HYPOTHETICAL DATA BASES AS VIEWS

by

Michael Stonebraker
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
UNIVERSITY OF CALIFORNIA
BERKELEY, CA.

ABSTRACT

In this paper we show that hypothetical data bases can be effectively supported by slight extensions to conventional view support mechanisms. Moreover, we argue that the resulting structure may well be quite efficient and that there are advantages to making hypothetical data bases central to the operation of a DBMS.

I INTRODUCTION

In a recent paper [STON80] we discussed the notion of hypothetical data bases (HDB's). These are "what if" data bases which result from a real data base by making some alternate assumption about reality. Such data bases are composed of hypothetical relations and are useful for debugging purposes, test data and a variety of simulations.

Also, in [STON80] we presented an implementation of HDB's using differential files [SEVR76]. In Section 2 of this paper we present a much more elegant implementation of HDB's by slightly extending the view mapping mechanism of [STON75]. Then, in Section 3 we indicate some of the advantages of treating all data bases as hypothetical (including real ones) and discuss the efficiency of the structure which we have proposed. Section 4 closes with some conclusions.

II VIEWS

Views are well known objects in a relational data base setting [CHAM75, STON75, ASTR76, DAYA78] and have been supported at least in INGRES [STON76] and System R [BLAS79]. The INGRES algorithms were presented in [STON75] and effectively map all RETRIEVES and many QUEL update commands on views into appropriate operations on base relations. In this section we extend these algorithms to handle two relational operators, UNION and DIFFERENCE, which are not present in QUEL and are not currently supported in INGRES. Then, we indicate how these operators can be used to support hypothetical relations.

2.1 Extensions to Views

In INGRES a view is defined as follows:

```
RANGE OF R1 IS REL-1
      .
      .
      .
RANGE OF Rn IS REL-n
```

This research was sponsored by the U.S. Air Force Office of Scientific Research Grant 78-3596, the U.S. Army Research Office Grant DAAG29-76-G-0245, and the Naval Electronics Systems Command Contract N00039-78-G-0013.

```
 DEFINE VIEW virtual-rel
     (col-1 = F1(R1,...,Rn),
             .
             .
             .
      col-k = Fk(R1,...,Rn))

   WHERE Q(R1,...,Rn)
```

Here, Q is a valid QUEL qualification and F1,...,Fk are valid target list functions.

For example, if EMP(name, salary, manager, age, dept) is a relation with the obvious semantics, we can define a view as follows:

```
 RANGE OF E IS EMP
 DEFINE YOUNG-EMP(
       name = E.name
       birth-year = 1980 - E.age
       age = E.age
       status = "young")

    WHERE E.age < 30
```

YOUNG-EMP has a row for each employee under 30 containing his name, his age, his year of birth and a constant value for a status field.

Now we extend this view definition mechanism with two additional operators as follows:

```
 RANGE OF R1 IS REL-1
       .
       .
       .
 RANGE OF Rn IS REL-n
 DEFINE VIEW virtual-rel(
      col-1 = F1(R1,...,Rn),
             .
             .
             .

      col-k = Fk(R1,...,Rn))

   WHERE Q(R1,...,Rn)

NEW-OPERATOR

    (col-1 = G1(R1,...,Rn),
            .
```

```
          .
          .

    col-k = Gk(R1,...,Rn))

  WHERE P(R1,...,Rn)
```

Here, G1,...,Gk are legal QUEL target list functions and P is a legal qualification. Moreover, NEW-OPERATOR is UNION or DIFFERENCE. In effect, virtual-rel is the UNION or DIFFERENCE of two union compatible [CODD72] relations.

The algorithms of [STON75] are powerful enough to handle the two individual views above; all we need to do now is to show how to handle both values for NEW-OPERATOR.

In the algorithms which follow there are two semantic difficulties. First, it is impossible to reinsert a deleted tuple or change a tuple back to a previous value. Second, an APPEND statement with no qualification is not correctly processed. These problems are discussed in Section 3.1.

2.2 Union

Consider the relation W = R UNION S expressed as a view as follows:

```
 RANGE OF R1 IS R
 RANGE OF S1 IS S
 DEFINE VIEW W( R.all)

   UNION
           (S.all)
```

We now indicate how to map the various QUEL commands defined on W.

```
RETRIEVE:  RANGE OF W1 IS W
           RETRIEVE (F(W1)) WHERE Q(W1)

becomes:   RANGE OF R1 IS R
           RANGE OF S1 IS S
           RETRIEVE (F(W1)) WHERE Q(W1)
                UNION
           RETRIEVE (F(S1)) WHERE Q(S1)

DELETE:    DELETE W1 WHERE Q(W1)

becomes:   DELETE R1 WHERE Q(R1)
           DELETE S1 WHERE Q(S1)
```

```
APPEND:    APPEND TO W( {col-i = Hi(W1)} )
              WHERE Q(W1)

becomes:   APPEND TO S( {col-i = Hi(R1)} )
              WHERE Q(R1)
          APPEND TO S( {col-i = Hi(S1)} )
              WHERE Q(S1)

REPLACE:   REPLACE W1( {col-i = Hi(W1)} )
              WHERE Q(W1)

becomes:   REPLACE R1( {col-i = Hi(R1)} )
              WHERE Q(R1)
          REPLACE S1( {col-i = Hi(S1)} )
              WHERE Q(S1)
```

