; the list of sets of values passed to the selective-retrieval procedure includes, for each field \( f \) from the \textit{condition}’s required pairs set, the corresponding set of values \( V \). Each tuple in the set returned by the
selective-retrieval procedure is then checked to make sure that is satisfies the entire \emph{condition} of the SELECT node:

\begin{verbatim}
SELECT.relation() {
    if (condition.required-pairs-set is non-empty)
        then f := the list of field numbers in condition.required-pairs-set;
            v := the corresponding list of value sets;
            ANSWER := {};
            forall u in R.selective-retrieval(f, v) do
                if (satisfies(u, condition))
                    then ANSWER := ANSWER \cup \{ u \};
            return(ANSWER);
}
\end{verbatim}

Figure 6.29
Relation-producing procedure for SELECT
Implemented using a selective-retrieval procedure

The cost of implementing \texttt{SELECTION} using a selective-retrieval procedure as illustrated in figure 6.29 is:

$$
\text{cost(R.selective-retrieval)} + \text{size(set returned by R.selective-retrieval)} \times \\
\text{cost(evaluating the satisfies predicate)}
$$

6.4. Choosing among several possible implementation

In sections 6.1 and 6.2 we saw that there are often several possible ways to implement a node's membership-test or selective-retrieval procedure. Similarly, as discussed in section 6.3, there may be a number of ways to implement a node's relation-producing procedure. We have said that cost estimates are computed for all procedures associated with a node and are used to choose the most efficient implementations for the procedures associated with the node's parent.
In this section, we consider some of the issues involved in estimating the cost of a procedure, and in choosing among several possible implementations.

In some cases, the cost of a procedure depends on the size of an intermediate value: the set returned by a selective-retrieval or a relation-producing procedure. The problem of estimating the sizes of intermediate relations has been studied in [Selinger et al 1979], [Demolombe 1980], [Richard 1981], [Christodoulakis 1983], and [Piatetsky-Shapiro 1984]; the accuracy of the estimate may vary from application to application and from instance to instance.

Sometimes the cost of a membership-test procedure depends on which fields of its parameter have wildcard values; similarly, the cost of a selective-retrieval procedure may depend on which fields are in the list of fields passed as its first parameter. For such procedures, a set of cost estimates and the associated criterion for each cost can be computed and passed up the query tree. These values can be used to choose among implementations at the next level up in the tree, and to compute a set of cost estimates and associated criteria for the chosen implementation.

Although in some applications it may not be possible to make accurate cost estimates for all procedures, one of the advantages of query evaluation using membership-test and selective-retrieval procedures is that the techniques can be used selectively. A worst-case and a best-case cost estimate can be made for each procedure. For operators that can be implemented using a membership-test or selective-retrieval procedure, the cost of doing so is computed using the
worst-case estimates; the cost of implementing the operator using only relation-
producing procedures is computed using the best-case estimates. If, taking this
obviously biased approach, the estimated cost of implementing the operator
using the membership-test or selective-retrieval procedure is lower than the
estimated cost of implementing the operator using only relation-producing pro-
cedures, it will surely be advantageous to use the membership-test or selective-
retrieval procedure.

We should also remember that the motivation for the development of query
evaluation using membership-test and selective-retrieval procedures was the pos-
sibility of using implicit relations in queries. Given the extremely high cost of
actually building an implicit relation, choosing between two possible procedure
implementations, one of which requires building the implicit relation and the
other of which does not, should not be too difficult.
Chapter 7

Incremental View Updating

As discussed in chapter 3, relational views can be used to indicate the presence of static-semantic errors and anomalies in a program. A view that is displayed during editing may require updating after every editing operation. Recomputing views from scratch is obviously too costly in this context; incremental view updating is needed.

This chapter presents a new technique for incremental view updating that relies on the concepts of membership testing and selective retrieval introduced in chapter 5. Although the motivation for developing the method was the need to keep views displayed by an interactive editing environment up-to-date, it can be used by any database-management system that supports view definitions.

This chapter is concerned with updating views and not with updating the relations used in view definitions; that issue is addressed in chapter 8. Here, the sets of changes made to the relations used in a view definition since the view was last updated are assumed to be given.

7.1. Algorithm overview

A change to relation R can be represented as two (disjoint) sets:

\[ \Delta^+_R \] a set of tuples not in R to be added to R

\[ \Delta^-_R \] a set of tuples in R to be removed from R
Our goal for incremental view updating is the following:

Given: $d$, $v$, $\Delta^+_{R_1}, \Delta^-_{R_1}, \ldots, \Delta^+_{R_m}, \Delta^-_{R_m}$

Compute: $\Delta^+_v$, $\Delta^-_v$

and use $\Delta^+_v$ and $\Delta^-_v$ to update $v$.

The cornerstone of our incremental updating algorithm is the use of the membership-test and selective-retrieval procedures for the relations represented by the internal nodes of the view-definition tree. These procedures are built once when the view is first defined. When a view update is required, a post-order traversal of the view-definition tree is made; changes $\Delta^+_n$ and $\Delta^-_n$ are computed for each node $n$ in turn, using the changes computed for $n$'s children, and their membership-test or selective-retrieval procedures.

7.2. Advantages of this approach

There are three major advantages to this approach as compared to previous work on incremental view updating [Koenig and Paige 1981] [Shmueli and Itai 1984]:

1. Very few intermediate values must be maintained between view updates or recomputed during view updating. The only relations other than leaf relations and the view itself that must be maintained or recomputed are the operands of CROSS-PRODUCT nodes. Maintaining other intermediate relations allows faster view updates at the expense of the space needed to store the relations. The update algorithm can be tuned, either statically or dynamically, according to the relative importance of time and space costs.
(2) Changes can cancel each other. For example, given expression $E : (R1 \rightarrow R2)$, simultaneously adding a tuple to both R1 and R2 causes no change to the value of $E$. Propagating changes through the view-definition tree in post order allows such changes to cancel as soon as possible.

(3) Although view updating can be performed after a single change to a leaf relation as in [Koenig and Paige 1981], it is also possible to delay updating until a series of changes has been made.

A complete comparison of the incremental view-updating method presented in this chapter with previous work on incremental view updating is given below in section 7.8.

### 7.3. Membership-test and selective-retrieval procedures

During view updating, a node $n$ of the view-definition tree can be thought of as representing two different relations:

- An *old* relation, reflecting the value of the subtree rooted at $n$ evaluated before the latest update to the underlying database is made.

- A *new* relation, reflecting the value of the subtree rooted at $n$ evaluated after the latest update to the underlying database is made.

Two membership-test and two selective-retrieval procedures are thus built at each node $n$ of the view-definition tree: $n$.old-membership, $n$.new-membership, $n$.old-selective-retrieval, and $n$.new-selective-retrieval. For example, given relations R1 and R2:

$$R1 = \{a, b, c\}$$

$$R2 = \{b, c, d\}$$
and changes to R1:

\[ \Delta_{R1}^+ = \{d\} \]

\[ \Delta_{R1}^- = \{b\} \]

for root \( n \) of the subtree: \((R1 \cap R2)\), the following hold:

- \( n.\text{old-membership}(d) = \text{false} \)
- \( n.\text{new-membership}(d) = \text{true} \)
- \( n.\text{old-membership}(b) = \text{true} \)
- \( n.\text{new-membership}(b) = \text{false} \)

It is easy to extend the methods of chapter 6 for building \( n.\text{membership} \) and \( n.\text{selective retrieval} \) to build \( n.\text{old-membership} \), \( n.\text{new-membership} \), \( n.\text{old-selective-retrieval} \), and \( n.\text{new-selective-retrieval} \). The design of these old and new procedures requires the following protocol:

1. Before each view update, for each leaf relation \( l \), all tuples in \( \Delta_l^+ \) are added to \( l \) and marked 'new'; all tuples in \( \Delta_l^- \) are marked 'deleted'.

2. When view updating is finished, for each leaf relation \( l \), all tuples in \( \Delta_l^+ \) are unmarked; all tuples in \( \Delta_l^- \) are removed from \( l \).

Old membership-test and selective-retrieval procedures ignore tuples marked 'new' and include tuples marked 'deleted'; new membership-test and selective-retrieval procedures include tuples marked 'new' and ignore tuples marked 'deleted'.

7.4. Computing changes to intermediate relations

Membership-test and selective-retrieval procedures are the key to our approach to incremental view updating. Using these procedures, we compute only the changes to the relations represented by the nodes of the view-definition tree. The computed changes are used to update maintained relations, including the view relation. The cost of incremental view updating is proportional to the sum of the sizes of the changes to the view-definition tree relations. By contrast, the cost of non-incremental view updating is proportional to the sum of the sizes of the entire new view-definition tree relations.

The following subsections show how membership-test and selective-retrieval procedures are used to compute the changes to each of the possible nodes of a view-definition tree, given the changes to the node's children. In each case, three versions of the formula for computing the changes are given: an English version, an equational version, and, to show clearly the use of the membership-test and selective-retrieval procedures, a procedural version.

7.4.1. Computing the changes to a UNION, INTERSECTION, or MINUS node

Computing the changes to the relation represented by a UNION, INTERSECTION, or MINUS node, given the changes to the node's children, is relatively straightforward. The procedures given below that compute these changes use
only the nodes' children's membership-test procedures; there is no need to use a selective-retrieval procedure.

7.4.1.1. UNION

For node \( n \) of the form: \((R1 \ UNION \ R2)\), \( \Delta^+_n \) includes:

- all tuples added to \( R1 \) that were not in the "old" \( R2 \)
- all tuples added to \( R2 \) that were not in the "old" \( R1 \)

\[
\Delta^+_n = \{ t \in \Delta^+_R | t \not\in \text{old-}R2 \} \cup \{ t \in \Delta^+_R | t \not\in \text{old-}R1 \}
\]

\( \Delta^-_n \) includes:

- all tuples removed from \( R1 \) that are not in the "new" \( R2 \)
- all tuples removed from \( R2 \) that are not in the "new" \( R1 \)

\[
\Delta^-_n = \{ t \in \Delta^-_R | t \not\in \text{new-}R2 \} \cup \{ t \in \Delta^-_R | t \not\in \text{new-}R1 \}
\]

In procedural form:

```plaintext
compute-UNION-changes(\( \Delta^+_R1, \Delta^-_R1, \Delta^+_R2, \Delta^-_R2 \)) {
    \( \Delta^+_n := \{ \}; \)
    \( \text{forall } t \in \Delta^+_R1 \) do
        if (\( \neg R2.\text{old-membership}(t) \))
        then \( \Delta^+_n := \Delta^+_n \cup \{t\}; \)
    \( \text{forall } t \in \Delta^+_R2 \) do
        if (\( \neg R1.\text{old-membership}(t) \))
        then \( \Delta^+_n := \Delta^+_n \cup \{t\}; \)
```

\[ \Delta_n^- := \{}; \]

forall \( t \in \Delta_{R1}^- \) do

if (\( \neg \text{R2.new-membership}(t) \))
then \( \Delta_n^- := \Delta_n^- \cup \{t\} \);

forall \( t \in \Delta_{R2}^- \) do

if (\( \neg \text{R1.new-membership}(t) \))
then \( \Delta_n^- := \Delta_n^- \cup \{t\} \);

return(\( \Delta_n^+, \Delta_n^- \));

\}

7.4.1.2. INTERSECTION

For node \( n \) of the form: (\( \text{R1 INTERSECT R2} \)), \( \Delta_n^+ \) includes:

- all tuples added to \( \text{R1} \) that are in the "new" \( \text{R2} \)
- all tuples added to \( \text{R2} \) that are in the "new" \( \text{R1} \)

\[ \Delta_n^+ = \{ t \in \Delta_{R1}^+ \mid t \in \text{new-R2} \} \cup \{ t \in \Delta_{R2}^+ \mid t \in \text{new-R1} \} \]

\( \Delta_n^- \) includes:

- all tuples removed from \( \text{R1} \) that were in the "old" \( \text{R2} \)
- all tuples removed from \( \text{R2} \) that were in the "old" \( \text{R1} \)

\[ \Delta_n^- = \{ t \in \Delta_{R1}^- \mid t \in \text{old-R2} \} \cup \{ t \in \Delta_{R2}^- \mid t \in \text{old-R1} \} \]
In procedural form:

```plaintext
compute-INTERSECT-changes(Δ⁺_{R1}, Δ⁻_{R1}, Δ⁺_{R2}, Δ⁻_{R2}) {
    Δ⁺_n := {};
    for all t ∈ Δ⁺_{R1} do
        if (R2.new-membership(t))
            then Δ⁺_n := Δ⁺_n ∪ {t};
    for all t ∈ Δ⁺_{R2} do
        if (R1.new-membership(t))
            then Δ⁺_n := Δ⁺_n ∪ {t};
    Δ⁻_n := {};
    for all t ∈ Δ⁻_{R1} do
        if (R2.old-membership(t))
            then Δ⁻_n := Δ⁻_n ∪ {t};
    for all t ∈ Δ⁻_{R2} do
        if (R1.old-membership(t))
            then Δ⁻_n := Δ⁻_n ∪ {t};
    return(Δ⁺_n, Δ⁻_n);
}
```

