VERSION MODELING USING PRODUCTION RULES IN THE POSTGRES DBMS

by

Lay-Peng Ong

Memorandum No. UCB/ERL M91/51

5 June 1991

VERSION MODELING USING PRODUCTION RULES IN THE POSTGRES DBMS

by

Lay-Peng Ong

Memorandum No. UCB/ERL M91/51

5 June 1991

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

VERSION MODELING USING PRODUCTION RULES IN THE POSTGRES DBMS

by

Lay-Peng Ong

Memorandum No. UCB/ERL M91/51

5 June 1991

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

1 Introduction

As noted in [STON80], versions (also called hypothetical relations) can be used to support "what if" scenarios in the database, and offer a mechanism for manipulating live data without fear of corrupting the database. In general, versions are implemented through differential files (deltas) [SEVR76], either via forward deltas [STON81, WOOD83], or via backward deltas [KATZ82]. The main drawback of version systems is that the semantics are hard-coded, and thus such systems are difficult to extend when new semantics are desired. Furthermore, no single system is general enough to subsume all of the proposals. This thesis shows that production rules in a database system can be used to implement versions, and that such rules are general and flexible enough to accommodate any new semantics that the user might want.

The organization of the thesis is as follows: Section 2 introduces the concept of using production rules in a database system to implement versions. Section 3 describes how the POSTGRES [STON86] database management system (DBMS) supports features beyond that provided by normal relational DBMS that are essential for version processing. Section 4 details the operations of such a version system through examples. Section 5 discusses perfor-

mance issues of such a system. Finally, section 6 discusses future extensions to the system.

2 Implementation

Previous proposals for implementing versions ([STON80], [STON81], [AGRA82], [KATZ82], [WOOD83]) each define slightly different version semantics. However, none of the above proposals have been able to provide a general enough framework to encompass all the other proposed systems. The POSTGRES Version System (PVS) built on top of POSTGRES, a DBMS under development at the University of California at Berkeley, uses production rules to implement versions. The next few sections will detail the implementation of versions using production rules, and show that these rules can be easily modified to implement different version semantics.

While [KATZ88] describes a version as a semantically meaningful snapshot of an object in time, others have extended versions to include those that are not snapshots; that is, such versions should be optionally able to "see" changes in the base relation. The user decides at the time he creates a version whether or not the version is a snapshot of the base relation. The PVS sup-

ports both types of versions. In addition, versions can be implemented using either forward or backward deltas (differential files). This section describes the implementation of versions using production rules.

2.1 Syntax

The syntax for creating a version is:

```
create [direction] version Vname from Bname[abstime]
```

where *direction* by default is *forward* for forward deltas, but it can be set to *backward* for backward deltas. *Vname* is the name of the version while *Bname* is the name of the base relation. If the optional *abstime* clause is specified, then the version created is a snapshot of the base relation as of the time specified.

2.2 Forward Deltas

Versions implemented via forward deltas are logical entites made up of 3 physical relations: **base**, **v_add**, and **v_del** (see figure (1)). The **base** relation is the relation being versioned. The **v_add** relation stores the net tuples added (this includes replaces of tuples in the base relation) to the version,

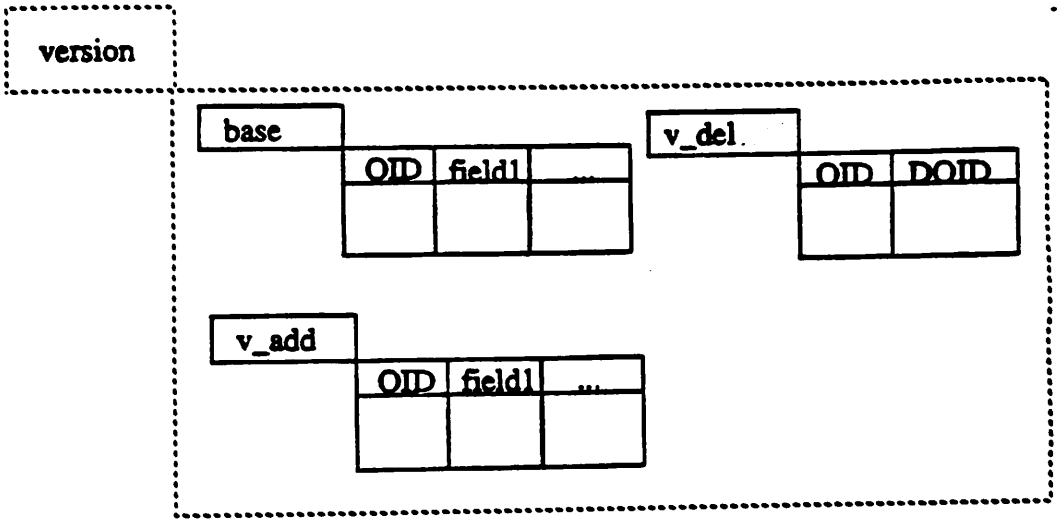

Figure 1: A version is made up of three physical relations.

while the **v_del** relation has a single attribute (DOID) which stores the OIDs of tuples from the **base** relation which have been "deleted" or "replaced" in the version. Together, the **v_add** and **v_del** relations form the delta relations of a version, and are created whenever the user creates a version.

