THE DESIGN AND IMPLEMENTATION OF QUERY MODIFICATION IN THE DATA BASE MANAGEMENT SYSTEM INGRES

by

Eric Allman

Memorandum No. UCB/ERL M81/11

10 March 1981

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Research sponsored by the U.S. Army Research Office Grant DAAG29-79-C-0182 and the Department of Energy Grant DE-AS01-78ET29135.

The Design and Implementation of Query Modification in the Data Base Management System INGRES

Eric Allman

ABSTRACT. Certain database services such as views, integrity assertions, and protection constraints can be provided using a technique known as Query Modification. This technique involves modifying a high-level query before processing into a query that can be run safely without further checks. The design and implementation of Query Modification in the database management system INGRES is described.

INGRES (INteractive Graphics and REtrieval System) is a relational database management system currently operational at the University of California at Berkeley. Recently, virtual views, semantic integrity assertions, and protection constraints were implementing using algorithms based on query modification. Query modification is a technique whereby user queries are modified into queries that are guaranteed to have no semantic or access violations. These queries can be executed directly on the database, with no runtime checking required.

This paper describes the design and implementation of these three modules in the database management system INGRES. Section one is a problem statement, describing desirable features of these modules. Section two gives a brief summary of the INGRES environment. Section three discusses the query modification technique. Section four describes the design and implementation of query modification, including an evaluation of how well the implementation satisfies the design goals, and in general to what extent the implementation has

succeeded. Finally, section five gives a summary and conclusion.

1. Problem Statement

1.1. Views

Views allow definition of "virtual relations", *i.e.*, relations that are defined as a function of other relations instead of being physically instantiated. Views are useful as a macro facility and to support obsolete schemas[1]. For example, if a schema had existed that kept the employee and department information in a single relation, it would be possible to maintain this view of the world even if the relations were split up.

In IMS- or DBTG-like languages, the sub-schemas (view definitions) are maintained by the database administrator – as are all aspects of the schema. In INGRES, the schema is dynamic and may be extended as convenient by the user. Logically, the same ability should be available for views. Thus, view definitions should not require another language that the user must learn; in other words, view definitions should be an extension of the existing query language.

1.2. Integrity

Frequently information is known about the possible values that a domain may legally take on that cannot be encoded in the domain's type. These can be expressed as semantic assertions on the data. For example, an assertion might be:

```
assert
  (salary > 0 or status = VOLUNTEER)
  and 1980 - birthdate ≥ 16
```

meaning that all records should specify someone with positive salary or else a volunteer, and that they must be at least 16 years old.

---

[1] *i.e.*, data descriptions.

Integrity assertions should be expressed as a qualification in the query language.

1.3. Protection

Protection is needed to restrict unauthorized access or update of the database. Ideally, the protection system will allow selective access to subsets of a relations – either a subset of the attributes or a subset of the tuples, or both.

Access to attributes can be constrained easily by examining the query in advance. However, access to specified tuples is difficult to control, since in the general case violations can not be determined until run time. For example, if each user is only authorized to see the tuples of the employee that s/he manages, then the query:

```
range of e is emp
retrieve (e.all) where e.dept = toy
```

may or may not have a protection violation, depending on whether the current user manages the toy department; in general this can not be determined until the query is run.

2. BACKGROUND

INGRES [HELD75] (INteractive Graphics and REtrieval System) is a relational database management system running on PDP-11 and VAX-11[2] series computers under the UNIX[3] operating system [RITC74]. The query language QUEL is based on the relational calculus [CODD71]. For a complete introduction to INGRES and QUEL, see [WOOD79] and [EPST77]. A subset of QUEL, necessary to understand query modification, is presented here. For a complete discussion, see [STON76]

2.1. QUEL

QUEL has one declarative statement and four commands. The declarative statement is of the form

RANGE OF variable IS relation

The *variable* is declared to "range over" the *relation*, that is, during query processing the *variable* takes on the value of successive tuples from *relation*[4].

QUEL commands are **retrieve**, **append**, **delete**, and **replace**, having the obvious meanings. The general syntax is:

command target (target-list) **where** qual

where *target* is where to place the results[5]. The optional **where** *qual* restricts the domain of the operation; *qual* is a predicate calculus statement.

The *target-list* is a sequence of expressions of the form

attribute = function

---

[2]PDP and VAX are trademarks of Digital Equipment Corporation.

[3]UNIX is a trademark of Bell Laboratories.

[4]This is conceptually only; the implementation will do everything possible to avoid looking at all tuples in *relation*.

[5]In a retrieve, *target* may be omitted, meaning "retrieve the data to the terminal"; in updates *target* is the relation name (for **append**) or variable (for **replace** and **delete**) to be updated.

where *attribute* is the name of an attribute in the *target* relation, and *function* is an arbitrary function.