7.4.1.3. MINUS

For node $n$ of the form: (R1 MINUS R2), $Δ⁺_n$ includes:

- all tuples added to R1 that are not in the "new" R2
- all tuples removed from R2 that are in the "new" R1

$$Δ⁺_n = \{ t ∈ Δ⁺_{R1} | t ∉ new-R2 \} ∪ \{ t ∈ Δ⁻_{R2} | t ∈ new-R1 \}$$

$Δ⁻_n$ includes:

- all tuples removed from R1 that are not in the "old" R2
- all tuples added to R2 that are in the "old" R1
\[ \Delta^-_n = \{ t \in \Delta^-_{R1} \mid t \not\in \text{old-R2} \} \cup \{ t \in \Delta^+_{R2} \mid t \in \text{old-R1} \} \]

In procedural form:

\[
\text{compute-MINUS-changes}(\Delta^+_{R1}, \Delta^-_{R1}, \Delta^+_{R2}, \Delta^-_{R2}) \{
\Delta^+_n := \{};
\text{forall } t \in \Delta^+_{R1} \text{ do}
\quad \text{if } (\neg \text{R2.new-membership}(t))
\quad \quad \text{then } \Delta^+_n := \Delta^+_n \cup \{ t \};
\text{forall } t \in \Delta^-_{R2} \text{ do}
\quad \text{if } (\text{R1.new-membership}(t))
\quad \quad \text{then } \Delta^+_n := \Delta^+_n \cup \{ t \};
\Delta^-_n := \{};
\text{forall } t \in \Delta^-_{R1} \text{ do}
\quad \text{if } (\neg \text{R2.old-membership}(t))
\quad \quad \text{then } \Delta^-_n := \Delta^-_n \cup \{ t \};
\text{forall } t \in \Delta^+_{R2} \text{ do}
\quad \text{if } (\text{R1.old-membership}(t))
\quad \quad \text{then } \Delta^-_n := \Delta^-_n \cup \{ t \};
\text{return}(\Delta^+_n, \Delta^-_n );
\}\n\]

7.4.2. Computing the changes to an IS IN or EQUI-JOIN node

The procedures that compute the changes to an IS IN or an EQUI-JOIN node are the only ones that use the node's children's selective-retrieval procedures. For an IS IN node, only the left child's selective-retrieval procedures are called; the right child's membership-test procedures are used. For an EQUI-JOIN node, both children's selective-retrieval procedures are called; no membership-test procedure is used.
In both cases, the procedure that computes the changes to the node takes advantage of the augmentation to the definition of selective retrieval introduced in section 6.2.1. A call to the node's child's selective-retrieval procedure passes a set of values corresponding to a single field. This means that the number of calls made to an IS IN or EQUI-JOIN node's child's selective-retrieval procedure by the procedure that computes the changes to the node is independent of the sizes of the node's children's change sets.

7.4.2.1. IS IN

For node $n$ of the form: (field $\#i$ of R1 IS IN R2), $\Delta^+_n$ includes:

- all tuples added to R1 whose value for field $\#i$ is in the "new" R2
- for all tuples $\langle v \rangle$ added to R2, the set of tuples in the "new" R1 whose value for field $\#i$ is $v$

$$\Delta^+_n = \{ t \in \Delta^+_R | t.\text{field } \#i \in \text{new-R2} \} \cup \{ t \in \text{new-R1} | t.\text{field } \#i \in \Delta^+_R \}$$

$\Delta^-_n$ includes:

- all tuples removed from R1 whose value for field $\#i$ is in the "old" R2
- for all tuples $\langle v \rangle$ removed from R2, the set of tuples in the "old" R1 whose value for field $\#i$ is $v$

$$\Delta^-_n = \{ t \in \Delta^-_R | t.\text{field } \#i \in \text{old-R2} \} \cup \{ t \in \text{old-R1} | t.\text{field } \#i \in \Delta^-_R \}$$
In procedural form:

\[
\text{compute-IS-IN-changes}(\Delta^+_R, \Delta^-_R, \Delta^+_R, \Delta^-_R) \{
\Delta^+_n := \{\};
\text{forall} \ v \ \text{in field} \ #i \ \text{of} \ \Delta^+_R \ \text{do}
\text{if} (R2.new-membership(v))
\text{then} \ \Delta^+_n := \Delta^+_n \cup \{\text{all tuples of} \ \Delta^+_R \ \text{with value} \ v \ \text{in field} \ #i\};
\Delta^+_n := \Delta^+_n \cup R1.new-selective-retrieval((#i), (\Delta^+_R));
\Delta^-_n := \{\};
\text{forall} \ v \ \text{in field} \ #i \ \text{of} \ \Delta^-_R \ \text{do}
\text{if} (R2.old-membership(v))
\text{then} \ \Delta^-_n := \Delta^-_n \cup \{\text{all tuples of} \ \Delta^-_R \ \text{with value} \ v \ \text{in field} \ #i\};
\Delta^-_n := \Delta^-_n \cup R1.old-selective-retrieval((#i), (\Delta^-_R));
\text{return}(\Delta^+_n, \Delta^-_n);
\}
\]

7.4.2.2. EQUI-JOIN

For node \( n \) of the form:

\((\text{JOIN}(R1, R2) \ \text{WHEN} \ (\text{field} \ #i \ \text{of} \ R1 = \ \text{field} \ #j \ \text{of} \ R2)), \ \Delta^+_n \) includes:

- for every tuple \( t \) added to \( R1 \), the join of \( t \) with all tuples in the "new" \( R2 \) that have the value of \( t.\text{field} \ #i \) in their \( j^{th} \) field
- for every tuple \( u \) added to \( R2 \), the join of all tuples in the "new" \( R1 \) that have the value of \( u.\text{field} \ #j \) in their \( i^{th} \) field, with \( u \)

\[
\Delta^+_n = \{\text{JOIN}(t, u) \mid (t.\text{field} \ #i = u.\text{field} \ #j) \land \\
((t \in \Delta^+_R \land u \in \text{new-R2}) \lor (t \in \text{new-R1} \land u \in \Delta^+_R))\}
\]
\( \Delta_n^- \) includes:

- for every tuple \( t \) removed from \( R1 \), the join of \( t \) with all tuples in the "old" \( R2 \) that have the value of \( t.\text{field } \#i \) in their \( j^{th} \) field

- for every tuple \( u \) removed from \( R2 \), the join of all tuples in the "old" \( R1 \) that have the value of \( u.\text{field } \#j \) in their \( i^{th} \) field, with \( u \)

\[
\Delta_n^- = \{ \text{JOIN}(t, u) \mid (t.\text{field } \#i = u.\text{field } \#j) \land \\
((t \in \Delta_n^{-R1} \land u \in \text{old-R2}) \lor (t \in \text{old-R1} \land u \in \Delta_n^{-R2})) \}
\]

In procedural form:

\[
\text{compute-EQUI-JOIN-changes}(\Delta_n^{+R1}, \Delta_n^{-R1}, \Delta_n^{+R2}, \Delta_n^{-R2}) \{
S := \text{the set of values in field } \#i \text{ of } \Delta_n^{+R1};
\Delta_n^{+} := \text{JOIN}(\Delta_n^{+R1}, R2.\text{new-selective-retrieval}((\#j), (S)));
S := \text{the set of values in field } \#j \text{ of } \Delta_n^{+R2};
\Delta_n^{+} := \text{JOIN}(R1.\text{new-selective-retrieval}((\#i), (S)), \Delta_n^{+R2});
\]

\[
S := \text{the set of values in field } \#i \text{ of } \Delta_n^{-R1};
\Delta_n^{-} := \text{JOIN}(\Delta_n^{-R1}, R2.\text{old-selective-retrieval}((\#j), (S)));
S := \text{the set of values in field } \#j \text{ of } \Delta_n^{-R2};
\Delta_n^{-} := \text{JOIN}(R1.\text{old-selective-retrieval}((\#i), (S)), \Delta_n^{-R2});
\]

\[
\text{return}(\Delta_n^{+}, \Delta_n^{-});
\}
\]

7.4.3. Computing the changes to a CROSS PRODUCT node

For node \( n \) of the form: \( (R1 \text{ CROSS } R2) \), \( \Delta_n^{+} \) includes:

- for every tuple added to \( R1 \), the cross product of that tuple and all tuples in the "new" \( R2 \)

- for every tuple added to \( R2 \), the cross product of that tuple and all tuples in the "new" \( R1 \)
\[ \Delta_n^+ = \{ <t, u> \mid (t \in \Delta_{R1}^+ \land u \in \text{new-R2}) \lor (t \in \text{new-R1} \land u \in \Delta_{R2}^+) \} \]

\[ \Delta_n^- = \{ <t, u> \mid (t \in \Delta_{R1}^- \land u \in \text{old-R2}) \lor (t \in \text{old-R1} \land u \in \Delta_{R2}^-) \} \]

\( \Delta_n^- \) includes:

- for every tuple removed from R1, the cross product of that tuple and all tuples in the “old” R2
- for every tuple removed from R2, the cross product of that tuple and all tuples in the “old” R1

CROSS PRODUCT is the only relational operator whose operand relations must be maintained in order to allow incremental view updating. This is because the change to the relation represented by the CROSS-PRODUCT node caused by a change to one of its operand relations involves all the tuples in the other operand relation.

The set of tuples to be removed from the cross product is computed first, using the maintained operand relations R1 and R2; the operand relations are then updated using \( \Delta_{R1}^+ \), \( \Delta_{R1}^- \), \( \Delta_{R2}^+ \), and \( \Delta_{R2}^- \), and the set of tuples to be added to the cross product is computed.
compute-CROSS-PRODUCT-changes(Δ_{R1}^+, Δ_{R1}^-, Δ_{R2}^+, Δ_{R2}^-) {
    Δ_n^- := Δ_{R1}^- × R2;
    Δ_n^+ := Δ_n^- ∪ (R1 × Δ_{R2}^-);

    R1 := (R1 ∪ Δ_{R1}^+) - Δ_{R1}^-;
    R2 := (R2 ∪ Δ_{R2}^+) - Δ_{R2}^-;

    Δ_n^+ := Δ_{R1}^+ × R2;
    Δ_n^- := Δ_n^+ ∪ (R1 × Δ_{R2}^+);

    return(Δ_n^+, Δ_n^-);
}

7.4.4. Computing the changes to a SELECT node

For node \( n \) of the form: \( (\text{SELECT}(\text{condition}) \text{ FROM } R) \), \( Δ_n^+ \) includes:

- all tuples added to \( R \) that satisfy the SELECT node's condition

  \[ Δ_n^+ = \{ t ∈ Δ_R^+ | \text{satisfies}(t, \text{condition}) \} \]

\( Δ_n^- \) includes:

- all tuples removed from \( R \) that are in the “old” relation represented by \( n \);
  i.e. all tuples removed from \( R \) that satisfy the SELECT node’s condition

  \[ Δ_n^- = \{ t ∈ Δ_R^- | \text{satisfies}(t, \text{condition}) \} \]

In procedural form:

compute-SELECT-changes(Δ_R^+, Δ_R^-) {
    Δ_n^+ := {};
    \text{forall } t ∈ Δ_R^+ \text{ do}
        \text{if } (\text{satisfies}(t, \text{condition}))
            \text{then } Δ_n^+ := Δ_n^+ ∪ \{ t \};
\[ \Delta_n := \{\}; \]
\[
\text{forall } t \in \Delta_R^- \text{ do}
\]
\[
\quad \text{if } (\text{satisfies}(t, \text{condition}))
\]
\[
\quad \text{then } \Delta_n^- := \Delta_n^- \cup \{t\};
\]
\[
\text{return} (\Delta_n^+, \Delta_n^-);
\]

7.4.5. Computing the changes to a PROJECT node

For node \(n\) of the form: (PROJECT list-of-fields FROM \(R\)), \(\Delta_n^+\) includes:

- for every tuple \(t\) added to \(R\) such that there is no tuple in the "old" \(R\) that matches \(t\) on all fields in list-of-fields, the projection: \(\Pi_{\text{list-of-fields}}(t)\)

\[
\Delta_n^+ = \{\Pi_{\text{list-of-fields}}(t) | t \in \Delta_R^+ \land \exists u \in \text{old-R such that} \Pi_{\text{list-of-fields}}(u) = \Pi_{\text{list-of-fields}}(t)\}
\]

\(\Delta_n^-\) includes:

- for every tuple \(t\) removed from \(R\) such that there is no tuple in the "new" \(R\) that matches \(t\) on all fields in list-of-fields, the projection: \(\Pi_{\text{list-of-fields}}(t)\)

\[
\Delta_n^- = \{\Pi_{\text{list-of-fields}}(t) | t \in \Delta_R^- \land \exists u \in \text{new-R such that} \Pi_{\text{list-of-fields}}(u) = \Pi_{\text{list-of-fields}}(t)\}
\]

In procedural form:

\[
\text{compute-PROJECT-changes}(\Delta_R^+, \Delta_R^-) \{
\Delta_n^+ := \{\};
\Delta^+_{\text{-projection}} := \Pi_{\text{list-of-fields}}(\Delta_R^+);
\text{forall } t \in \Delta^+_{\text{-projection} \text{ do}}
\]
\[
\quad t' := t \text{ with wildcard values in all fields of } R \\
\quad \text{not included in list-of-fields;}
\]
\[
\quad \text{if } (\neg \text{R.old-membership}(t'))
\]
\[
\quad \text{then } \Delta_n^+ := \Delta_n^+ \cup \{t\};
\]
\( \Delta_n^- := \{\}; \)
\( \Delta^-\)-projection := \( \Pi_{\text{list-of-fields}}(\Delta_R^-); \)
\textbf{forall} \( t \in \Delta^-\)-projection \textbf{do}
\[ t' := t \text{ with wildcard values in all fields of } R \]
\[ \text{not included in list-of-fields}; \]
\[ \text{if } (\neg R.\text{new-membership}(t')) \]
\[ \text{then } \Delta_n^- := \Delta_n^- \cup \{t\}; \]
\[ \text{return}(\Delta_n^+, \Delta_n^-); \]
\}

7.5. Incremental-update example

To illustrate the incremental view updating process, reconsider the example program of figure 3.1a and the corresponding example view definition of figure 3.3, reproduced below.