Versions maintained via forward deltas allow 2 alternative update semantics:

1. Versions which can "see" changes made in the base relation; that is, the changes are propagated.

2. Versions which are snapshots of a base relation at a point in time; In this case, any changes to the base relation after that point in time will not be reflected in the version.

In addition, the update semantics for both types of versions are such that

the **base** relation is never modified by updates, deletes or appends to the version.

2.2.1 Version Creation

The query, "create [forward] version v1 from base", performs the following functions:

1. Creates a dummy relation named "v1" .
2. Creates the delta relations(v_add and v_del).
3. Define the set of rules that govern the semantics of versions.

The first command creates a dummy relation with the same schema as the base relation but with no tuples in it. It is necessary to create such a relation because the PRS2 rules can only be defined on, and triggered off, existing relations. The following section describes the rules that are defined.

2.2.2 Version Semantics

This section describes the semantics of versions implemented via forward deltas. In POSTGRES, the semantics of such versions are:

$$version = (base - net\_deleted\_from\_base) \cup net\_added$$

Given a version **v1**, the **retrieve** rule defined for it transforms a retrieve on **v1** into a retrieve of all qualifying tuples in **v_add** $\cup$ (**base** $-$ **v_del**). Thus, a retrieve to a version is transformed into a retrieve on the union of **v_add** and **base**. The processing of union queries is described in section 3.3. Similarly, a rule is defined to transform appends to **v1** into appends to **v_add**.

Deleting tuples from a version is more complex. Since tuples in a version can reside in either the **base** or **v_add** relations, there are two cases to consider when we define the **delete** rule for versions:

1. Delete all qualifying tuples in the **v_add** relation.
2. Qualifying tuples in the **base** relation are "deleted" by appending their oids to the **v_del** relation.

The second case is necessary because the semantics of versions are such that modifications to versions are not propagated to the base relation. Thus, tuples in **base** are never actually deleted. Finally, the **update** rule for **v1** performs the following:

1. Replace all qualifying tuples in the **v_add** relation.
2. Invalidate all qualifying tuples in the **base** relation by appending their oids to the **v_del** relation.
3. If this is the first time a tuple from the **base** relation is updated in the version **v1**, append a copy of this tuple to **v_add**.

Once again, tuples in the **base** relation are never actually modified. A detailed listing of the rule definitions is shown in appendix A.

2.3 Derivatives

As noted in [KATZ82], versions implemented using forward deltas necessarily results in an inefficient access to the most recent version which must be painstakingly reconstructed from the base and the forward deltas from intervening versions. This is an especially crippling deficiency in systems where typically, the "current" or "working" version is often the most recent version. A way to provide efficient access to the latest version is to implement versions via backward deltas or "negative" differential files [KATZ82].

In the versioning scheme using backward deltas, changes to the base are made in place and the old values are recorded in the deltas. The old version (base) is defined by the new version and the differential files in [KATZ82] as:

$$base = (new\_version \cup total\_deleted) - total\_inserted$$

In the PVS, another method (derivative) is used to provide efficient access to the latest version. Such versions are called "derivatives" in POSTGRES. The command, "define derivative v1 from base", performs the following 2

functions:

```
rename base to v1

retrieve into base (v1.all) where 1 = 2
```

The first command renames the current relation, **base**, to be **v1** on which all updates and queries on the new version will be performed. Indices defined on **base** are preserved in **v1**. since the rename command preserves indices. The second command creates a dummy relation (one with attributes and relevant structural information but physically containing no tuples) with the same name as the relation being versioned so that rules can be defined on the "older" version. Since changes to the new version are made in place. no special rules are needed for retrievals or updates to it. However. a retrieval rule is needed when one wants to access the "older" version. The retrieval rule "freezes" the "older" version at the time when the "new" version is created. This ensures that subsequent changes to the version will not affect the previous versions, and that the previous version cannot be modified. Thus the semantics for versions implemented via backward deltas are that updates are only allowed for the most current version, and not for the previous versions. Subsequent versions will perform similar renamings and rule definitions.