Functions can be attributes (*i.e.*, values in a relation), arithmetic formulae, aggregates, or any reasonable combination. Attributes are of the form *var.dom*, where *var* is a variable as declared in a range statement and *dom* is a domain name. QUEL also provides "aggregates" and "aggregate functions". Aggregates are of the form

aggr(function **where** qual)

where *aggr* includes **avg**, **sum**, **min**, **max**, and **count**; aggregates return a single value. Aggregate functions are of the form:

aggr(function$_1$ **by** function$_2$ **where** qual)

Aggregate functions return a single aggregate value for each unique value of *function$_2$*

2.2. Internal Format

Queries are represented internally as trees. The following discussion will use the tree shown in figure 1. This tree uses the relation

emp(enum, dept, sal, posn, step)

and represents the query

```
range of e is emp
replace e (
  sal = 1.1 * avg(e.sal by e.dept),
  step = e.step + 1)
where e.posn = "programmer"
and e.sal < avg(e.sal where e.posn = "programmer")
```

This query sets the salary of all programmers who earn less than the average salary for programmers to 10% higher than the average salary in their

department, and promotes them one step[6].

A brief discussion of the nodes should suffice to explain the semantics of the tree.

ROOT — The ROOT node is a placeholder representing the entire query. On the left is the target list and on the right is

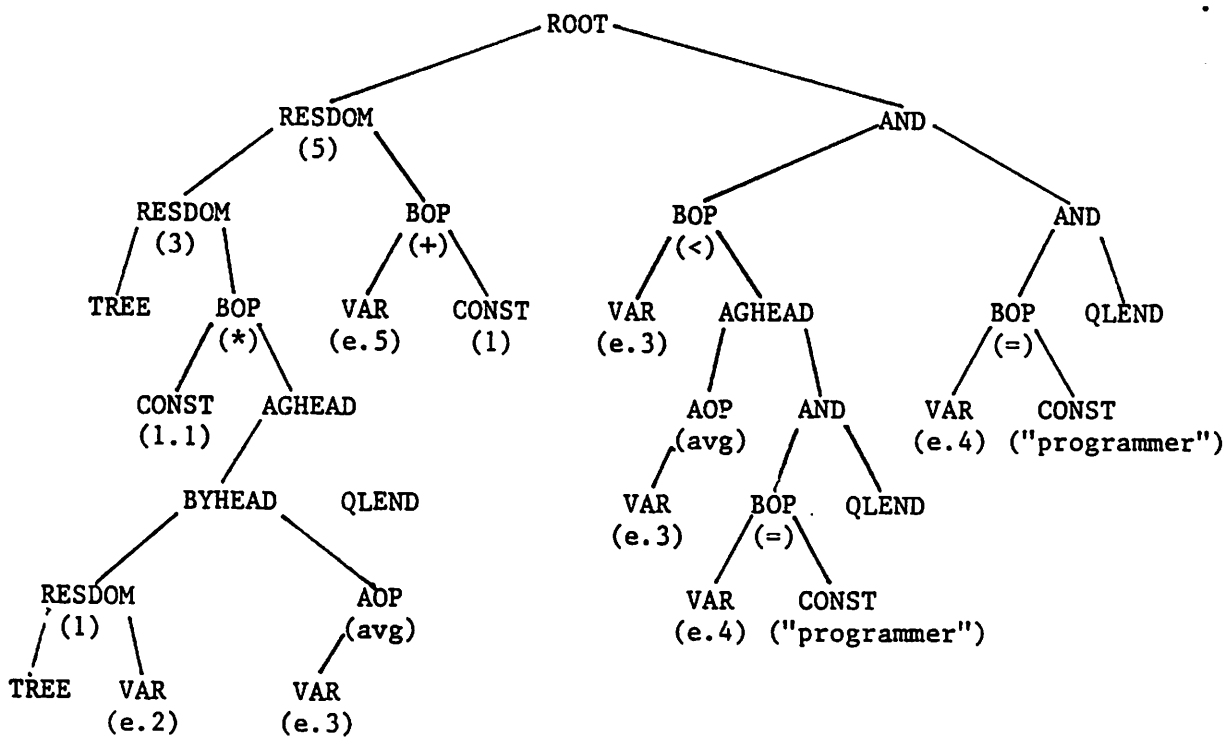


Figure 1: Query tree for the query:

```
range of e is emp
replace e (
  sal = 1.1 * avg(e.sal by e.dept),
  step = e.step + 1)
where e.posn = "programmer"
and e.sal < avg(e.sal where e.posn = "programmer")
```

[6]Hardly an equitable salary schedule.

the qualification.