(1) PROCEDURE main() {
(2) \hspace{1em} PRINT(sum())
(3) \hspace{1em} }
(4) PROCEDURE sum(x, y, z) {
(5) \hspace{1em} a := x + y
(6) \hspace{1em} RETURN(a)
(7) \hspace{1em} }

Example program of figure 3.1a
Changing line (5) of the program to:

\[ a := y + z \]

does not affect the DEFINITION relation, but does cause the tuple \(< (4), z >\) to be added to the LIVE relation, and the tuple \(< (4), x >\) to be removed from the LIVE relation. Using the notation of this section:

\[ \Delta^+_{\text{LIVE}} = \{ < (4), z > \} \]

\[ \Delta^-_{\text{LIVE}} = \{ < (4), x > \} \]

Changes to the UNUSED_DEFINITION view are computed using the changes to the LIVE relation and the procedures DEFINITION.new_membership and DEFINITION.old_membership:
\[ \Delta^+_{\text{MINUS}} = \{ t \in \Delta^-_{\text{LIVE}} \mid t \in \text{new-DEFINITION} \} \]
\[ = \{ < (4), x > \} \]

\[ \Delta^-_{\text{MINUS}} = \{ t \in \Delta^+_{\text{LIVE}} \mid t \in \text{old-DEFINITION} \} \]
\[ = \{ < (4), z > \} \]

The values \( \Delta^+_{\text{MINUS}} \) and \( \Delta^-_{\text{MINUS}} \) are used to update the \text{UNUSED-DEFINITION} view; the tuple \( < (4), x > \) is inserted, and the tuple \( < (4), z > \) is deleted. The cost of recomputing the view from scratch, even using the evaluation method of chapters 5 and 6, would be proportional to the size of the entire \text{DEFINITION} relation. By contrast, the cost of the view update is proportional to the sum of the sizes of the changes to the \text{LIVE} relation and the changes to the view relation.

In this particular example, there are no internal nodes in the view-definition tree. When a view-definition tree does have internal nodes, the cost of updating the view depends in part on which of the relations represented by the internal nodes are maintained. The cost of updating such views is discussed in the following section.

7.6. The cost of incremental view updating

The cost of incremental view updating has four components:

1. When a view is first defined, membership-test and selective-retrieval procedures are built for all nodes of the view-definition tree. This is done only once for each view, and has cost proportional to the size of the tree.
(2) When a view update is required, for each leaf relation \( l \) all tuples in \( \Delta_l^+ \) are added to \( l \) and marked 'new'; all tuples in \( \Delta_l^- \) are marked 'deleted'. This has cost proportional to the number of changes made to leaf relations.

(3) Sets \( \Delta^+ \) and \( \Delta^- \) are computed for all internal nodes and for the root node of the view-definition tree. The cost of computing these sets is discussed below.

(4) When the sets \( \Delta^+ \) and \( \Delta^- \) have been computed for the root node, the view relation is updated; all tuples in the sets \( \Delta_l^+ \) are unmarked, and all tuples in the sets \( \Delta_l^- \) are deleted. These operations have costs proportional to the size of the change to the view relation and to the number of changes made to leaf relations, respectively.

To summarize: the cost of incrementally updating a view, given membership-test and selective-retrieval procedures for each node of the view-definition tree and given sets \( \Delta_l^+ \) and \( \Delta_l^- \) for each leaf \( l \) of the tree is:

\[
\sum_{l \text{ a leaf}} (\text{size}(\Delta_l^+) + \text{size}(\Delta_l^-)) + \sum_{n \text{ an internal or root node}} (\text{cost(computing } \Delta_n^+ \text{ and } \Delta_n^-)) + \text{size(change to view)}
\]

The costs of computing the sets \( \Delta_n^+ \) and \( \Delta_n^- \) depend on the costs of the membership-test and selective-retrieval procedures of the nodes of the view-definition tree. These in turn depend on which intermediate relations are maintained and what access is provided to maintained relations. The costs associated with various access methods were discussed in chapter 5; costs are lowest when hashed access is provided. In this case, calling the membership-test procedure of a maintained relation with an argument that includes no wildcard values has a constant cost; calling the selective-retrieval procedure of a maintained relation
has a cost proportional to the size of the answer set. The origin of a call to a
membership-test procedure with an argument that includes wildcard values is a
PROJECT node. The wildcard values in the argument will be in the fields not
included in the PROJECT node's list-of-fields. If the operands of PROJECT
nodes are maintained, constant-time membership-test procedures can be pro-
vided by storing these relations hashed on the fields of the parent PROJECT
node's list-of-fields. We will assume that the appropriate hashed access is pro-
vided for all maintained relations, and that all leaf relations are maintained.

If all intermediate relations are maintained, all membership-test procedures
have constant cost, and all selective-retrieval procedures have cost proportional
to the size of the answer set. The cost of computing $\Delta^+_n$ and $\Delta^-_n$ for any node $n$
is proportional to the sizes of the changes to the relations represented by $n$'s
children, and to the sizes of $\Delta^+_n$ and $\Delta^-_n$. The cost of updating a view with
definition tree $d$ is:

$$O \left( \sum_{n \in d} \left( \text{size}(\Delta^+_n) + \text{size}(\Delta^-_n) \right) \right)$$

the sum of the sizes of the changes to all relations represented by the nodes of
the view definition: leaf relations, intermediate relations, and the view relation.

Although one would prefer an incremental view-updating algorithm with
cost independent of the sizes of changes to intermediate relations, the observa-
tion due to [Chandra and Newey 1985] given below indicates the unlikelihood of
such an algorithm. In this observation, we show how a simple view definition
with one intermediate relation can be used to model Boolean matrix multiplication. A scenario is constructed to show that if it were possible to update this view in time independent of the size of the change to the intermediate relation, it would also be possible to multiply two \( n \times n \) Boolean matrices in time \( O(n^2) \). Since the best known method for multiplying two \( n \times n \) Boolean matrices requires time \( O(n^{2.49}) \) [Coppersmith and Winograd 1982], it is unlikely that general views can be updated in time independent of the sizes of changes to the intermediate relations represented by the internal nodes of the view definition tree.

**Observation**

Boolean matrix \( M \) can be represented as a binary relation: tuple \(<i, j>\) is in the relation iff \( M[i, j] = 1 \). Boolean matrix multiplication can be modelled as a join followed by a projection: given that the relational representation of \( M_1 \) is \( R_1 \), and the relational representation of \( M_2 \) is \( R_2 \), the relational representation of \( (M_1M_2) \) is:

\[
II_{5,4}(\text{JOIN}_{R_1,\text{field } 5 = R_2,\text{field } 4}(R_1, R_2))
\]

The projection of the first and last fields of the join of relations \( R_1 \) and \( R_2 \) forms a simple view definition with one intermediate value (the result of the join). When \( R_1 = \{\} \), and \( R_2 = \text{relational-representation}(M_2) \), the view is empty. Adding \( O(n^2) \) tuples to \( R_1 \) so that \( R_1 = \text{relational-representation}(M_1) \), produces a view relation that is the relational representation of \( (M_1M_2) \). The view relation has at most \( n^2 \) tuples; thus, the sum
of the size of the change to the underlying database and the change to the view relation is \( O(n^2) \).

Although maintaining all intermediate relations provides fast view updating, this approach may be impractical because of its space requirements. Maintaining fewer intermediate relations requires less space, but requires more time for view updating. The following subsections consider the costs of membership testing and selective retrieval, and the resulting costs of view updating, as fewer and fewer intermediate relations are maintained.

7.6.1. Maintaining the operands of IS IN, EQUI-JOIN, PROJECT, and CROSS-PRODUCT nodes

We begin by assuming that the operands of all IS IN, EQUI-JOIN, PROJECT, and CROSS-PRODUCT nodes are maintained. The next two paragraphs discuss the worst-case costs of calls to a node's membership-test and selective-retrieval procedures under this assumption.

Membership-test procedure calls with wildcard values in their arguments originate at PROJECT nodes. When the operands of PROJECT nodes are maintained, they are used to answer these membership questions, and no further calls with wildcard values in their arguments are made. The membership-test procedures of all other nodes, called with wildcard-value-free arguments, at worst call all their children's membership-test procedures; thus, the worst-case
cost of a call to the membership-test procedure at node $n$ is proportional to the
size of the subtree rooted at $n$.

The only selective-retrieval procedures called directly during view updating
are those associated with the operands of EQUI-JOIN nodes, and the left
operands of IS IN nodes. When these relations are maintained, the cost of cal-
ing a selective-retrieval procedure during view updating is proportional to the
size of the answer set. The size of the answer set is at most the size of the
change to the relation represented by the EQUI-JOIN or IN IS node, which is
already included in the cost of view updating. When the operands of IS IN,
EQUI-JOIN, PROJECT and CROSS-PRODUCT nodes are maintained, the
worst-case cost of updating a view with definition tree $d$ is:

$$O\left(\text{size}(d) \times \sum_{n \in d} (\text{size}(\Delta^+_n) + \text{size}(\Delta^-_n))\right)$$

the size of the definition tree times the sum of the sizes of the changes to all
relations represented by the nodes of the tree.

7.6.2. **Maintaining the operands of PROJECT and CROSS-PRODUCT
nodes**

When the operands of IS IN and EQUI-JOIN nodes are not maintained, a
change to an operand of one of these nodes can cause the selective-retrieval pro-
cedures of the nodes in the subtree rooted at the IS IN or EQUI-JOIN node to
be called. Looking back at section 6.2 we see that for all nodes except CROSS-
PRODUCT nodes, selective-retrieval can be implemented by calling each child's
selective-retrieval procedure and combining the results of the calls. A CROSS-
PRODUCT node’s selective-retrieval procedure may have to call one child’s
relation-producing procedure, however, as we are assuming that the operands of
CROSS-PRODUCT nodes are maintained, this is not a problem. The worst-case
cost of calling node n’s selective-retrieval procedure is thus:

\[ \text{cost(calling n’s children’s selective-retrieval procedures)} + f_n (\text{sizes of the sets returned by n’s children’s selective-retrieval procedures}) \]

The function \( f_n \) depends on the particular operator at node n. For an INTER-
SECTION node, for example, \( f_n \) is the cost of computing the intersection of
two sets as a function of their sizes. For a SELECT node, \( f_n \) is the cost of
checking the tuples of a set to see if they satisfy the SELECT node’s condition,
again a function of the size of the set. Assuming reasonable implementations of
the operations performed on the sets returned by the node’s children’s selective-
retrieval procedures, \( f_n \) should be no worse than \( s \log s \), where \( s \) is the sum
of the sizes of the sets returned by the node’s children’s selective-retrieval pro-
cedures. The sizes of these sets can be no larger than the size \( S \) of the largest
relation represented by a node of the view-definition tree.