3 Support

This section describes the underlying support needed by the PVS. As a successor to INGRES, one of the aims of POSTGRES is to provide additional functionality that the relational model lacks. Among the advanced features provided by POSTGRES are:

1. The concept of having an object identifier (OID) for each tuple which is globally unique within a database.

2. A rule system that supports production rules.

3. The concept of historical data.

While the above mentioned features provide the framework from which the version system is implemented, that alone is not enough for the system also requires support for union queries, and semantic optimization is necessary in order for the system to perform efficiently. Thus, POSTGRES needs to be extended to support the last two requirements.

3.1 The POSTGRES Rules System.

The POSTGRES rules system (PRS2) [STON90] follows the production rule paradigm of a rule being an event-action pair where a typical rule looks like:

```
ON <event/condition> THEN DO [INSTEAD] <action>
```

If the "INSTEAD" keyword is used, the original query which triggered the rule is not executed, and the action of the rule is executed in its place. If "INSTEAD" is not used, then both the query that triggers the rule and the action of the rule are executed.

In PRS2, an event can be any one of the usual retrieve, replace, append, or delete queries on a relation, and an action is any set of Postquel queries which optionally references the triggering event-relation. In addition, both event and action clauses can contain references to **current** and **new** wherever normal tuple variables are allowed. The semantics of **current** and **new** are as follows: When a tuple, **A**, is accessed (i.e., retrieved, replaced, or deleted), there is a tuple variable called **current** that refers to **A** . In addition, in the case of an append or replace, there is also a tuple called **new** which contains the new values for **A**. While POSTGRES is not the only data manager that has a rules subsystem (for instance, both Iris and Starburst have similar rules

subsystems). it is the only one whose triggering events include the "retrieve" event.

The PRS2 rules system is made up of 2 rule subsystems: The Query Rewrite System (QRS)[GOH90] and the Tuple Level System (TLS). The user can optionally define rules using either system by simply changing a single keyword. The following is an example of a rule in the PRS2 using the QRS to keep an audit trail of all employees who have a raise that causes their salaries to be $50000 or more.

```
define rewrite rule audit_raise is
on replace to emp.salary do
append audit ( name = CURRENT.name,
old_sal = CURRENT.salary, new_sal = NEW.salary )
where CURRENT.salary < 50000
and NEW.age >= 50000
```

To define the same rule using the TLS subsystem, the user need only change the keyword "rewrite" to "tuple", and the above rule will become a tuple level rule.

3.2 Historical data.

POSTGRES supports the notion of historical data, that is, data that is not part of the "current" state of the database. For example, tuples which have

been logically deleted over time are not part of the "current" state of the database. but are instead part of some "past" database state. Similarly, "old" values of tuples which have been replaced are also part of that "past" database state. In POSTGRES, tuples are never physically overwritten or deleted. Instead, both the old and new versions of the tuples are retained. but with the old tuple invalidated. Each tuple has a "timespan" during which in was valid. Thus in Postgres, it is possible to query historical data by simplying specifying the timespan you are interested in. For example, the following time-range query finds all employees that earned more than $10000 in the 12th of January 1989:

```
retrieve (e.all) from e in emp["Jan 12 1989"]
where e.salary > 10000
```

3.3 Union Queries

Queries on versions implemented via forward deltas are transformed by the QRS into union queries. Union queries are queries of the form:

retrieve ({**A** | **B**}.*all*) **where** {**A** | **B**}.*attribute* = *foo*

The relations **A** and **B** form the union set of the query. A scan on a union set is called a union scan, while a join is called a union join. Currently,

POSTGRES has no support for union queries, thus it needs to be extended in order to support version processing. This section describes the semantics of such queries.

A primary aim when extending POSTGRES to support union queries is to make minimal changes to the DBMS. The next few sections describes in greater detail the parser and planner preprocessor extensions.

3.3.1 Parser Extensions

The primary change to the parser involves the addition of a relational union operator, denoted by the symbol "|", to POSTQUEL. For example, the following query retrieves the names of everyone in the emp and student emp (stud_emp) relation who makes more than $10000:

retrieve (*all_emp.all*)
from *all_emp in* (*emp* | *stud_emp*) **where** *all_emp.salary* > 10000

3.3.2 Planner Preprocessor Extensions

The planner preprocessor intercepts all union queries and converts them into an equivalent set of "normal" queries that will be processed by the planner and executor. The algorithm for transforming a union query into its equivalent set of non-union queries is as follows: Given a union scan over N

relations. N simple scans will be produced, one for each of the N relations. For example, suppose we have the following union scan over the relations A,B and C. which retrieves everyone named "foo" from relation A, and everyone named "foo" who is also 50 years old from relations B and C:

retrieve ($\{A \mid B \mid C\}.all$) **where** $\{A \mid B \mid C\}.name$ = "foo"
and $\{B \mid C\}.age = 50$

The planner preprocessor will split this union scan into the following 3 simple scans:

retrieve ($A.all$) **where** $A.name$ = "foo"
retrieve ($B.all$) **where** $B.name$ = "foo" **and** $B.age = 50$
retrieve ($C.all$) **where** $C.name$ = "foo" **and** $C.age = 50$

Thus. the 3 simple scans provide the same semantics as that of the original union query.