RESDOM — This represents a result domain. The argument is the domain number in the result relation. On the right is the function to replace this domain; on the left is the rest of the target list.

TREE — This node terminates a target list.

BOP — A binary operator. There is also a UOP for unary operator that has one argument on the right.

CONST — A constant. The argument is the constant value. In the actual implementation there are several node types representing constants of different data types.

VAR — An attribute. The argument is the variable and domain number.

AND — The logical *and* operator. There is also an OR node. Qualifications are kept in conjunctive normal form, with the disjunctive clauses on the left.

QLEND — Terminates a qualification list.

AGHEAD — Introduces an aggregate or aggregate function. The qualification is on the right. In an aggregate there is an AOP on the left; in an aggregate function there is a BYHEAD on the left.

AOP — An aggregate operator. The argument defines what aggregate to use. The function to be aggregated is on the right. The left is unused.

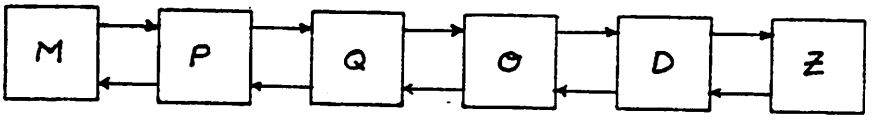
BYHEAD — Introduces the BY list for an aggregate function. The by list looks like a target list and is on the left. The right

must be an AOP.

Range variables in the function part or the qualification part are local to that aggregate. Range variables inside a BY list bind out to the rest of the query.

2.3. Process Structure

Since processes on the PDP-11 are limited to 65K bytes, INGRES must run in several processes. These processes are represented in figure 2. Processes communicate via UNIX pipes.



M = tty monitor
P = parser
Q = query modification
O = one variable query processor
D = decomposition
Z = database utilities

Figure 2: INGRES Process Structure

3. QUERY MODIFICATION

The view, integrity, and protection subsystems can all be implemented using a query modification technique. This technique can be summarized as taking a user query and modifying it into a query that can be run safely on the physical database; the essential feature is that no extra code is needed in the query processor to implement these subsystems.

3.1. Design Goals

3.1.1. Maximize code overlap

Whenever possible, code should be shared among these three subsystems. This can be thought of as, exploit natural similarities whenever possible. The goal was to have at least fifty percent of query modification code non-specific to any one subsystem.

3.1.2. Minimize system impact

Whenever possible, it was desired that the rest of INGRES be unaware that query modification exist. Naturally, the parser must be expanded to include the syntax necessary to define views and constraints; however, the changes necessary to query optimization (decomposition), execution (one variable query processor), and the access methods should be kept to a minimum.

3.1.3. Use standard interfaces

This follows as a corollary to the above goal. The internal form of queries that query modification reads should be indistinguishable from the form that it generates. Besides minimizing the effort necessary to implement the interfaces, this also makes the system more understandable and permits progress on the rest of INGRES to continue concurrently with

development of query modification.

3.2. Syntax

In the following descriptions, braces ("[ ]") represent optional syntax.

3.2.1. Views

The syntax to define a view is:

define view name (target-list) [**where** qual]

The semantics are nearly identical to the **retrieve into** statement, except that instead of instantiating the result, the definition is stored.

3.2.2. Integrity

The syntax to define an integrity assertion is:

define integrity on variable **is** qual

The semantics are that when updating *variable*, insure that the corresponding *qual* must hold after the update completes.

3.2.3. Protection

The syntax to define a protection constraint is as follows:

define permit oper **on** relation [(domain-list)]
to name
[**at** term]
[**from** time **to** time]
[**on** day **to** day]
[**where** qual]

This has the semantics: let operation *oper* be performed on the domains listed in *domain-list* in *relation* to user with login name *name*. This permission only applies while logged in at terminal *term* between the *times* listed and on the *days* listed. Append the *qual* to the query. The "(domain-list)" may be omitted, as may the **at**, **from**, **on**, and **where** clauses, in which case they default to the greatest possible range of values.

The user *name* may be the keyword **all**.

3.3. Evaluation Order

The order in which these three algorithms are performed has a significant effect on the usability and functionality of the system. The relationships will be explored in pairs.

3.3.1. Views and integrity

It is generally important for views to be processed before integrity assertions, since otherwise defining views might accidently subvert the integrity mechanism. The integrity assertion declaration module could apply the view algorithm to the assertion, but this could have unexpected and possibly bizarre side effects on other queries.

3.3.2. Views and protection

The implications of choosing evaluation order for the view and protection mechanism are severe. Executing protection before views (i.e., protecting views) gives a finer granularity to the protection mechanism. For example, a database administrator could create the view that is the average salary of employees, and then grant permission on that view while denying permission on the employee relation, thereby prohibiting users from finding other aggregate information (e.g., minimum salary, or the count of the number of employees). This is the approach taken by System-R [GRIT76].