The procedure compute-IS-IN-changes, given in section 7.4.2.1, calls the IS
IN node’s left child’s selective-retrieval procedure twice; the additional cost of a
single call to compute-IS-IN-changes when its left operand is not maintained is
at worst:
\( O(\text{size(subtree rooted at IS IN node's left child}) * (S \log S)) \),

where \( S \) is, as above, the size of the largest relation represented by a node of the view-definition tree.

The procedure compute-EQUI-JOIN-changes calls each of the EQUI-JOIN node's children's selective-retrieval procedures twice; the additional cost of a single call to compute-EQUI-JOIN-changes when neither of its operands is maintained is at worst:

\( O(\text{size(subtree rooted at EQUI-JOIN node}) * (S \log S)) \).

The total cost added to view updating when the operands of IS IN and EQUI-JOIN nodes are not maintained is:

\[
\left( \sum_{n \text{ an IS-IN node}} \text{(size(subtree rooted at n's left child))} \right) + \\
\left( \sum_{n \text{ an EQUI-JOIN node}} \text{(size(subtree rooted at n))} \right) * (S \log S)
\]

**7.6.3. Maintaining only the operands of CROSS-PRODUCT nodes**

When the operands of PROJECT nodes are not maintained, some membership-test procedures are called with wildcard-valued arguments in the course of view updating. These membership-test procedures may be forced to call their children's selective-retrieval procedures. This can, in the worst case, add a *factor* of:

\[
\left( \sum_{n \text{ a PROJECT node}} \text{(size of subtree rooted at n)} \right) * (S \log S)
\]

to the cost of view updating.
7.6.4. The cost of incremental view updating: conclusions

The cost of updating a view depends on which intermediate values are maintained between updates. The results of the previous sections indicate that at least the operands of CROSS-PRODUCT and PROJECT nodes should be maintained to provide a reasonable view-update time. However, the cost estimates given in the previous sections are worst-case costs, and are based on the assumption that, for each relational operator, either all or none of the instances of that operator's operand relations in the view-definition tree are maintained. In practice, the individual cost estimates computed for each node's membership-test and selective-retrieval procedures would be used to decide whether to maintain the relation represented by that node. Further, an advantage of the incremental view-update method presented in this chapter is that the tradeoff between time and space consumption can be tuned dynamically: all intermediate values can be stored as long as there is sufficient space to do so. When space consumption becomes a problem, the less-crucial intermediate values can be deleted, to be recreated when and if the necessary space is once more available.

7.7. An open problem

If incremental view updating is to be used successfully in the context of editing environments as described in this thesis, the algorithm presented in this chapter must be adapted to handle views that use implicit relations. Some implicit relations are subject to extensive changes following small editing operations. The ANCESTOR relation is an example: adding or deleting a program subtree of size
$n$ adds or deletes $O(n^2 + n \times (\text{depth of subtree}))$ tuples to the ANCESTOR relation. Computing the actual changes to the relation after every editing operation, and using these values as input to the incremental view-updating algorithm is obviously undesirable.

A possible solution is to choose an internal node $n$ of the view-definition tree found on the path from the use of the ANCESTOR relation to the root of the tree, and to designate $n$ as a *virtual leaf*. This means that for the purposes of incremental view updating, node $n$ is considered to be a leaf of the view-definition tree: the relation represented by $n$ is maintained, and is recomputed after each editing operation; the changes to this relation are input to the incremental view-updating algorithm, which is unaware of the existence of the subtree rooted at $n$.

The details of this solution, for example, how to choose node $n$, and the possibility of discovering other solutions to the problem of incrementally updating views that use implicit relations, are left as areas for future research.

### 7.8. Relation to previous work

Very little work has been done on incremental view updating. Our approach is most similar to that of [Koenig and Paige 1981], which applies the method of finite differencing [Paige and Koenig 1982] to view updating. A comparison of the two methods follows.
The method of Paige and Koenig is geared toward set and arithmetic operators, while our method is geared toward the relational operators, which include set but not arithmetic operators. During view updating, selective retrieval is used only for EQUI-JOIN and IS IN operators; since these are neither set nor arithmetic operators, selective retrieval is not used by Paige and Koenig. Membership testing is used, though never formalized. The idea of providing membership-test procedures for relations that are never actually built was first discussed in [Horwitz and Teitelbaum 1985], which postdates both [Koenig and Paige 1981] and [Paige and Koenig 1982]; [Horwitz and Teitelbaum 1985] also originates the idea of using such procedures during view updating to avoid maintaining intermediate relations.

The method of Paige and Koenig handles changes consisting of a single tuple insertion or deletion; a change is propagated all the way to the view relation before the next change is considered. This means that although two changes may cancel at some intermediate relation, they are not given the opportunity to do so. By contrast, our method can handle sets of tuple insertions and deletions as well as single changes. Because changes are propagated in postorder, changes can cancel before reaching the view relation.

A more recent method for incremental view updating is proposed in [Shmueli and Itai 1984]. This method is very limited; natural join is the only operator permitted in a view definition. Although this method appears at first to be very different from ours, it would have some similarity to a version of our
method in which EQUI-JOIN was the only operator allowed and all operand relations were maintained.

The basic difference between the way the two methods handle joins is that we maintain the operands of joins directly as relations, while Shmueli and Itai maintain structures that include pointers to tuples of leaf relations, which can be used to build the operands of joins as needed. Maintaining pointers has a lower space overhead than maintaining actual tuples; however, when a change to the underlying database causes tuples to be added to the view relation, building the new tuples using the structures of Shmueli and Itai will often be more expensive than building the new tuples using our maintained tuples.

Another difference between the two methods is that while the method of Shmueli and Itai is more complicated and expensive for cyclic queries than for tree queries [Bernstein and Chiu 1981], our method is the same for both types of query.

Other work that is relevant to the work described here is that of [Severance and Lohman 1976] and [Stonebraker and Keller 1980]. These papers discuss the use of differential files to record changes to a database, rather than actually updating the database. When view updates are not made after every tuple insertion or deletion, sets $\Delta^+$ and $\Delta^-$ must be maintained for each changed leaf relation. This could be done using differential files.
Chapter 8

Implementation issues

The first part of this thesis proposed a language-independent model of editing in which a program is represented as an attributed abstract-syntax tree with an associated relational database. The advantages of this model were discussed.

The second part of the thesis presented new algorithms for query-evaluation and incremental view updating, motivated by the efficiency needs of an editor based on the model of part one. This chapter, the third part of the thesis, addresses the practical problems associated with an actual implementation based on the results of parts one and two.

We begin by describing our prototype implementation of an editor generator. The topics covered are:

1. The specification of locally-derived and dataflow relations.
2. Maintaining locally-derived relations at edit time.
3. Performing dataflow analysis at edit time.
4. The internal representation of relations.
5. Writing and executing queries.
6. Representing and using program points in relations.

We then present a number of practical problems that were outside the scope of our prototype implementation. Possible solutions are given for some of these problems; some are presented as areas for further research.
8.1. A prototype implementation of an editor generator

Our prototype implementation does not provide a complete editing environment as described in part one of this thesis. Generated editors do support:

(1) Locally-derived relations
(2) Dataflow relations
(3) Unconstrained relations created through query execution
(4) Writing and executing queries using explicit relations

Generated editors do not support:

(1) View relations
(2) Unconstrained relations created by some means other than through query execution
(3) The use of implicit relations in queries

The presentation of the implementation of our prototype editor generator is organized as follows:

Sections 8.1.1 and 8.1.2 describe how an editor designer specifies the locally-derived and dataflow relations that are to be maintained by the generated editor.

Our prototype editor generator is built on top of the Synthesizer Generator, a generator of pure attribute-grammar-based editors. Section 8.1.3 covers the augmentations that were made to the Synthesizer Generator to allow the edit-time support of locally-derived and dataflow relations.

Sections 8.1.4 through 8.1.6 discuss several aspects of the prototype system that are relation specific, but that did not require augmenting the Synthesizer Generator.

8.1.1. Specifying locally-derived relations

The declaration of a locally-derived relation gives its name and the names and types of its fields. For example, the declaration of the locally-derived CALLS
relation was given in figure 4.1c:

**locally-derived relation** CALLS("call from" STRING, "call to" STRING);

The tuples of locally-derived relations are defined on a per-production basis using tuple membership assertions, as defined in chapter 4. A tuple membership assertion includes a defining expression for each field of the tuple; expressions can use the tokens, program points, and attribute values of the production. Again from figure 4.1c:

\[
\textit{stmt} \rightarrow \text{"CALL" IDENT "(" argument_list ")"}
\]

\[
\{ \langle \text{stmt.proc\_name, IDENT} \rangle \in \text{CALLS}; \}
\]

means that for every instance of the derivation: \((\text{stmt} \rightarrow \text{"CALL" IDENT "(" argument_list ")")\)}\) in the program tree, the CALLS relation shall include a tuple whose fields have the values of the \(\text{stmt.proc\_name}\) attribute and the IDENT token associated with that derivation instance.

### 8.1.2. Specifying dataflow relations

Dataflow problems are defined in terms of the lattice model of [Kildall 1973]. The declaration of a dataflow problem includes the name of the problem, its direction (forward or backward), a type declaration used for associating functions with control-flow edges, and the names of procedures to do the following:

1. test two lattice elements for equality
2. compute the meet of two lattice elements
3. construct the lattice top
4. construct the initial value to be associated with the root node of the control-flow graph
(5) apply the function associated with an edge of the control-flow graph to a lattice element

(6) build the dataflow relation from the results of dataflow analysis

The purpose of the sixth procedure is to convert the results of dataflow analysis to a relation in first normal form (a relation is said to be in first normal form when the types of all fields are atomic). The objective of dataflow analysis is to assign to every program point a fact that will be true at that point in all possible executions; conceptually, the result of dataflow analysis is a relation with tuples of the form: (program point, fact). The fact is often a set; for example, for live variable analysis, facts are the sets of variables that are live at the corresponding program point; for constant propagation, facts are the sets of pairs of variable names and the values they will always have at the corresponding program point. The relational operators treat fields of tuples as indivisible objects; thus, the results of dataflow analysis must be put into first normal form before being used in queries or views. The sixth procedure supplied as part of a dataflow problem definition takes as parameters a program point and the corresponding fact and returns a set of tuples that are to be part of the dataflow relation. Figure 8.1 uses the example program introduced in chapter 3 to illustrate the difference between dataflow analysis results and the corresponding dataflow relation.
(1) PROCEDURE main() {
(2)     PRINT(sum())
(3) }
(4) PROCEDURE sum(x, y, z) {
(5)     a := x + y
(6)     RETURN(a)
(7) }

Figure 8.1a
Example program

<(4), \{x, y\}>
<(5), \{a\}>

Figure 8.1b
Results of live variable analysis

<(4), x>
<(4), y>
<(5), a>

Figure 8.1c
The LIVE dataflow relation

The procedures named as part of the declaration of a dataflow problem must be supplied as part of the definition of the dataflow problem. In addition, definitions of tuples for the relations CONTROL FLOW, ROOT, and EDGE FUNCTION must be supplied. The information in CONTROL FLOW allows a control-flow graph to be built for every program derivable by the generated editor. The information in ROOT allows the roots of the control-flow
graphs to be identified. The information in `EDGE_FUNCTION` allows dataflow functions to be associated with the edges of the control-flow graph.

The control-flow graph of a program depends only on the semantics of the programming language, and not on the particular dataflow problems to be solved; thus, the tuples of the `CONTROL_FLOW` and `ROOT` relations are defined independently of the particular dataflow problems chosen by the editor designer. The functions associated with the edges of the control-flow graph, however, are different for each dataflow problem; thus, the tuples of the `EDGE_FUNCTION` relation must be defined for each dataflow problem chosen by the editor designer. The first field of the `EDGE_FUNCTION` relation identifies the dataflow problem for which the given function is associated with the given edge.

### 8.1.2.1. `CONTROLFLOW`

Dataflow problems are solved using a control-flow graph representation of the program. The locally-derived relation:

```
CONTROLFLOW( "from" control-flow node, "to" control-flow node )
```

is used to define the edges of the control-flow graph. Each control-flow node corresponds to a node of the abstract-syntact tree. This correspondence could be handled in a number of ways. One possibility is to include a single node in the control-flow graph for each of a subset of the nodes of the abstract-syntact tree, for example, the leaves of the tree. This approach has the advantage of defining
a compact control-flow graph, but has the disadvantage of requiring the use of attributes to pass the program points of some tree nodes to the nodes that correspond to their predecessors or successors in the control-flow graph.

In our prototype implementation we have chosen a different approach, following the model of [Rosen 1977]: each node $n$ in the abstract-synta tree has two corresponding nodes, entering($n$) and leaving($n$), in the control-flow graph. ("Entering" and "leaving" are thus functions from program points to control-flow nodes.) Control-flow graphs defined under this model are larger than necessary, however, the specification of local control flow does not require the use of attributes. Figure 8.2 shows the control flow associated with the production:

$$stmt \rightarrow stmt \ ";" \ stmt$$

under this model; tree edges are shown as dashed lines, control-flow edges as solid arrows.