3.4 The Semantic Optimizer

In general, a union scan over N relations is transformed by the planner preprocessor into N simple scans. As noted in the previous sections, transforming a union scan into its equivalent set of simple scans results in redundant queries. This section describes the various rules for semantic optimization

that is used to improve the efficiency of the PVS. In general. semantic optimization is done in 2 phases described in the following subsections.

3.4.1 Phase 1 (P_1) Semantic Optimization

P_1 semantic optimization involves the following:

1. optimize tautological qualifications
2. optimize contradictory qualifications

Tautological qualifications are qualifications which will always be true. for example. consider the query:

retrieve (*A.all*)
from *A in emp. B in stud_emp* **where** *A.name* = "john"
and *A.oid not_in B.oid*

In POSTGRES. each tuple in a database has a OID which is globally unique within that database. The "not_in" operator is a boolean relational operator that returns true if the tuple on the left hand side of the operator is not in the specified set of tuples on the right hand side of the operator. Thus. in the above query. the qualification " A.oid not_in B.oid" will always be true and can be semantically transformed into the equivalent qualification " 1 = 1". The transformed qualification is what is known as a constant qualification. and in POSTGRES, constant qualifications need only be evaluated once per query instead of once for every tuple in the relation.

Contradictory qualifications are qualifications that will never be true. Examples include qualifications of the form:

1. " A.oid = B.oid" where A and B are 2 different relations.
2. "A.oid not_in A.oid"

These qualifications will be transformed by the semantic optimizer into the following constant qualification: "1 = 2", which is always false.

As a further refinement of to the first phase. the semantic optimization will also remove redundant queries; that is, queries whose qualifications are never satisfied. For example, the query:

retrieve (*A.all*)
from *A in emp* **where** *A.name* = "john"
and *A.oid not_in A.oid*

will be transformed by the semantic optimizer into :

retrieve (*A.all*)
from *A in emp* **where** *A.name* = "john"
and 1 = 2

Because the constant qualification is always false, the query will never return any tuples. Thus, the P_1 semantic optimizer picks out all such redundant queries the removes them, thereby saving the time it takes to plan and execute such queries. In general, about half of the generated queries for

version processing are redundant queries. Thus, P_1 semantic optimization effectively reduces the number of queries by half.

3.4.2 Phase 2 (P_2) Semantic Optimization

The idea behind the third phase of semantic optimization is the following: Consider the following query

retrieve (*A.all*)
from *A in emp*, *B in emp* **where** *A.name* = "john"
and *A.oid* = *B.oid*
and *B.salary* < 15000

which is semantically equivalent to :

retrieve (*A.all*)
from *A in emp* **where** *A.name* = "john"
and *A.salary* < 15000

Without P_3 semantic optimization, the above query results in a 2-way join on the "emp" relation. However, because of the unique properties of OIDs, and because A and B are both range variables for the "emp" relation, the qualification "A.oid = B.oid" is actually a one-to-one mapping of each tuple in emp to itself. P_3 semantic optimization removes such redundant joins and converts them to their semantically equivalent scans. In version

processing. all generated queries involving replaces and deletes to versions result in redundant joins. thus P_3 optimization will speed up such queries.

4 Examples

In this section. the concepts described in sections 2 and 3 are illustrated by showing examples of the following:

1. A simple version
2. A cascaded version
3. A version join

4.1 Simple Versions

Given the set of rules shown in Appendix A. and the data shown in figure (2). suppose we want to rename everybody named "John" to "Joe". The query will be:

```
replace v1 (name = "joe") where v1.name = "John"
```

The QRS will transform this query into:

```
replace v1_add (name = "joe")
        where {v1_add | base}.name = "john"
        and base.OID not_in v1_del.DOID
```

```
        and {v1_add | base}.OID = v1_add.OID

append v1_del(DOID = base.OID)
       where {v1_add | base}.name = "john"
       and base.OID not_in v1_deletd.DOID
       and {v1_add | base}.OID = base.OID

append v1_added(name = "joe",
     salary = {v1_add | base }.salary, ...)
       where {v1_add | base}.name = "john"
           and {v1_add | base}.OID not_in v1_add.OID
          and base.OID not_in v1_del.DOID
       and {v1_add | base}.OID = base.OID
```

Queries of the form "retrieve ($\{\mathbf{A} \mid \mathbf{B}\}.all$) where $\{\mathbf{A} \mid \mathbf{B}\}.name =$ "foo"" are called union queries. Relations **A** and **B** form the union set of the query. A scan on a union set is called a union scan. while a join on such a set is called a union join.