However, this has pitfalls. A database administrator could create a view to a sensitive relation without setting the protections properly, thereby subverting the protection mechanism without realizing it. Also, a single piece of data might be accessible through one view but not another. For example, consider:

```
range of e is employee
define view V1 (e.name, e.dept, e.salary)
where e.dept = "toy"
define view V2 (e.name, e.salary, e.manager)
where e.age < 30
range of v1 is V1
define permit retrieve of v1 to all
range of v2 is V2
define permit retrieve of v2 to all
```

The salary of a particular employee may be available via one of the views but not the other. On the other hand, the statements:

```
define permit retrieve of e (name, dept, salary)
where e.dept = "toy"
define permit retrieve of e (name, salary, manager)
where e.age < 30
```

will always yield the desired data in a single query.

3.3.3. Integrity and protection

Protection constraints can never have any effect on the integrity algorithm, since protection never changes the set of domains being updated, and integrity only applies to domains being updated. However, integrity assertions can effect the operation of the protection system, since they may add domains to the reference set of the query. For this reason, protection should be performed before integrity.

4. IMPLEMENTATION

This section describes the changes and additions necessary to INGRES to support the query modification system.

4.1. System Catalogs

Several system catalogs were added to each database to support query modification.

4.1.1. Tree catalog

The tree catalog contains the query trees needed by the various query modification subsystems. As such, it is not particular to any of the modules. It is of the format:

treerelid	c12	relation name
treerelowner	c2	relation owner
treeid	i2	internal name
treeseq	i2	sequence number
treetype	c1	type information
treetree	c100	contents of tree

Treerelid and treeowner uniquely identify the relation that this tree applies to. Treetype is 'p', 'i', or 'v' for a protection constraint, integrity assertion, or view definition respectively. Within a relation and type, treeid identifies the tree. The concatenation of treerelid, treerelowner, treetype, and treeid uniquely identify the query tree.

The treetree domain contains the text of the tree. It is actually not of type character: this is the closest approximation (i100 would be preferred). Since trees can be longer than 100 bytes, the treeseq field is a sequence number within a group of tuples defining a tree.

In early versions of INGRES, character type domains where defined to be blank-padded. The access method routines that could insert tuples turned all null-terminated strings into blank-padded domains. It was

necessary to remove this "feature", since null bytes are legal in trees. Fortunately, this was anticipated well in advance: release 6.1 changed the blank-padding code to give an error if a null byte appeared, and the modules that created values insured that character domains were blank-padded before insertion.

4.1.2. Protect catalog

There is one tuple in the *protect* catalog for each protection statement that has been issued. The format of this catalog is:

prorelid	c12	relation name
prorelown	c2	relation owner
propermid	i2	sequence number
prouser	c2	user to which this applies
proterm	c1	terminal ident
proopset	i1	operation set
protodbgn	i2	beginning time of day
protodend	i2	ending time of day
prodowbgn	i1	beginning day of week
prodowend	i1	ending day of week
prodomset	i16[7]	domain set permitted
protree	i2	tree sequence number
proresvar	i1	Resultvar number in tree

Prorelid and prorelown define the relation that this constraint applies to. The propermid domain identifies this statement so that it can be removed later. Prouser is an internal user code; a value of blank means that the constraint applies to everyone. Proterm gives the terminal that the the constraint applies to; blank means all. Proopset tells what operations the constraint applies to. Times of day are stored as minutes-since-midnight. Days of the week are stored with zero = Sunday. If protree is negative, there is no qualification associated with this protection constraint; otherwise, it is a link to the treeid field in the tree catalog for the qualification associated with this constraint. The entries zero and one are reserved to

[7]Since INGRES does not support an i16 type, this (and other similar domains in other relations) is implemented as four i4 domains, named prodomset1 through prodomset4.

mean the special cases **permit all to all** and **permit retrieve to all** respectively; these special cases do *not* have entries in the protect catalog; Status bits in the relation catalog define these cases (described in more detail below). Proresvar tells which variable in the tree is being constrained.[8]

4.1.3. Integrities catalog

This catalog has one tuple for each integrity assertion known to the system. It contains:

intrelid	c12	relation name
intrelowner	c2	relation owner
intdomset	i16	set of domains affected
inttree	i2	tree sequence number
intresvar	i1	variable affected in tree

The intdomset field is for efficiency only; it permits us to avoid adding assertions to the qualification of the query that cannot possibly fail. Other domains are similar to the corresponding fields in the **protect** catalog.

Since currently INGRES only supports single variable assertions, the intresvar field is not needed; it is included for generality.