Figure 8.2
Control-flow example
This control flow is specified as follows:

\[ stmt \rightarrow stmt ;; stmt \]
\[ \{ \langle entering(\text{program-point}(stmt1))\rangle, \]
\[ \langle entering(\text{program-point}(stmt2))\rangle \in \text{CONTROL}_FLO\]
\[ \langle leaving(\text{program-point}(stmt2))\rangle, \]
\[ \langle entering(\text{program-point}(stmt3))\rangle \in \text{CONTROL}_FLO\]
\[ \langle leaving(\text{program-point}(stmt3))\rangle, \]
\[ \langle leaving(\text{program-point}(stmt1))\rangle \in \text{CONTROL}_FLO \} \]

Specifying non-local control flow requires the use of attributes. For example, for a language that allows statements of the form: GOTO label, the program point associated with the label must be known at the GOTO statement in order to specify the associated control flow. Set-valued attributes whose elements are pairs of the form: \langle label, program point \rangle, can provide this information. The specification of control flow for the GOTO statement is given in figure 8.3; for the purposes of this example, we assume that:

1. Every stmt nonterminal has a set-valued attribute called ‘label prog pt’ as described above.

2. The function ‘member’ takes as arguments a label \( l \), and a set \( s \) of pairs of the form: \langle label, program point \rangle. The function returns ‘true’ if \( l \) is in some pair in \( s \), and otherwise returns ‘false’.

3. The function ‘label lookup’ takes the same arguments as ‘member’, and returns the program point associated with \( l \) in \( s \).
\[ \text{stmt} \rightarrow \text{"GOTO" IDENT} \\
\quad \{ \text{if member(IDENT, stmt.label prog pt)} \\
\quad \text{then } \langle \text{entering(program-point(stmt))}, \\
\quad \text{entering(label lookup(IDENT, stmt.label prog pt))} \\
\quad > \in \text{CONTROL FLOW}; \} \]

Figure 8.3
Specifying non-local control flow

8.1.2.2. ROOT

Identification of the root node of a flowgraph is a prerequisite for dataflow analysis. Which node is the root of a flowgraph depends on whether forward or backward analysis is done. Given the specification of control flow described above, there will be some node \( n \) of the program tree such that entering\((n)\) is the root for forward problems and leaving\((n)\) is the root for backward problems. The flowgraph root is specified simply as \( n \); entering\((n)\) or leaving\((n)\) will be used as appropriate. Remember that a single ROOT relation is maintained for the entire editing environment. Although there can be only one tuple membership assertion for the ROOT relation, the relation will contain a tuple for every program in the environment. Figure 8.4 shows an example specification of a flowgraph root.

\[ \text{program} \rightarrow \text{"PROCEDURE" IDENT \(" (" parameter-list ")\) \{" stmt \"\}\} \\
\quad \{ <\text{program}> \in \text{ROOT}; \} \]

Figure 8.4
Specifying the flowgraph root
8.1.2.3. EDGE_FUNCTION

Each dataflow problem associates a function with every edge of the control-flow graph. The definition of a dataflow problem must thus include the association of functions with edges. Again, this association is made on a per-production basis by defining tuples of the EDGE_FUNCTION relation. The first field of a tuple of EDGE_FUNCTION is of type STRING, and gives the name of the dataflow problem for which an edge/function association is being made. The next two fields are of type control-flow node, and specify the edge. The type of the fourth field, which specifies the function, is the union of all the types given for this purpose in all dataflow problem declarations. Figure 8.5a specifies the control flow associated with two grammar productions; figure 8.5b associates functions for live variable analysis with all edges for which the associated function is not the identity function; figure 8.5c shows control flow and functions pictorially.
\[ stmt \rightarrow \text{IDENT } "::=" \ exp \\
\begin{array}{l}
\{ <\text{entering(program-point}(stmt))>, \\
\quad \text{entering(program-point}(exp))> \in \text{CONTROL}_FLO\_W; \\
\quad <\text{leaving(program-point}(exp))>, \\
\quad \text{entering(program-point}(\text{IDENT}))> \in \text{CONTROL}_FLO\_W; \\
\quad <\text{entering(program-point}(\text{IDENT}))>, \\
\quad \text{leaving(program-point}(\text{IDENT}))> \in \text{CONTROL}_FLO\_W; \\
\quad <\text{leaving(program-point}(\text{IDENT}))>, \\
\quad \text{leaving(program-point}(\text{stmt}))> \in \text{CONTROL}_FLO\_W; \\
\} \\
\end{array} \\
\]
\[ exp \rightarrow \text{IDENT} \\
\begin{array}{l}
\{ <\text{entering(program-point}(exp))>, \\
\quad \text{entering(program-point}(\text{IDENT}))> \in \text{CONTROL}_FLO\_W; \\
\quad <\text{entering(program-point}(\text{IDENT}))>, \\
\quad \text{leaving(program-point}(\text{IDENT}))> \in \text{CONTROL}_FLO\_W; \\
\quad <\text{leaving(program-point}(\text{IDENT}))>, \\
\quad \text{leaving(program-point}(\text{exp}))> \in \text{CONTROL}_FLO\_W; \\
\} \\
\end{array} \\
\]

Figure 8.5a
Specifying control flow
stmt → IDENT ":=" exp
{ "LIVE",
  entering(program-point(IDENT)),
  leaving(program-point(IDENT)),
  λ live_set. live_set = {IDENT}
  > ∈ EDGE_FUNCTION;
}

exp → IDENT
{ "LIVE",
  entering(program-point(IDENT)),
  leaving(program-point(IDENT)),
  λ live_set. live_set ∪ {IDENT}
  > ∈ EDGE_FUNCTION;
}

Figure 8.5b
Specifying functions associated with control-flow edges
Although the purpose of the \textsc{Edge}\_\textsc{Function} relation is to associate functions with edges, the value of its fourth field need not be an actual function. For example, for live variable analysis, the function associated with edge $e$ of the flowgraph is always of the form:

$$
\lambda \text{live}\_\text{set}.(\text{live}\_\text{set} - \text{defs}_e) \cup \text{uses}_e
$$

where "\text{defs}_e" is the set of variables given values when control flows along edge $e$, and "\text{uses}_e" is the set of variables whose values are used when control flows
along edge $e$. All that really needs to be associated with edge $e$ in the
EDGE_FUNCTION relation are the sets $\text{def}_e$ and $\text{use}_e$. Figure 8.6 repeats
the specification of figure 8.5b, defining the fourth field of EDGE_FUNCTION
as the pair $<\text{def}_e, \text{use}_e>$ rather than as the actual function for edge $e$.

\[
\text{stmt} \rightarrow \text{IDENT} \quad "\cdot:=" \quad \text{exp} \\
\{ \text{"LIVE"}, \\
\text{entering}(\text{program-point}(\text{IDENT})), \\
\text{leaving}(\text{program-point}(\text{IDENT})), \\
<\{\text{IDENT}\}, \{\}> \\
> \epsilon \text{ EDGE_FUNCTION; } \}
\]

\[
\text{exp} \rightarrow \text{IDENT} \\
\{ \text{"LIVE"}, \\
\text{entering}(\text{program-point}(\text{IDENT})), \\
\text{leaving}(\text{program-point}(\text{IDENT})), \\
<\{\}, \{\text{IDENT}\}> \\
> \epsilon \text{ EDGE_FUNCTION; } \}
\]

Figure 8.6
An alternate function specification

The preceding observations lead to the idea of built-in dataflow problems.
When the functions associated with a dataflow problem can be parameterized in
a straightforward way, as in the above example, the dataflow problem can be
largely predefined. The editor specification must select the desired built-in
dataflow problem by name, and must define the tuples of the
CONTROL_FLOW, ROOT, and EDGE_FUNCTION relations, using a
predefined type for the fourth field of EDGE_FUNCTION tuples. The direction
of the dataflow problem, and all the procedures usually included in a dataflow
problem definition are predefined.
8.1.3. Maintaining locally-derived and dataflow relations

Sections 8.1.1 and 8.1.2 above described how an editor designer specifies the locally-derived and dataflow relations that are to be maintained by the editing environment. Actually maintaining these relations at edit time was outside the scope of the original Synthesizer Generator. This section describes how the Synthesizer Generator was augmented to support the maintenance of locally-derived and dataflow relations.

One of our goals in augmenting the Synthesizer Generator was to make the changes as non-specific to relations as possible, so that the augmentations might prove useful in supporting as yet unthought-of enhancements to editing environments. The problems of maintaining locally-derived and dataflow relations were thus viewed as specific examples of more general problems.

8.1.3.1. External stores

The values of tuples of locally-derived relations are defined on a per-production basis, depending on the values of the tokens, program points, and attributes of the production; thus, a tuple is essentially a local attribute*. The same mechanism that insures that attribute values are made consistent after every editing operation insures that the tuples of locally-derived relations are made consistent with the program state after every editing operation. The problem of maintain-

*The concept of local attributes, introduced in [Reps and Teitelbaum 1984], is an extension to standard attribute grammar specifications. A local attribute is associated with a production rather than with a nonterminal.
ing locally-derived relations can thus be viewed more generally as a problem of grouping related attributes scattered throughout the program tree.

One possible way to group attributes is to add extra “link” fields to attributes designated as belonging to a group, keeping all such attributes chained together. This approach has the following general disadvantages:

(1) Extra time and space would be required to maintain links.

(2) Subtree derivation and/or subtree replacement would become more complicated.

(3) New mechanisms would have to be added for navigating linked lists of attributes to allow a user to view a group of linked attributes.

(4) Maintaining ordered groups of attributes would be expensive.

For our particular application, maintaining locally-derived relations, this approach has the following additional disadvantages:

(1) It would not be possible to use an existing database management system to manage locally-derived relations.

(2) Since the tuples of dataflow, unconstrained, and view relations are not attributes, a single representation could not be used for all the relations in the environment.

A second possibility follows from the observation that a node of an attributed tree need not know the physical addresses of its attributes; all it needs are facilities for setting and retrieving their values. Related attributes can thus be maintained externally to the tree, using a mechanism we call external stores.
An external store is defined statically, as part of the editor specification, as a *type* together with four *procedures* to initialize the store, and to insert, delete, and fetch values, respectively. For example, the declaration:

```
external_store DEF_store( STRING,
    DEF_init,
    DEF_insert,
    DEF_delete,
    DEF_fetch )
```

plus the actual procedures DEF_init, DEF_insert, etc. define an external store for attributes of type STRING. An attribute declaration can include a list of external stores, for example:

```
stmt → IDENT "::=" exp
    { def: local STRING attribute stored in DEF_store;
      def = IDENT;
    }
```

declares an attribute of type STRING, local to this production, stored in the external store named DEF_store, whose value is equal to that of the IDENT token.

The insert, delete, and fetch procedures require as an argument a unique key for the attribute instance to be inserted, deleted, or fetched. The insert procedure also requires the current value of the attribute instance as a second argument. Editors generated by the modified Synthesizer Generator observe the following protocol:

1. For every external store *e*, procedure *e.initialize* is called once when the editor is first invoked.
(2) Whenever attribute $a$ of nonterminal $t$ receives a new value $v$, two procedure calls are made for each external store $e$ named in $a$'s declaration. First: $e$.delete(unique key for $t.a$), then: $e$.insert(unique key for $t.a$, $v$). In our current implementation, the pair:

$$<\text{address}(t), \text{attribute-number}(a)>$$

is used as the unique key for $t.a$.

Every external store $e$ must observe the following rule:

The value of $e$.fetch($k$) must be the value $v$ most recently associated with unique key $k$ in a call to $e$.insert.

The major advantage of using external stores to group attributes is that the problems of maintaining the groups are removed from the basic system. This means that:

(1) Basic internal structures, such as nodes and attributes, are unchanged.

(2) The structures used to maintain grouped attributes can be tailored to the individual needs of each group.

(3) Extensive parallel processing may be possible; each external store could potentially be implemented as a separate process.

Using external stores rather than links allows an existing database management system to be used to manage locally-derived relations. In this case, the insert, delete, and fetch procedures associated with each external store would contain calls on the appropriate database-management-system procedures. External stores also allow a single representation to be used for locally-derived, dataflow, view, and unconstrained relations.

The editor designer does not need to know anything about external stores. Locally-derived relations are declared and defined as described in section 8.1.1;
this high-level specification is then automatically translated into a low-level specification including the appropriate external store definitions, as part of the editor-generation process.

8.1.3.2. External computers

Locally-derived relations are, from the internal system's point of view, local attributes, and are thus maintained by the system's attribute-updating mechanisms. Dataflow relations, on the other hand, must be explicitly computed using dataflow analysis, and, if dataflow relations are to be kept consistent with the program state, some analysis must be performed after every editing operation.