The above union queries will be spilt by the planner preprocessor into their equivalent sets of non-union queries respectively according to the algorithm described in section 3.3. After undergoing the 2 phases of semantic optimization described in section 3.4, the resulting query becomes:

```
replace v1_add (name = "joe")
  where v1_add.name = "john"

append v1_del(DOID = base.oid)
```

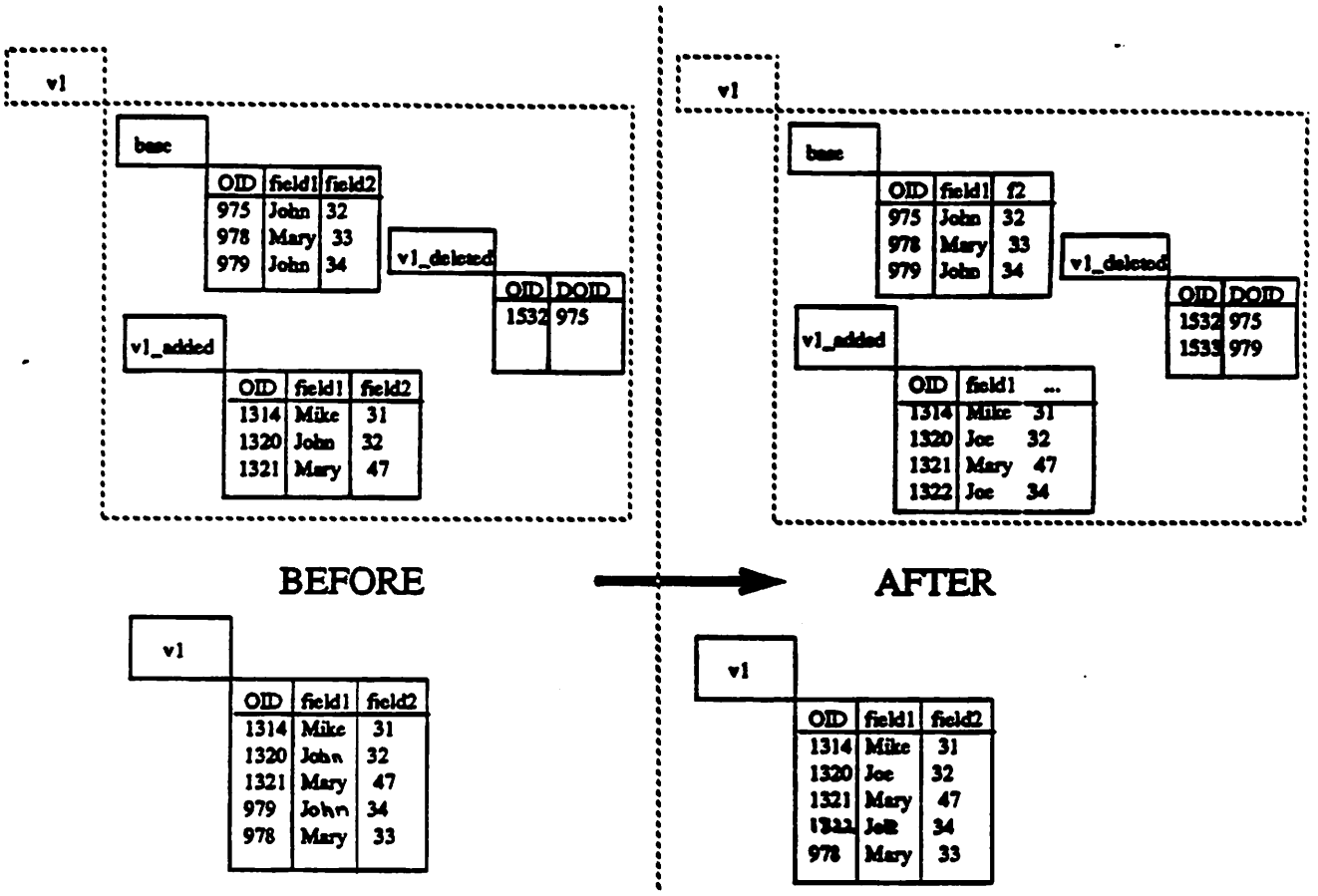

Figure 2: v1 is a version of base.

```
  where base.name = "john"
  and base.oid not_in v1_del.DOID

append v1_add(name = "joe", salary = base.salary, ... )
  where base.name = "john"
  and base.oid not_in v1_del.DOID
```

Thus three things happens on an update to a version:

1. All qualifying tuples that are in the v1_add relation are updated

2. Qualifying tuples in the base relation which are not in the v1_del rela-

tion are "invalidated" by appending their oids to the v1_del relation.