4.1.4. Relation catalog

The relation catalog has not changed in structure to add query modification. However, the semantics of certain fields have been extended. The relevant parts of the relation catalog are:

---

[8]Since two-variable constraints with both variables ranging over the same relation are allowed, it is important to tell which variable is which. For example, take the constraint:

```
range of e is employee  /* employees */
range of m is employee  /* managers */
define permit replace on e (salary) to all
where e.number = CURRENT_USER
and e.salary > m.salary
and e.manager = m.number
```

(which says, permit all employees who earn more than their managers to update their own salaries.) It is critical to store that fact that variable e is being constrained instead of m; reversal would change the semantics to read, permit all employees who earn more than their managers to update their *manager's* salary.

relid	c12	relation name
relowner	c2	relation owner
relspec	i1	storage mode
relindxd	i1	set if indexed
relstat	i2	status bits
relsave	i4	purge date
reltups	i4	number of tuples
relatts	i2	number of attributes
relwid	i2	tuple width
relprim	i4	number of primary pages

This catalog has one tuple for every relation known to the system. The main extension is to the relstat field. The following bits have been added:

S_VIEW — If set, this relation is a view. The semantics of several of the other domains change, as described below.

S_VBASE — If set, this relation is the base for a view.

S_INTEG — If set, there are integrity constraints defined on this relation.

S_PROTUPS — If set, there are tuples existent in the *protect* relation that apply to this relation.

S_PROTALL — If *clear*, this relation has **permit all to all** permission.

S_PROTRET — If *clear*, this relation has **permit retrieve to all** permission.

S_BINARY — If set, assume that there may be non-printable characters in characters domains. These characters will be printed in octal. This bit is included for efficiency reasons, since it is much less expensive to print if this check need not be made.

The S_PROTALL and S_PROTRET flags are asserted low to maintain compatibility with the semantics of existing databases. It was felt that it would be wiser to avoid having existing relations suddenly become

inaccessible.

Flags S_INTEG, S_PROTUPS, S_PROTALL, S_PROTRET, S_BINARY, and S_VBASE are provided for efficiency only and are not strictly necessary in the implementation. The first four enable the algorithms to avoid scanning system catalogs for certain common special cases. The S_VBASE allows INGRES to avoid **destroy**ing a relation that is the base of a view.

If the relation is a view (i.e., the S_VIEW bit is set), the domains relspec, relindxd, reltups, and relprim are unused. The domains relid, relowner, relstat, relatts, and relwid are unchanged.

4.2. Code Structure

Query modification is implemented as a process following immediately after the parser. It reads query trees in conjunctive normal form as produced by the parser and produces modified trees, also in conjunctive normal form.

The process can be broken into four main parts:

(1) Code incidental to query modification. This includes the code to read the input pipe and decide if a request should be processed (*i.e.*, is a query versus some other request, such as a database utility request), code to read and build trees, code to write trees, and code to support the environment (*e.g.*, handle interrupt signals).

(2) Code to actually perform query modification. This is in four sections: (a) the view module, (b) the integrity module, (c) the protect module, and (d) the code to dispatch the query to these three modules.

(3) Code to define new query modification statements in the database. This is in four sections: (a) code to define views, (b) code to define integrity assertions, (c) code to define protection statements, and (d) code shared

among these three to store a definition tree into the database.

(4) Utility code used by (2) and (3) above. These include codes to issue other queries needed during processing, routines to manipulate the range table, routines to scan trees for various conditions (*e.g.*, existence of aggregates or particular VAR nodes), routines to build bit vectors representing sets of variables referenced, routines to concatenate qualifications, etc.

4.3. Views

4.3.1. Update anomalies

There are several updates that can not be performed on certain views, necessitating that the update be aborted. Some updates can be performed, but will have anomalous results. INGRES only allows updates that can be guaranteed to have consistent results, defined by:

> An update is *consistent* if and only if the result of performing the update on the view and then materializing[9] the view is the same as the result obtained by materializing the view and then performing the update.

The following discussion summarizes these problems.

4.3.1.1. syntactic problems

A view may have a domain defined as a non-simple value (that is, not a simple attribute). A **replace** on this domain and any **append** will require this domain to take on a value. For example, if the view is defined as

```
range of b is baserel
define view v1 (d = b.x + b.y)
```

then the update

[9] *i.e.*, turning the view into a physical relation, perhaps by issuing the query:
```
range of v is view
retrieve into vtemp (v.all)
```

append to v1 (d = 5)

would expand to

append to baserel (x + y = 5)

which cannot be resolved.

All **append** statements must be rejected in this case, since even domains not mentioned are set by default to zero. Also, all **delete** statements must be rejected if the view includes an aggregate, since they might modify the value of the aggregate.