The information needed for dataflow analysis is contained in the locally-derived relations CONTROL_FLOW, ROOT, and EDGE_FUNCTION; thus, the problem of providing dataflow analysis in an editing environment can be viewed more generally as the problem of allowing arbitrary computations using relations to be performed following attribute updating after every editing operation.

This capability was added to the Synthesizer Generator by allowing the declaration of external computers. An external computer is simply a procedure that is to be called after a subtree replacement is made, and any attribute updating necessitated by the subtree replacement is done. A single parameter, the root of the current program tree, is passed to the called procedure. External computers are allowed to do anything that does not affect a program tree or its attributes.
In our prototype implementation, the dataflow analysis external computer re-solves all defined dataflow analysis problems for the current program.

8.1.4. The internal representation of relations

In our prototype system we have chosen to represent relations as trees. While this means that we are unable to use an existing database management system, this approach has the advantage that the same facilities used for viewing and navigating programs are available for viewing and navigating relations.

Regardless of what internal representation is chosen for relations, there are several questions that the system designer must consider regarding the way locally-derived and dataflow relations are stored. These are discussed below.

8.1.4.1. Representing locally-derived relations

Looking back at the discussion on external stores, we see that associated with each tuple of a locally-derived relation is information that is not really part of the relation, namely the “unique key” that identifies the local attribute corresponding to the tuple. It is desirable to store locally-derived relations so that:

(1) The unique key associated with a tuple is invisible to the user.
(2) When two unique keys have the same associated tuple*, the user sees only one copy of the tuple in the relation. A request to delete the tuple associated with just one of the unique keys must not cause the tuple to be removed from the relation.

(3) Insertion and deletion of tuples are done efficiently.

One way to satisfy these requirements is to use a database management system with a view facility. The actual tuples of locally-derived relations include the unique-key field, and are hashed or indexed on this field to provide efficient insertion and deletion of tuples. The locally-derived relation seen by the user is a projection of all fields other than the unique-key field. Of course, if the user is to be permitted to display a locally-derived relation while editing the program, an efficient view-updating method is needed.

In our prototype system, the tree representation of a locally-derived relation, which is what is visible to the user, does not include the unique-key field. Instead, we maintain two hash tables for each locally-derived relation as well as maintaining its tree representation. Table 1 is for unique keys, and table 2 is for tuples. All unique keys associated with a single tuple are chained together and to the entry for the tuple in table 2. The entry in table 2 includes a pointer to the appropriate tuple in the tree representation, and a pointer to the first unique key in table 1 on the chain for that tuple.

An insertion of unique key \( k \) with corresponding tuple \( t \) is performed as follows:

*This happens, for example, in the CALLS relation when procedure P1 contains multiple calls to procedure P2.
(1) Unique key $k$ is added to table 1.

(2) Tuple $t$ is looked up in table 2. If the tuple is already there, key $k$ is linked in to its chain. If the tuple is not already there, it is added to table 2 and to the tree representation of the relation, and key $k$ is linked to the entry for $t$ in table 2.

A deletion of unique key $k$ is performed as follows:

Key $k$ is looked up, removed from the chain of the corresponding tuple, and removed from table 1. If $k$ was the only key on the chain of the corresponding tuple, the tuple is removed from both table 2 and the relation tree.

8.1.4.2. Representing dataflow relations

A single relation is maintained for each dataflow problem selected or defined in the editor specification for all programs in the editing environment. If, as in our prototype system, dataflow analysis is redone after a subtree replacement only for the program that was modified, it is desirable to group the tuples for each program tree so that all and only the tuples corresponding to the modified program can be efficiently replaced.

In our prototype system, dataflow relations are represented as right-recursive lists of tuples, with the tuples for each program stored contiguously on the list. For each dataflow problem, a structure is maintained with an entry for each program; a program's entry contains pointers to the first and last tuples in the dataflow relation for that program. When a dataflow problem is re-solved for a modified program, the pointers are used to replace that program's dataflow tuples with the newly created list of dataflow tuples.
8.1.5. Writing and executing queries

A structure editor for building SEQUEL-like queries is predefined and is accessible from any generated editor. A query can be executed by giving a command that includes the name of a buffer to hold the query result. Our prototype system cannot handle queries that use implicit relations.

8.1.6. Program points

In our prototype system, program points are represented as tree-node addresses. Fields of type program point can be used in queries; for example, program points play an important role in the view definition tree of figure 3.3. Views are not supported in our prototype system, however, the UNUSED_DEFINITION relation defined in figure 3.3 could be defined as a query and executed as needed.

Another desirable use for program points is to tie the display of a program to the particular program points in a relation. For example, the TO_DO relation of section 3.2 contains program points and associated comments about what has to be done at that point in the program. Having created the TO_DO relation in some earlier programming session, the programmer would like to be able to indicate a program point in a tuple of the TO_DO relation, and have the editing cursor positioned at the corresponding point in the program.

Ideally, this linkage of the editing cursor to the value of a program point in a relation would take place on a screen with multiple windows. The program would be displayed in one window, and the relation containing program points
in another window. As the user moved from one program point to another in the relation window, the display of the program would change accordingly, with the editing cursor always positioned at the program point currently selected in the relation.

Our prototype implementation does not include windowing; nonetheless, we have implemented a limited linkage capability to provide some feeling for what this facility would be like in a more complete system.

8.2. Practical problems

This section deals with important practical problems that were not addressed in our prototype implementation, and thus were not discussed in the preceding section. These problems will arise in implementations that seek to be more efficient, or to provide more of the facilities described in parts one and two of this thesis, than our prototype.

8.2.1. Allowing attributes to depend on relations

According to the definition of a relationally-attributed grammar given in chapter 3, the semantic equations for an attribute can include operations on named relations, provided this introduces no circular dependencies. This extension to standard attribute grammars is not supported in our prototype implementation. Problems and potential solutions associated with allowing attributes to depend on relations are discussed below.
There are three practical problems that arise when semantic equations can include values extracted from relations:

1. The values in relations can depend on attribute values, and attribute values can depend on values from relations; thus, a circularity test is needed.

2. When a change is made to a relation used in one or more semantic equations, the attributes defined by these equations become potentially inconsistent and must be reevaluated. This requires being able to locate all attribute instances that depend on a given relation.

3. The attribute-updating algorithm of [Reps 1984] is guaranteed to be time optimal only when all initially inconsistent attribute values are associated with tree nodes corresponding to a single production instance. The attributes made inconsistent by a change to a relation can be associated with nodes anywhere in the program tree; thus, an efficient attribute-updating algorithm that can handle an arbitrary set of initially inconsistent attributes is needed.

8.2.1.1. Circularity testing

Relationally-attributed grammar \( G \) can be tested for circularity using the standard attribute grammar circularity test [Knuth 1968] [Knuth 1971] by first transforming \( G \) into a pure attribute grammar \( G' \) such that \( G \) is circular iff \( G' \) is circular. This transformation is done as follows:

1. For every relation \( R \) that is potentially involved in a cyclic dependency* add synthesized attributes and semantic equations to build \( R \) as an attribute of the root of the program tree. Tuple membership assertions are replaced by semantic equations that add the appropriate tuple to the value built up so far.

---

*Relation \( R \) can be involved in a cyclic dependency if it is both used in a semantic equation, and includes tuples defined using attribute values.
(2) Add inherited attributes and semantic equations to broadcast the value of R throughout the program tree. Semantic equations that extract values from R are replaced by semantic equations that extract values from the appropriate inherited attribute.

Testing an attribute grammar for circularity involves building a collection of graphs for each nonterminal of the grammar. The time required for the circularity test depends on the number of graphs per nonterminal; in the worst case, this number is exponential in the size of the grammar, however, empirical evidence [Raiha and Saarinen 1982] indicates that the expected number of graphs per nonterminal is very small. Although we have not looked at enough relationally-attributed grammars to make a strong statement about the likely number of graphs per nonterminal in an attribute grammar derived from a relationally-attributed grammar, we suspect that there will not be a significant increase. Further, as circularity testing is done at editor-specification time rather than at edit time, a fast circularity test is not essential.

8.2.1.2. Locating potentially inconsistent attributes

An editor specification defines, for each relation R, the set A of attributes that may depend on R. During editing, the set of the instances of these attributes currently in the program tree can be maintained and used to locate attribute instances potentially affected by a change to R. An element is added to a relation's attribute-instance set when a new program subtree containing an instance of an attribute in set A is created. Elements are removed from a
relation's attribute-instance set when a program subtree containing instances of attributes in set A is deleted.

8.2.1.3. Attribute updating given multiple change sites

This is the most interesting and most difficult problem that arises when attributes are allowed to depend on values extracted from relations. Of course, this is not the only conceivable way to produce multiple change sites; globally changing the name of a variable, for example, could have the same effect; thus, this problem is of interest to anyone working in the area of attribute-grammar-based editors.

A method is currently being developed by [Reps et al 1985] to handle incremental attribute updating given multiple change sites. This method works for a limited class of grammars similar to absolutely non-circular grammars. The method achieves running time \( O(\log n \ast (k + AFFECTED)) \), where \( n \) is the size of the program tree, \( k \) is the number of change sites, and \( AFFECTED \) is the number of attributes that actually change value due to the initial changes. Although this sounds like a reasonable time bound, high constant factors may make it unreasonable in practice.

A more straightforward technique is nullification followed by reevaluation:

**Nullification**

1. Start at attributes that depend on changed relations; give these attributes a special "null" value.
(2) Follow dependencies backwards, giving every attribute that depends on a nullified attribute the "null" value.

Reevaluation

(1) Recompute values for all nullified attributes; compute the value of an attribute only when all inputs are non-null.

This technique will be reasonable when the relevant attribute dependency chains are very short (few attributes are nullified), or when the effects of changing relations are far-reaching (reevaluation of most of the nullified attributes produces values different from the attribute's previous value).

Section 4.2 includes an example in which the types of variables are extracted from a symbol-table relation and used to compute the types of expressions. If the expression types are used only for local calculations, for example, to insure that the type of the left-hand side of an assignment matches the type of the right-hand side of the assignment, this would be a good candidate for nullification followed by reevaluation after a change to the symbol-table relation. In this example, the number of attributes indirectly dependent on the symbol-table relation is small (the sizes of expression trees are quite small compared to the sizes of programs), and a change to the type of a variable used in an expression might well affect the type of the entire expression.
8.2.2. Problems related to program points

As discussed in section 8.1.6, the programmer can ask to see the point in the program denoted by a program point in a relation. This means that the program-point fields of relations must contain well-defined values.

Program points can appear in any of the kinds of relations in the editing environment: locally-derived, unconstrained, dataflow, or view relations. (See, for example, the DEFINITION relation of section 3.1, the TO_DO relation of section 3.2, the LIVE relation of section 3.3, and the UNUSED_DEFINITION relation of section 3.4.) The presence of program points in locally-derived and dataflow relations is not a problem. These relations are kept consistent with the current state of the program; thus, all program points in a locally-derived or a dataflow relation correspond to actual points in the program. The program points in view relations come from the relations used to define the view. As long as the values of the program-point fields in all relations used in view definitions contain well-defined values, the presence of program points in view relations is not a problem.

The problems related to program points arise because of the possible presence of program points in unconstrained relations. What should happen when the programmer tries to delete part of a program containing points that are in unconstrained relations? There are a number of possibilities:

(1) Such deletions are disallowed; the programmer is told which unconstrained relations prevent the deletion.
(2) The deletion is performed; all tuples of unconstrained relations with program points in the deleted subtree are deleted from the relations.

(3) The deletion is performed; all corresponding program points in unconstrained relations are replaced by a special ‘NULL’ value.

(4) The deletion is performed; all corresponding program points in unconstrained relations are replaced by the program point of the parent of the root of the deleted subtree.

In all the cases listed above, it must be possible to determine whether a given program subtree includes program points that appear in unconstrained relations. In addition, it must be possible to determine which unconstrained relations, or which tuples of unconstrained relations contain these program points.

Some systems may require that deleted subtrees be traversed for reasons independent of the problems associated with program points. In our prototype system, for example, deleted subtrees are traversed to reclaim storage*. In such systems, or in systems that would not otherwise traverse deleted subtrees but which consider this cost to be acceptable, the two requirements mentioned above can be met in a straightforward way: at each point in the program tree, a list is kept of the unconstrained relations, or of the actual tuples, in which that program point occurs. Whenever a tuple is added to or deleted from an unconstrained relation with a field of type ‘program point’, information is added to or deleted from the list at the program point that appears in the tuple. Whenever

*This includes the storage used by the attributes of the deleted subtree; this is the means by which the tuples of locally-derived relations associated with deleted subtrees are removed from the relations.
the programmer tries to delete a program subtree, the subtree is traversed, and, for every point with a non-empty list, the appropriate action is taken according to the decision of the system designer: the values on the list are used to warn the programmer, and/or to modify the appropriate unconstrained relations.