3. If this is the first time a base tuple is updated in the version, it will be appended to the v1_add relation.

The net effect of these three queries will be to replace all qualifying tuples in the version. Given that subsequent retrieves on the version will retrieve tuples from

$$v\_add \cup (base - v\_del)$$

the replace is correctly processed.

4.2 Cascaded Versions

We can also create versions of versions using the rules stated in section Appendix A. When we create a version of **v1** named **v2**, the schema of **v2** looks like figure (3).

Now suppose we retrieve everyone is v2 whose salary is less than $5000 :

```
retrieve (v2.name) where v2.salary < 5000
```

This query will trigger the retrieve rule for v2 and be transformed by the QRS into :

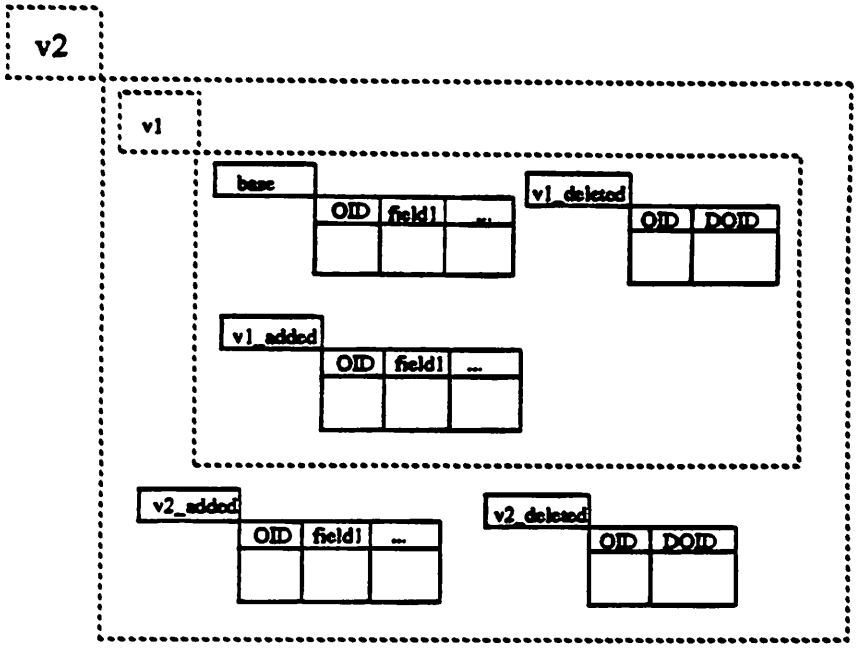

Figure 3: a version **v2** of a version **v1**

```
retrieve ({ v2_add | v1}.name) where
  {v2_add | v1}.salary < 5000
  and v1_OID not_in v2_del_DOID
```

Since v1 is itself a version, the retrieve rule for v1 will be triggered, and the query gets rewritten by the QRS into :

```
retrieve ({v2_add | v1_add | base}.name) where
  {v2_add | v1_add | base}.salary < 5000
  and {v1_add | base}.oid not_in v2_del.DOID
  and base.oid not_in v1_del.DOID
```

The union query will be split by the planner preprocessor into retrieves on the individual relations v2_add, v1_add, and base:

```
retrieve (v2_add.name) where v2_add.salary < 5000

retrieve (v1_add.name) where v1_add.salary < 5000
  and v1_add.oid not_in v2_del.DOID

retrieve (base.name) where base.salary < 5000
  and base.oid not_in v2_del.DOID
  and base.oid not_in v1_del.DOID
```

The net result of evaluating these 3 queries will be to a retrieval of all tuples in v2 which satisfy the user qualification. Thus, we have achieved the desired retrieval semantics for cascaded versions.