There is a range of cases that might be handled. In particular, any domain that is defined as an invertible function could be made to work, but we do not feel this is cost effective at this time.

4.3.1.2. disappearing tuple anomaly

If a view has a qualification, any update of any domain mentioned in that qualification allows even non-duplicate tuples to disappear. For example, consider the view defined as

```
range of b is baserel
define view v2 (d = x.a)
where x.a = 4
```

The tuple added by the update

append to v2 (d = 7)

will not appear in the view (even though it will be appended to *baserel*).

Under these conditions all updates except **deletes** must be disallowed.

4.3.1.3. join problems

Views defined as joins have many update problems. For example, take the relations R1(a, b) and R2(b, c) as shown in figure 3(a). The view

R1	a	b
	7	0
	8	0

R2	b	c
	0	3
	0	4

(a)

V3	a	b	c
	7	0	3
	7	0	4
	8	0	3
	8	0	4

(b)

Figure 3: Relations & equijoin.

defined as the equijoin

```
range of x is R1
range of y is R2
define view v3 (x.a, x.b, y.c)
where x.b = y.b
```

is shown as it would appear materialized in figure 3(b).

Assuming we have

```
range of v is v3
```

the update

```
delete v where v.a = 7 and v.c = 4
```

will produce the relation shown in figure 4(a); the update

```
replace v (v.b = 1)
where v.a = 8 and v.c = 3
```

will produce the relation shown in figure 4(b), and the update

```
append to v3 (v.a = 6, v.b = 0, v.c = 4)
```

will produce the relation shown in figure 4(c). All the relations shown in

V3	a	b	c
	7	0	3
	8	0	3
	8	0	4

(a)

V3	a	b	c
	7	0	3
	7	0	4
	8	1	3
	8	0	4

(b)

V3	a	b	c
	7	0	3
	7	0	4
	8	0	3
	8	0	4
	8	0	4

(c)

Figure 4: Anomalous updates.

figure 4 are unexpressible in R1 and R2 without modifying the definition of the view in some arbitrarily complex manner.

To avoid these anomalies, we must disallow any update on a view with more than one variable in the target list.

If we had more information about the semantics of the join domains consisted of unique values, we could relax this constraint considerably. For example, suppose that the view *empdept* consisted of the the *employee* and the *dept* relations joined on *dept#* (*i.e.*, each tuple contains all the information on each employee and the department in which they work). Since we have semantic information about this view, we know that

updates of (*e.g.*) employee names do not affect the department information, and changing the manager of the department does not change any information regarding the employees. However, INGRES does not maintain this information.

4.3.2. View algorithm

The following algorithm processes views:

(1) Perform steps 2 – 7 until no view substitution takes place, *i.e.*, until the query is totally in terms of instantiated relations.

(2) Perform steps 3 – 7 for all relations in the query that are views.

(3) If the query is an update and the view defines a domain using an aggregate, abort the query. This prevents a query from assigning a value to an aggregate.

(4) If the query is not a retrieve, and if the variable we are doing view replacement for is the variable being updated, perform the following steps:

   (a) If this query is a **delete** or **append**, and if the view is defined over more than one relation, abort the query since it would take at least two queries to satisfy it.

   (b) If a **delete**, go on to step 5.

   (c) If an **append**, and the view has a qualification, abort the query because we cannot guarantee that the tuple will appear in the view – *i.e.*, the disappearing tuple anomaly.

   (d) For each result domain in the query, locate the definition for that domain in the view and perform steps (e) – (g).

   (e) If the definition is not a simple variable, abort the query, since it would require setting an expression to a value.

(f) If this variable appears in the qualification of the view, abort the query for the same reason as (c) above. This test is here because the requirements for **replace** are less strict than for **append**.

(g) Replace the result domain by the substitution in the view definition.

(5) For each variable mentioned in the query that is for the current view, replace the variable by its definition.

(6) If the query is a **replace**, and if the new query ranges over more than one relation, abort the query because it could result in a non-functional update.

(7) Append the qualification from the view to the query tree.

(8) After all view processing is complete, renormalize the tree into conjunctive normal form.

4.3.3. Defining views

Defining a view is trivial: there are no exceptional conditions that can occur. The definition is stored directly into the database schema.

Views defined on views are left in this form. It might be advantageous to do the view expansion at definition time, thereby avoiding repeated expansion when used. This has one disadvantage: when the user prints the definition of the view, it will not match the definition input, which could be confusing. Since we anticipate views on views will be a rare occurrence, we feel the efficiency issue is not of major import.

4.4. Integrity Assertions

4.4.1. Integrity algorithm

The integrity processor is extremely straightforward. The processor does nothing if the query is not an update or if there are no integrity

assertions on the relation being updated (determined by the S_INTEG bit in the relstat field of the relation catalog). Also, since single-variable aggregate-free assertions can never be violated by deleting tuples, **delete** statements are passed through unmodified.