Some systems may defer reclaiming the storage used by a deleted subtree, and thus may not traverse the subtree when it is first deleted. The designers of such systems may wish to find an alternative to the strategy described above for handling the problems associated with program points. The ideal approach would be one that could produce, in time proportional to the size of the answer set rather than in time proportional to the size of the deleted subtree, the set of nodes of the subtree whose program points were in unconstrained relations. The methods of [Tarjan 1983] are relevant to this problem, but have the disadvantage of extra storage overhead. Solving the problem of deleting program subtrees in the presence of program points in unconstrained relations without traversing the deleted subtree is left as a topic for future research.

8.2.3. Dataflow analysis

In our prototype implementation, dataflow analysis is done using Kildall's iterative method, and every dataflow problem selected or defined in the editor specification is re-solved from scratch after every program modification. This approach, which is conceptually straightforward and easy to program, is reasonable for a prototype, but is not practical for a "real" system.
In this section, we discuss a number of the aspects of our implementation that could be changed to provide dataflow analysis without causing intolerable delays between editing operations.

8.2.3.1. The dataflow-analysis algorithm

Although Kildall's iterative method is easy to understand and to program, it is not necessarily the best dataflow-analysis algorithm to use in an editing environment. A survey of dataflow analysis algorithms is given in [Muchnick and Jones 1981], and is beyond the scope of this thesis. We will simply point out that just because one algorithm outperforms another in theory, does not mean that it will do so in practice. Further, different dataflow analysis algorithms may be best for editing environments built for different programming languages.

8.2.3.2. Recomputation from scratch versus incremental analysis

The use of efficient incremental algorithms has obvious advantages for interactive systems such as editing environments. Possible approaches to incremental dataflow analysis are given in [Ryder 1983] and [Zadeck 1984]. These methods are limited in their applicability; that of [Ryder 1983] applies only to a change in the function associated with an edge of the control-flow graph, and not to a change in the structure of the control-flow graph. The method of [Zadeck 1984] deals with structural as well as functional changes in the control-flow graph, but cannot be applied to all dataflow problems definable using Kildall's lattice model.
In spite of these limitations, using these incremental algorithms where appropriate will probably be significantly better than always redoing dataflow analysis from scratch.

8.2.3.3. When to perform dataflow analysis

In our prototype system, dataflow analysis is performed after every program modification; no further action by the programmer is possible until dataflow analysis is finished. There are several alternatives to this approach. One is to perform dataflow analysis only when its results are needed: when the programmer asks to see a dataflow relation, when a query that uses a dataflow relation is executed, or when a view that uses a dataflow relation is displayed. This approach is reasonable when there are significantly fewer demands for the results of dataflow analysis than there are program modifications.

Another possibility is to make use of "wasted cycles": any time the system would normally be waiting for user input, use that time to do dataflow analysis, being prepared to interrupt the dataflow analysis as soon as the user input arrives.

This approach is similar to the previous one in that the user never has to wait for dataflow analysis when he has not required its results. By using wasted cycles to do dataflow analysis whenever possible, it is more likely that the results of dataflow analysis will be available when demanded, thus cutting down on user wait-time still further.
A third possibility is to have dataflow analysis done in a separate process, running concurrently with the main editing process. Again, the programmer will have to wait for dataflow analysis only when its results are required, and only if the dataflow analysis process has not finished at that time. The practicality of this approach will depend on the capabilities of the system underlying the editing environment.

8.2.4. Scoped Languages

This section addresses the problems associated with naming variables in scoped languages. Unlike the other practical problems considered above, this problem arises at the editor-designer level, rather than at the system-designer level.

We first point out that in a non-sScoped language, or in a scoped language in which variable declarations cannot be inherited from enclosing scopes, naming variables is not a problem. In a non-scoped language, identifiers alone are sufficient to name variables. In a scoped language in which variable declarations cannot be inherited from enclosing scopes, the pair \texttt{<IDENTIFIER, current-scope-name>} can be used as a variable name. It is convenient to use the program point of the beginning of a scope as the scope name; this value, which is unaffected by editing operations, can be broadcast throughout the body of the scope using attributes.

In a scoped language in which variable declarations \textit{can} be inherited from enclosing scopes, naming variables becomes a problem. In a pure attribute-
grammar-based system, this problem is can be solved using symbol-table attributes. The symbol table for scope S includes information passed down from the scope enclosing S, combined with information from the declarations of S. The entry for each identifier i includes some piece of information specifying which declaration of i applies to all uses of i in this scope. This could be, for example, the name of the least enclosing scope in which i is declared, or the program point of the declaration.

Symbol-table attributes can be used in a relationally-attributed-grammar-based system as well. Assuming that the entry for identifier i in a symbol-table attribute includes the program point of the appropriate declaration of i, symbol-table attributes can be used to define the locally-derived relation:

\[ \text{VAR\_USE} (\text{decl-program-point}, \text{IDENT}, \text{use-program-point}) \]

Figure 8.8 below illustrates the use of symbol-table attributes in defining the VAR\_USE relation by showing an example grammar production and the corresponding tuple membership assertion.

\[
\text{stmt} \rightarrow \text{IDENT} \quad \text{":="} \quad \text{exp} \\
\{ \text{if (member( IDENT, stmt.symbol-table ))} \\
\quad \text{then} <\text{lookup-decl-prog-pt( IDENT, stmt.symbol-table)}, \\
\quad \text{IDENT,} \\
\quad \text{program-point( stmt )} \rangle \in \text{VAR\_USE}; \\
\text{else} <\bot, \\
\quad \text{IDENT,} \\
\quad \text{program-point( stmt )} \rangle \in \text{VAR\_USE}; \} 
\]

Figure 8.8
Example tuple membership assertion for the VAR\_USE relation
The locally-derived VAR_USE relation can be used in queries and view definitions that involve variable names, for example:

(1) Query: Find all variable uses that correspond to the declaration of 'x' at program point 'p'.

(2) View: Compute the set of variable uses for which there are no corresponding declarations.

Figure 8.9
Example query and view involving variable names

Figures 8.10a and 8.10b below show the query and view-definition trees for these examples. The query tree for example (1) simply selects the tuples with the given values in the 'IDENT' and 'decl-program-point' fields, and then projects the 'IDENT' and 'use-program-point' fields. The view-definition tree for example (2) takes advantage of the fact that the 'decl-program-point' field of the tuple for an undeclared variable has the special value '⊥'; all tuples with this value in the 'decl-program-point' field are selected, and again, the 'IDENT' and 'use-program-point' fields are projected.

Although the use of symbol-table attributes solves a relational system's variable-naming problems, there is a major disadvantage to this solution: a change to any variable declaration causes a change to all symbol-table attributes in the scope of the declaration, which in turn forces the recomputation of all variable names in the scope of the changed declaration. For example, when any declaration is changed, the symbol table lookups in the tuple membership asser-
tion of figure 8.8 are redone at every instance of an assignment statement in the scope of the declaration, not just at the statements that assign to the variable whose declaration has changed.
A possible alternative to maintaining VAR_USE as a locally-derived relation using symbol-table attributes, is to maintain VAR_USE as a view relation. The VAR_USE view is defined using the locally-derived relations:

\[\text{LOCAL\_VAR\_USE( IDENT, use\text{-}program\text{-}point )}\]

\[\text{VAR\_DECL( decl\text{-}scope, IDENT, decl\text{-}program\text{-}point )}\]

and the implicit relation:

\[\text{ANCESTOR( ancestor, descendent ).}\]

The definitions of LOCAL\_VAR\_USE and VAR\_DECL do not depend on the existence of symbol-table attributes. Attributes must be used to broadcast the program point of the beginning of each scope throughout its declarations to provide values for the ‘decl\text{-}scope’ field of the VAR\_DECL relation; however, this use of attributes has very little overhead as these values are unaffected by editing operations. Of course, as discussed in section 7.7, incrementally updating a view defined using the ANCESTOR relation may be a problem.

Figure 8.11 gives example grammar productions and the corresponding tuple membership assertions for the LOCAL\_VAR\_USE and VAR\_DECL relations:

\[\text{stmt} \rightarrow \text{IDENT } \text{";=" } \text{exp} \]
\[\{ <\text{IDENT, program\text{-}point( stmt )} > \in \text{LOCAL\_VAR\_USE}; \}\]

**Figure 8.11a**

Example tuple membership assertion for the LOCAL\_VAR\_USE relation
Figure 8.11b
Example tuple membership assertion for the VAR_DECL relation

Figure 8.12 shows the view-definition tree that defines the VAR_USE relation using LOCAL_VAR_USE, VAR_DECL, and ANCESTOR. Conceptually, the view is computed as follows:

(1) Create the intermediate relation:
(IDENT, use-program-point, decl-scope, decl-program-point), where 'decl-scope' is any scope in which IDENT is declared.

(2) Limit (1) to include only tuples in which 'decl-scope' is an ancestor of 'use-program-point'.

(3) Relation (2) may associate a number of ancestor 'decl-scopes' with each variable use; limit this relation to include only tuples in which 'decl-scope' is the least such ancestor:

(3)(a) Isolate and save the uses with which only a single decl-scope is associated.

(3)(b) For all other uses, compute the least ancestor decl-scope.

(4) Relation (2) includes entries only for declared variables; use this information to create a tuple for each undeclared variable with the special value '⊥' in its 'decl-program-point' field.

(5) Combine the results of (3) and (4).

Note that the values of the 'decl-scope' fields rather than the values of the 'decl-program-point' fields are used to determine whether a declaration is an
ancestor of a use; although the declaration's scope may be the ancestor of the use, the declaration itself is not the ancestor in the program tree of any use.

The view definition tree shown below in figure 8.12 is built up in stages corresponding to steps (1), (2), (3), (4), and (5) above.

(1)

\[
\begin{align*}
\text{PROJECT} \\
\text{IDENT, use-program-point,} \\
\text{decl-scope, decl-program-point}
\end{align*}
\]

\[
\begin{align*}
\text{JOIN} \\
\text{LOCAL_VAR_USE.IDENT = VARDECL.IDENT}
\end{align*}
\]

\[
\begin{align*}
\text{LOCAL_VAR_USE} & \quad \text{VARDECL}
\end{align*}
\]
(2)  
PROJECT  
IDENT, use-program-point,  
decl-scope, decl-program-point  

JOIN  
use-program-point = descendant AND  
decl-scope = ancestor  

(1)  
ANCESTOR  

(3)(a)  
MINUS  

PROJECT  
decl-program-point,  
IDENT, use-program-point  

PROJECT  
decl-program-point$1$,  
IDENT, use-program-point  

JOIN  
use-program-point = use-program-point AND  
decl-scope ≠ decl-scope  

(2)  

(2)  

(2)
(3)(b)

MINUS

PROJECT
decl-program-point, IDENT, use-program-point

PROJECT
decl-program-point$1, IDENT, use-program-point

JOIN
decl-scope$1 = ancestor AND decl-scope$2 = descendent

JOIN
use-program-point = use-program-point AND decl-scope ≠ decl-scope

ANCESTOR

(2)

(2)

(2)
Figure 8.12
Definition tree for the VAR_USE view relation
Maintaining the VAR_USE relation as a view is equivalent to maintaining it as a locally-derived relation in terms of its usefulness; the query and view-definition trees of figure 8.10 can be used in either case to define the example query and view of figure 8.9. Some of the disadvantages of using symbol-table attributes to define VAR_USE were discussed above; maintaining VAR_USE as a view also has disadvantages:

1. Redundancy: the information in the LOCAL_VAR_USE relation is duplicated in the VAR_USE relation.

2. Costly update: keeping the VAR_USE view relation up-to-date may be very expensive, even given a solution to the problem of updating views that use implicit relations.

To elaborate on the second point, consider what must be done to update the VAR_USE view when a new declaration of identifier \(i\) is added in scope \(S\) of the program. The new declaration could replace any of the previous declarations of \(i\) associated with a use of \(i\) in a scope subordinate to \(S\). The "least ancestor" computation is done in Step (3)(b); in the worst case, the size of the change to the intermediate relation represented by the "JOIN (use-program-point = use-program-point AND decl-scope \(\neq\) decl-scope)" node is:

\[
O \left( ([\# \text{ of declarations of } i]) \times (\# \text{ of uses of } i) \right)^2.
\]

The disadvantages of maintaining VAR_USE as a view can be partially overcome by redefining VAR_USE to contain less specific information, computing this information as needed in the queries and views that use VAR_USE. For example, one could define the view relation:
VAR_USE' (decl-program-point, IDENT, use-scope)

using the locally-derived relations:

LOCAL_VAR_USE' (use-scope, IDENT, use-program-point)

VAR_DECL (decl-scope, IDENT, decl-program-point)

and the implicit relation:

ANCESTOR (ancestor, descendent).