4.3 Version Joins

As shown in the previous sections, a scan on a version is transformed by the QRS into a set of union scans. Similarly, a join between 2 versions is transformed by the QRS into a set of union joins. Consider the following query on emp1 (a version of emp) and dept1 (a version of dept) which lists the employee's name and the floor he works on:

```
retrieve (emp1.name, dept1.floor)
  where emp1.dname = dept1.name
```

This query is transformed by the QRS into :

```
retrieve ({emp1_add | emp}.name, {dept1_add | dept}.floor)
```

```
  where {emp1_add | emp}.name = {dept1_add | dept}.floor
  and emp.oid not_in emp1_del.DOID
  and dept.oid not_in dept1_del.DOID
```

In general, given 2 versions V_1 with a union set of size N, and V_2 with a union set of size M, a join of V_1 and V_2 results in a cross-product with M * N queries being generated. Thus, in the planner preprocessor, the above union join is split and semantically optimized into the following equivalent set of simple joins:

```
retrieve (emp1_add.name, dept1_add.floor)
  where emp1_add.name = dept1_add.floor

retrieve (emp1_add.name, dept.floor)
  where emp1_add.name = dept.floor
  and dept.oid not_in dept1_del.DOID

retrieve (emp.name,  dept.floor)
  where emp.name = dept.floor
  and emp.oid not_in emp1_del.DOID
  and dept.oid not_in dept1_del.DOID

retrieve ( emp.name, dept1_add.floor)
  where emp.name = dept1_add.floor
  and emp.oid not_in emp1_del.DOID
```

The net result of running the above queries will be a retrieval of all employees in emp1 and the floor they work on.

5 Performance Analysis

This section evaluates the performance of the PVS which has been implemented as described in section 2 with the algorithms and semantic optimizations described in section 3. This performance analysis is aimed at comparing the performance of standard POSTQUEL commands on real relations versus the same ones for versions. In addition, improvements provided by semantic optimization are noted, and the overhead of the rules system (QRS) is discussed. The tests are run on a multi-user DECStation 3100. The base relations, r1k and r5k, are standard relations used in the Wisconsin benchmark, with 1000 and 5000 tuples respectively. In the timings, v1 is a first level version, while v2 is a second level version (i.e, v2 is a version of v1). Both v1 and v2 are versions implemented via forward deltas. The following is the set of queries used in the timings:

1. query (Q1): retrieve (*relname.u1*)
2. query (Q2): replace *relname* (u1 = a.u1) from a in *relname*
3. query (Q3): append *relname* (tup.all)

In the timings, *relname* is replaced by the name of the version being timed, and **tup** is a relation containing either 1000 or 5000 tuples.

Relation	Elapsed time(s)	Semant Opt.
base	0:11	nil
v1	1:08	nil
v1	0:48	P_1 Opt
v1	0:25	$P_1 + P_2$ Opt

Table 1: Running Q2: Time for updates with and without semantic opt.

Table 1 indicates that semantic optimization on average improves the performance of the PVS by approximately 75%. Thus, even with the overhead incurred by doing semantic optimization, the improvement made by semantically optimizing version queries is sizable.

Relation-type	% change	Elapsed time(s)	% degradation
base	-	0:03	-
v1	0%	0:06	100
v1	50%	0:07	133
v1	100%	0:07	133
v2	0%	0:07	133
v2	50%	0:08	166
v2	100%	0:09	200
v3	100%	0:12	300

Table 2: Running Q1: Retrieving 1000 tuples.

Relation-type	% change	Elapsed time(s)	% degradation
base	-	0:18	-
v1	0%	0:21	17
v1	50%	0:25	38
v1	100%	0:36	100
v2	0%	0:21	17
v2	100%	0:41	127

Table 3: Running Q1: Retrieving 5000 tuples.

Relation-type	% of tuples replaced	Elapsed time(s)	% degradation
base	100%	0:11	-
v1	50%	0:18	63
v1	100%	0:25	127
v2	50%	0:23	109
v2	100%	0:31	181
v3	100%	0:38	245

Table 4: Running Q2: Replace on 1000 tuples.

Relation-type	% of tuples replaced	Elapsed time(s)	% degradation
base	50%	0:41	-
base	100%	0:47	14
v1	50%	1:16	62
v1	100%	1:45	123
v2	100%	2:04	164

Table 5: Running Q2: Replace on 5000 tuples.