A domain set is built that can be compared against the domain set stored in the *integrities* catalog. Assertions that can not possibly be violated are excluded[10]. The integrity qualification is then merged into the qualification of the query.

Only single-variable aggregate-free assertions are supported, owing to the difficulty and cost of supporting more complex assertions. In particular, any other form of assertion requires more complex processing, up to re-running the assertion as though the update had been made in advance to assure that the update will be acceptable. For example, an assertion such as:

```
range of e is employee
range of d is department
define integrity on e is e.dept# = d.dept#
```

(meaning that every employee must be in a department that exists) requires a scan of the *department* relation every time an employee is hired or transferred to determine existence.

Updates that can not be made because they violate an assertion are silently ignored. There are two alternative approaches that could be taken:

(1) A separate query could be run in advance to retrieve the tuples that will fail the update. Two resolutions could be chosen: the violations could be stored in another relation (or printed) and the update could be run to completion, or the update could be aborted until all

[10] **appends** must always have all assertions included, since all domains are always updated.

violations were fixed. This would require running another query on every update, which could be prohibitively expensive.

(2) A new node type could be added to the tree representing a qualification that *must* be true. If such a qualification were to fail, an error message would be issued. This requires considerable perturbation in the query processing modules, and could slow execution on all queries (including **retrieves**, and updates of relations with no integrity assertions).

4.4.2. Integrity definition

During integrity assertion definition, checks must be made to insure that the assertion is single-variable and aggregate-free. Also, checks are made to insure that assertions are not defined on views (since the evaluation order is such that assertions on views would never be honored) and that the user defining the assertion owns the relation being constrained. The assertion is then inverted, and a query is issed to check that the assertion already holds. If it does not, the definition is aborted. Only after satisfying these conditions is the assertion added to the database.

It would be possible to apply the view algorithm to the integrity assertion to translate assertions on views into assertions on physical relations. However, this could result in surprising assertions that would show through in possibly disturbing ways.

4.5. Protection Constraints

4.5.1. Protection algorithm

The protection system first looks through the set of variables that are referenced in the query and creates a set of variables that might have violations. Variables ranging over relations owned by the user executing the

query, relations with the S_PROTALL bit asserted (clear), and relations with the S_PROTRET bit asserted during a retrieve are *not* candidates for violations. These three cases are expected to handle most queries.

Of the remaining variables, each is taken one at a time. Four sets are created representing usage in various contexts. These sets are:

uset — The set of domains updated. In a **retrieve**, this set will be empty.

rset — The set of domains retrieved to the user. This includes domains involved in some computation, including aggregate functions and aggregates with qualifications.

aset — The set of domains with the aggregate value retrieved. No qualifications are allowed.

qset — The set of domains used in any context in a qualification.

From these sets, it is possible to create a set of operations performed in this query. This will be at most one of {**delete**, **append**, **replace**} together with any of {**retrieve**, **aggregate**, **test**}. This set will represent the set of operations that have not yet had at least one protection statement qualify – necessary to carry out a "default to deny" policy.

The processing of known protection statements then begins. For each statement regarding the relation we are currently processing, we check if the statement applies – if not it is ignored[11]. A statement applies if it is specified as being for the current user on the current terminal, and the current time falls within the specified limits.

This statement is then examined to see what operations it might give permissions for. If it won't satisfy anything in the query, it is thrown out.

[11]Since protection statements only *add* permissions, it is never dangerous to throw a statement away.

The process of "satisfying" requires that the domain set of the statement intersect the domain set of the query in an appropriate context.

At this point we know this protection statement will give us useful permissions. The qualification associated with the protection statement is disjoined to all qualification lists for operations to which this statement applies. For example, if the protection qualification is

where e.manager = CURRENT_USER

and the domain *manager* is used for update and test, then that qualification will be appended to the qualifications being built for update and test, but not the qualifications being built for retrieve and aggregate.

Since at least one protection clause has applied to this query, the set of operations not yet satisfied is reduced by the set of operations satisfied by this protection statement.

When all applicable protections statements have been processed, the operation set is checked. If this set is not now null, the query is aborted for lack of permission. If the query is ok, then the qualification lists created above are conjoined to the user query. The query is then normalized to conjunctive normal form and returned.

4.5.2. Defining protection clauses

The definition phase of protection first checks all the possible parameters for validity (*e.g.*, users must exist, times must be in range, etc.) and builds the set of domains referenced. Protection statements on views are disallowed. The special cases **permit all to all** and **permit retrieve to all** are identified and extracted. Finally, the necessary inserts to system catalogs are made.