The definition for the VAR_USE' view relation, given below in figure 8.13, is very similar to the definition for the VAR_USE relation given above in figure 8.12. The important difference between VAR_USE and VAR_USE' is that while VAR_USE includes, for each scope, a tuple for every use of an identifier in the scope, VAR_USE' includes, for each scope, a tuple for every identifier used in the scope.
(1)

\[
\text{PROJECT}
\]

use-scope, IDENT,
decl-scope, decl-program-point

\[
\text{JOIN}
\]

LOCAL_VAR_USE'.IDENT = VARDECL.IDENT

\[
\text{PROJECT}
\]

use-scope, IDENT

LOCAL_VAR_USE'

(2)

\[
\text{PROJECT}
\]

use-scope, IDENT,
decl-scope, decl-program-point

\[
\text{JOIN}
\]

use-scope = descendant AND decl-scope = ancestor

\[
\text{ANCESTOR}
\]

(1)
(3)(a)

```
MINUS

PROJECT
decl-program-point$1,
IDENT, use-scope

PROJECT
decl-program-point,
IDENT, use-scope

JOIN
IDENT = IDENT AND
use-scope = use-scope AND
decl-scope ≠ decl-scope

(2)

(2)
```
(3)(b)

MINUS

PROJECT
decl-program-point,
IDENT, use-scope

PROJECT
decl-program-point$1,
IDENT, use-scope

JOIN
dcl-scope$1 = ancestor AND
dcl-scope$2 = descendant

JOIN
IDENT = IDENT AND
use-scope = use-scope AND
dcl-scope ≠ decl-scope

(2)

(2)

(2)

ANCESTOR
Figure 8.13
Definition tree for the VAR_USE' view relation
VAR_USE' is smaller than VAR_USE, less redundant, and less costly to update. In both cases, the most expensive view updates are in response to adding or deleting a declaration. The dominating cost of the update is the computation of the change set for the JOIN node of Step (3)(a); for the VAR_USE view this cost is, in the worst case:

\[ O \left( \left[ \# \text{ of declarations of } i \right] \times \left[ \# \text{ of uses of } i \right] \right)^2; \]

for the VAR_USE' view this cost is, in the worst case:

\[ O \left( \left[ \# \text{ of declarations of } i \right] \times \left[ \# \text{ of scopes in which } i \text{ is used} \right] \right)^2. \]

The cost of keeping the VAR_USE' view relation up-to-date seems quite reasonable; how it compares to the cost of keeping the locally-derived VAR_USE relation up-to-date using symbol-table attributes depends on the programming style of the user.

There is, of course, one further consideration: VAR_USE' contains less information than VAR_USE; thus, we cannot simply substitute VAR_USE' for VAR_USE in the query and view-definition trees of figure 8.10 to define the example query and view of figure 8.9. New versions of the query and view-definition trees appear below in figure 8.14.
Figure 8.14a
Query tree for example (1) of figure 8.9
using the VAR_USE' relation
8.2.4.1. Conclusions

Naming variables in scoped languages is essentially a problem of associating uses of identifiers with the appropriate declarations. This section has presented three possible solutions to this problem:

(1) Use symbol-table attributes to define a locally-derived relation in which each use of an identifier is associated with the appropriate declaration.
(2) Same as (1), but using a view relation in place of symbol-table attributes and a locally-derived relation.

(3) Define a view relation in which, for each identifier, for each scope in which the identifier is used, the use-scope is associated with the appropriate declaration.

Which of these three solutions will prove best in practice, or whether there is another solution, superior to all of these, are left as topics for future research.
References

[Aho and Ullman 1979]
Aho, A.V. and J.D. Ullman
Universality of data retrieval languages.
*Conf. record of the 6th annual ACM Symp. on POPL*
San Antonio, TX, pp. 110-120, Jan. 1979

[Astrahan and Chamberlin 1975]
Astrahan, M.M. and D.D. Chamberlin
Implementation of a structured English query language.

[Bernstein and Chiu 1981]
Bernstein, P.A. and D.W. Chiu
Using semi-joins to solve relational queries.

[Beshers and Campbell 1985]
Beshers, G. and R. Campbell
Maintained and constructor attributes.
*Proc. of the SIGPLAN 85 Symp. on lang. issues in prog. environments*
Seattle, WA, pp. 93-106, June 1985

[Blasgen and Eswaran 1977]
Blasgen, M.W. and K.P. Eswaran
Storage and access in relational databases.

[Chandra 1981]
Chandra, A.K.
Programming primitives for database languages.
*Conf. record of the 8th annual ACM Symp. on POPL*

[Chandra and Newey 1985]
Chandra, A.K. and M. Newey
Private communication.
[Christodoulakis 1983]
Christodoulakis, S.
Estimating block transfers and join sizes.
Proc. ACM SIGMOD Conference
San Jose, CA, pp. 40-54, 1983

[Closkin and Mellish 1981]
Closkin, W.F. and C.S. Mellish
Programming in Prolog
Springer-Verlag
Berlin, Heidelberg, New York, Tokyo, 1981

[Codd 1970]
Codd, E.F.
A relational model of data for large shared data banks.

[Coppersmith and Winograd 1982]
Coppersmith, D. and S. Winograd.
On the asymptotic complexity of matrix multiplication.

[Demers et al 1985]
Demers, A., A. Rogers, and F.K. Zadeck
Attribute propagation by message passing.
Proc. of the SIGPLAN 85 Symp. on lang. issues in prog. environments
Seattle, WA, pp. 43-59, June 1985

[Demolombe 1980]
Demolombe, R.
Estimation of the number of tuples satisfying a query expressed in
predicate calculus language.
Proc. 6th Int. Conf. on Very Large Data Bases
Montreal, Canada, pp. 55-63, 1980
[Donzeau-Gouge et al 1980]
Donzeau-Gouge, V., G. Huet, G. Kahn, and B. Lang
Programming environments based on structured editors:
The MENTOR experience.
INRIA Rapports de Recherche No. 26, July 1980

[Eswaran et al 1976]
Eswaran, K.P., J.N. Gray, R.A. Lorie, and I.L. Traiger
The notions of consistency and predicate locks in a database system.
Communications of the ACM Vol. 19 No. 11, pp. 624-633, 1976

[Freundenberger 1984]
Freundenberger, S.
On the use of global optimization algorithms for the detection of semantic
programming errors.

[Gillet 1977]
Gillett, W.D.
Iterative global flow techniques for detecting program anomalies.
Ph.D. Thesis. University of Illinois at Urbana-Champaign, Jan. 1977

[Gotlieb 1975]
Gotlieb, L. R.
Computing joins of relations.
ACM SIGMOD Int. Symp. on Management of Data. pp. 55-63, 1975

[Hall 1976]
Hall, P. A. V.
Optimization of single expressions in a relational data base system.

[Horwitz and Teitelbaum 1985]
Horwitz, S. and T. Teitelbaum
Relations and attributes: A symbiotic basis for editing environments.
Proc. of the SIGPLAN 85 Symp. on lang. issues in prog. environments
Seattle, WA, pp. 93-106, June 1985
[Johnson 1983]
Johnson, G.
An approach to incremental semantics.
*Ph.D. Thesis*, University of Wisconsin - Madison, 1983

[Kildall 1973]
Kildall, G.
A unified approach to global program optimization.
*Conf. record of the 1st ACM Symp. on POPL*.

[Knuth 1968]
Knuth, D.E.
Semantics of context-free languages.

[Knuth 1971]
Knuth, D.E.
Semantics of context-free languages: correction.

[Koenig and Paige 1981]
Koenig, S. and R. Paige
A transformational framework for the automatic control of derived data.
*Proc. 7th Int. Conf. on Very Large Data Bases*.

[Linton 1984]
Linton, M.A.
Implementing relational views of programs.
*Proc. of the ACM SIGSOFT/SIGPLAN software eng. symp. on practical software development environments*.
Pittsburgh, PA, pp. 132-140, April 1984

[Liu 1979]
Liu, L.
Essential uses of expressions in set-oriented expressions.
*Ph.D. Thesis*, Cornell University, May 1979
[Masinter 1980]
Masinter, L.M.
Global program analysis in an interactive environment.

[Medina-Mora and Notkin 1981]
Medina-Mora, R. and D.S. Notkin
ALOE users' and implementors' guide.

[Melhorn 1984]
Melhorn, K.
Data structures and algorithms 3:
Multi-dimensional searching and computational geometry
Springer-Verlag
Berlin, Heidelberg, New York, Tokyo, 1984

[Muchnick and Jones 1981]
Muchnick, S.S. and N.D. Jones (editors)
Program flow analysis: Theory and applications
Prentice-Hall Inc.

[Osterweil and Fosdick 1976]
Osterweil, L.J. and L.D. Fosdick
DAVE - A validation error detection and documentation system for
Fortran programs.
Software - Practice and experience Vol. 6, pp. 473-486, 1976

[Paige and Koenig 1982]
Paige, R. and S. Koenig
Finite differencing of computable expressions.
ACM Trans. on Programming Languages and Systems
Vol. 4, No. 3, pp. 402-454, July 1982

[Piatetsky-Shapiro 1984]
Piatetsky-Shapiro, G.
A self-organizing database system - a different approach to query optimization.
Ph.D. Thesis, New York University, April 1984
[Raiha and Saarinen 1982]
Raiha, K. and J. Saarinen
Testing attribute grammars for circularity.
*Acta Informatica* Vol. 17, Fasc. 2, pp. 185-192, June, 1982

[Reiss 1984]
Reiss, S.
Graphical program development with PECAN program development system.
*Proc. of the ACM SIGSOFT/SIGPLAN software eng. symp. on practical software development environments*
Pittsburgh, PA, pp. 42-48, April 1984

[Reps 1984]
Reps, T.
*Generating language-based environments.*

[Reps and Teitelbaum 1984]
Reps, T. and T. Teitelbaum
The Synthesizer Generator.
*Proc. of the ACM SIGSOFT/SIGPLAN software eng. symp. on practical software development environments*
Pittsburgh, PA, pp. 42-48, April 1984

[Reps et al 1985]
Reps, T., C. Marceau, and T. Teitelbaum
Remote attribute updating for language-based editors.
In progress.

[Richard 1981]
Richard, P.
Evaluation of the size of a query expressed in relational algebra.
*Proc. ACM SIGMOD Conference*
Ann Arbor, MI, pp. 155-163, 1981

[Rosen 1977]
Rosen, B.
High-level data flow analysis
[Ryder 1983]
Ryder, B.
Incremental data flow analysis.
_Conf. record of the 10th annual ACM Symp. on POPL_
Austin, TX, pp.167-176, Jan. 1983

[Selinger et al 1979]
Access path selection in a relational database management system.
_Proc. ACM SIGMOD Conference_
Boston, MA, pp. 23-34, June 1979

[Severance and Lohman 1976]
Severance, D.G. and G.M. Lohman
Differential files: Their application to the maintenance of large databases.

[Shmueli and Itai 1984]
Shmueli, O. and A. Itai
Maintenance of views.
_Proc. ACM SIGMOD Conference_
Boston, MA, pp. 240-255, 1984

[Smith and Chang 1975]
Smith, J.M. and P. Chang
Optimizing the performance of a relational algebra database interface.

[Snodgrass 1984]
Snodgrass, R.
Monitoring in a software development environment: A relational approach.
_Proc. of the ACM SIGSOFT/SIGPLAN software eng. symp. on practical software development environments_
Pittsburgh, PA, pp. 124-131, April 1984
[Stonebraker and Keller 1980]
Stonebraker, M. and K. Keller
Embedding expert knowledge and hypothetical data bases into a data base system.
*Proc. ACM SIGMOD Conference*
Santa Monica, CA, pp. 58-66, May 1980

[Stonebraker et al 1976]
Stonebraker, M., E. Wong, and P. Kreps
The design and implementation of INGRES.
*ACM Trans. on Database Systems* Vol. 1 No 3, pp. 189-222, Sept. 1976

[Tarjan 1983]
Tarjan, R.E.
*Data structures and network algorithms.*
Society for industrial and applied mathematics
Philadelphia, PA. 1983

[Wong and Youssefi 1976]
Wong, E. and K. Youssefi
Decomposition - a strategy for query processing.

[Yao 1979]
Yao, S.B.
Optimization of query evaluation algorithms.
*ACM Transactions on Database Systems* Vol. 4 No. 2, pp. 133-155, 1979

[Zadeck 1984]
Zadeck, F.K.
Incremental data flow analysis in a structured program editor.
*Proc. of the SIGPLAN 84 Symp. on Compiler Construction*
Montreal, Canada, pp.132-143, June 1984