Relation-type	# of tuples appended	Elapsed time(s)	% degradation
base	1000	0:11	-
base	5000	0:42	-
v1	1000	0:12	9
v1	5000	0:44	4

Table 6: Running Q3: Appends on base relations and versions.

relation-type	operation	#tuples	% degradation
v1	append	5000	12.5
v1	retrieve	10000	62

Table 7: Timings from Woodfill

From the tables, we see that the performance of versions is bounded by a factor of its level from the base. In the case of appends, there is only a minimal overhead for appending to a version versus appending to a base relation. This overhead is the cost associated with the processing of the append rule of the version. A comparison between timings from [WOOD83] shows the the PVS actually performs better for appends to versions. In the case for retrieves, the performance of the PVS is no worst than half that of the implementation by [WOOD83]. Timings from tables 4 and 5 indicate that, in general, updates to versions are very expensive, and is about twice as slow as the implementation by [WOOD83]. This is because every update to

a version is transformed by the QRS into 3 queries: an update to the v_add relation, an append to the v_del relation and an append to the v_add relation. Thus, the timings in tables 4 and 5 reflect the composite times for running the 3 queries. From the timings, the PVS, on average, runs slightly slower than the hard-coded implementation by [WOOD83]. However, the PVS provides a high-level abstraction for managing version semantics that allows the user to easily define the semantics he wants, while the implementation by [WOOD83] offers no such flexibility.

6 Future Extensions and Conclusion

Currently, we only support versions of data; that is, the version must have the same attributes as the base relation. This should be extended so that we can support versions which have different attributes from the base relation.

This thesis has shown that with minimal extensions to POSTGRES, it is possible to design a version system based on production rules. In addition, because of the generality and flexibility of rules, such a system need not be bound to any particular semantics, and is able to subsume most, if not all, of the variations of versioning techniques. Finally, performance analysis of

such a system has shown that it compares with traditional systems.

7 Appendix A

Set of rules that govern the semantics of versions implemented via forward deltas:

APPEND :

```
define rewrite rule v_append
on append to v1 then do instead
append v_add (f1 = NEW.f1, ...)

/* NEW refers to the tuple_values being added to
   the v1  relation } */
```

DELETE :

```
define rewrite rule v_delete
on delete to v1 then  do instead
 {
    delete v_add
           where CURRENT.oid = v_add.oid

    append v_del (DOID = base.OID)
           where CURRENT.oid = base.oid
 }

/* CURRENT refers to the tuples being deleted from v1 */
```

RETRIEVE :

```
define rewrite rule v_retrieve is
on retrieve to v1 do instead
```

```
retrieve (_v1.all)
from b_base in base, _v1 in (v_add | b_base)
where b_base.oid not_in v_del.DOID

/* ''|'' is the symbol for a relational union operator
 * which will be described in section 4.3/
 */
```

REPLACE :

```
define rewrite rule v_replace
on replace to v1 then do instead
 {
    replace v_add  (f1 = NEW.f1, ...)
            where CURRENT.oid = v_add.oid

    append v_del (DOID = base.oid)
           where CURRENT.oid = base.oid

    /*  The next rule is needed to append the
     *  the base tuple to the v_add relation
     *  if this is the first time the base tuple is
     *  updated in the version.
     */

    append v_add (f1 = NEW.f1, ...)
           where CURRENT.oid not_in v_add.oid
           and CURRENT.oid = base.oid
 }
```

References:

[AGRA82] Agrawal,R. and DeWitt, D. J.,
"Updating Hypothetical Data Bases"
Unpublished working paper

[GOH90] Jeffrey K. Goh,
"Rule Processing with Query Rewrite"
Masters Report, EECS Division, UC Berkeley (Dec 1990)

[KATZ82] Katz, R.H. and Lehman, T.J.,
"Storage Structures for Versions and Alternatives"
Computer Sciences Tech Report #479, CSD, U. Wisconsin-Madison
(July 1982)

[KATZ88] Katz R. H.,
"Towards a Unified Framework for Version Modelling,"
EECS Division Tech. Report, UCB/CSD 88/484, U.C. Berkeley
(December 1988)

[SEVE76] Severence, D.G., G.M. Lohman,
"Differential Files: Their Application to the Maintainance of Large Databases,"
ACM Trans. on Database Systems,V1, N.3 (September 1976)

[STON80] Stonebraker, M.R. and Keller, K.
"Embedding Expert Knowledge and Hypothetical Data Bases
into a Data Base System". Proc. ACM SIGMOD Conference,
Santa Monica, California, (May 1980).

[STON81] Stonebraker, M.R., "Hypothetical Databases as Views"
Proc. ACM SIGMOD Conference, Ann Arbor, Michigan, (May 1981)

[STON86] Stonebraker, M., L. Rowe,
"Design of Postgres,"
Proc. ACM-SIGMOD Conference on Management of Data, 1986

[STON90] Stonebraker, M.R., A. Jhingran, J. Goh, S. Potiamanos,
"On Rules, Procedures, Caching, and Views in Database Systems,"
to appear in ACM-SIGMOD Conference on Management of Data, 1990

[WOOD83] Woodfill, J., M. R. Stonebraker,
"An implementation of Hypothetical Relations,"
Computer Science Tech. Report, UCB/ERL M83/2, U.C. Berkeley
(January 1983)