4.6. Evaluation

4.6.1. Statistics

The following chart shows the size of the query modification code.

Process Size Breakdown		
line	function	bytes
1	Total for process	50920
2	Just query modification	16394
3	Non QM code (1–2)	34526
4	Access method code	13714
5	overhead (3–4)	20812

Everything from line 3 down results from utility libraries that can be shared with the rest of INGRES in a large address space machine such as a VAX. For this reason, the rest of this computation will use the total for query modification alone as the base.

The following breakdown shows what percentage of the code is in use for various functions.

Query Modification Code Size Breakdown		
Module	bytes	percent
Views	910	5.5%
Protection	3398	20.7%
Integrity	920	5.6%
dispatching	76	0.5%
TOTAL unshared QM	5304	32.4%
View definition	380	2.3%
Protection definition	2516	15.3%
Integrity definition	1080	6.6%
shared definition	992	6.1%
TOTAL definition	4968	30.3%
shared QM specific code	3836	23.4%
environment support	2286	13.9%
TOTAL shared	6122	27.3%
GRAND TOTAL	16394	100.0%

4.6.2. Match with design goals

4.6.2.1. maximize code overlap

The query modification code was split approximately equally three ways between query modification processing, definition, and shared utilities. This fell short of the desired goal of one half shared code; however, the overlap is skewed by having constraint definition in the same module.

4.6.2.2. minimize system impact

The INGRES parser was modified to support the syntax required by query modification. The following table summarizes the productions added.

Production Summary	
views	5
protection	33
integrity	10
shared	2
TOTAL QM	50
TOTAL in grammar	201

Although query modification uses almost 25% of the productions in the grammar, the net effect on the parser is somewhat smaller; most query modification actions are small relative to the size of QUEL actions.

The parser was further modified to pass an extended range table to query modification so that it would be unnecessary for query modification to scan the system catalogs for most common cases.

Most of the database utilities (such as **print**, **copy**, and **destroy**) had to be modified to correctly process protection and views. These changes where minor.

No changes were required in the query processing modules.

4.6.2.3. use standard interfaces

As noted above, the range table was extended. This extension was totally downward compatible. This was the only change in the interface. We feel this goal was met.

4.6.3. Lack of detection of violations

Both the protection and integrity systems should be modified to catch violations. Logging of protection violations would help discourage attempts at compromising the system. Integrity violations should be flagged so that at least the user would be aware that the update has not been made.

Detection of violations of single-variable constraints would not be difficult if changes where made to the query processing modules. The simplest technique would attach qualifications to the query tree which would be required to be **true** for all tuples otherwise satisfying the qualification. This technique has the disadvantage that it does not extend to the multivariable case; use of this technique to catch protection violations would require limiting protection qualifications to single-variable aggregate-free predicates.

Certain protection violations are detected before query processing. Violations of this type can be easily logged without changes to the rest of INGRES.

Integrity violations could be detected by first running the query with the integrity clause inverted. This is easy to do, but would tend to increase the time needed to process any update by a factor of at least two.

5. CONCLUSIONS

Query modification has been used successfully to implement views, protection constraints, and integrity assertions in INGRES. Although there are several additions that should be made before this particular implementation would be salable in a production environment, the general technique is viable, being flexible, having little impact on the rest of the system, and functionally powerful.

Future areas of possible research include expanding the range of possible updates to views and expanding integrity assertions beyond single-variable aggregate-free.

6. REFERENCES

[CODD71] Codd, E.F., "A Data Base Sublanguage Founded on the Relational Calculus", *Proc. 1971 ACM-SIGFIDET Workshop on Data Description, Access, and Control*, San Diego, California, November 1971.

[EPST77] Epstein, Robert S., "A Tutorial on INGRES", Electronics Research Laboratory Memorandum UCB/ERL M77/25. University of California, Berkeley, December 1977.

[GRIT76] Griffiths, Patricia P., and Wade, Bradford W., "An Authorization Mechanism for a Relational Data Base System". IBM Research Report RJ 1721 (#25154), February 1976.

[HELD75] Held, G.D., Stonebraker, M.R., and Wong, E., "INGRES – A Relational Data Base Management System", *Proc. 1975 National Computer Conference*, AFIPS Press, 1975.

[STON76] Stonebraker, M.R., Wong, E., Held, G.D., and Kreps, P., "The Design and Implementation of INGRES", *ACM Transactions on Database Systems*, *1*, 3.

[RITC74] Ritchie, D.M., and Thompson, K., "The UNIX Timesharing System", *Communications of the ACM*, *17*, 7 (July 1974).

[WOOD79] Woodfill, J., *et al*, "The INGRES 6.2 Reference Manual", Electronics Research Laboratory Memorandum UCB/ERL M79/43. University of California, Berkeley.