2.3 Difference

Consider the relation W = R - S. The four commands from the previous subsection map as follows:

```
RETRIEVE:  RANGE OF W1 IS W
          RETRIEVE (F(W1)) WHERE Q(W1)

becomes:   RANGE OF R1 IS R
          RANGE OF S1 IS S
          RETRIEVE (F(R1)) WHERE Q(R1)
                 -
          RETRIEVE (F(S1)) WHERE Q(S1)

DELETE:    DELETE W1 WHERE Q(W1)

becomes:   APPEND TO S(R1.all) WHERE Q(R1)

APPEND:    APPEND TO W( {col-i = Hi(W1)} )
                WHERE Q(W1)

becomes:   APPEND TO R( {col-i = Hi(R1)})
                WHERE Q(R1)
          APPEND TO S( {col-i = Hi(S1)})
                WHERE Q(S1)

REPLACE:   REPLACE W1( {col-i = Hi(W1)} )
                WHERE Q(W1)

becomes:   APPEND to S(R1.all) WHERE Q(R1)
          APPEND TO R(R1.other, {col-i =
           Hi(R1)}) WHERE Q(R1)
          APPEND TO S(S1.other, {col-i =
           Hi(S1)}) WHERE Q(S1)
```

2.4 Hypothetical Data Bases

Consider the relation, W = (R UNION S) - T. For this composite view the two previous algorithms can be applied in turn to produce the following update rules for W.

```
RETRIEVE:   RANGE OF W1 IS W
            RETRIEVE (F(W1)) WHERE Q(W1)

becomes:    RANGE OF R1 IS R
            RANGE OF S1 IS S
            RANGE OF T1 IS T
            RETRIEVE (F(R1)) WHERE Q(R1)
                  UNION
            RETRIEVE (F(S1)) WHERE Q(S1)
                    -
            RETRIEVE (F(T1)) WHERE Q(T1)

DELETE:     DELETE W1 WHERE Q(W1)

becomes:    APPEND TO T(S1.all) WHERE Q(S1)
            APPEND TO T(R1.all) WHERE Q(R1)

APPEND:     APPEND TO W({col-i = Hi(W1)}
                  WHERE Q(W1)

becomes:    APPEND TO T( {col-i = Hi(T1)})
                  WHERE Q(T1)
            APPEND TO S( {col-i = Hi(S1)})
                  WHERE Q(S1)
            APPEND TO S( {col-i = Hi(R1)})
                  WHERE Q(R1)


REPLACE:    REPLACE W1( {col-i = Hi(W1)}
                  WHERE Q(W1)

becomes:    APPEND TO T(T.other, {col-i =
              Hi(T1)}) WHERE Q(T1)
            APPEND TO T(S1.all) WHERE Q(S1)
            APPEND TO S(S1.other, col-i =
              Hi(S1)}) WHERE Q(S1)
            APPEND TO T(R.all) WHERE Q(R1)
            APPEND TO S(R1.other, {col-i =
              Hi(R1)}) WHERE Q(R1)
```

Intuitively, we can see that R is a read-only portion of W and is not updated by any command. Moreover, S is a differential file containing the additions to R while T is a differential file with the deleted tuples. In the next section we do a simple example to clarify the situation.

2.5 Example

Consider the employee relation, W = W(name, age salary) with component relations R, S and T with the same domain names. Suppose S and T are initially empty and R has the following data:

R	name	age	salary
	Smith	30	1000
	Jones	40	2000

After the command:

```
 RANGE OF W1 IS W
 APPEND TO W(name = "Baker",
             age =  20,
          salary = .8 * W1.salary)
     WHERE W1.name = "Smith"
```

we would find R and T unchanged while S had the following data:

S	name	age	salary
	Baker	20	800

After the update:

```
  REPLACE W1(salary = 1.1*W1.salary)
    WHERE W1.age < 35
```

we would leave R unchanged and update S and T to:

S	name	age	salary
	Baker	20	800
	Baker	20	880
	Smith	20	1100

T	name	age	salary
	Baker	20	800
	Smith	20	1000

III DISCUSSION OF HDB’s AS VIEWS

In this section we indicate some of the considerations that apply to the use of such HDB’s.

3.1 Problem Updates

The algorithms of the previous section successfully map all update operations to hypothetical relations except two. First, it is impossible to insert a previously deleted tuple. Such an operation will be transformed by the above algorithms but will result in no effective change to the data base. The problem with this operation is that the effect of a tuple in T can never be undone because it is applied to the result of (R UNION S). The same issue makes it impossible to update a tuple to have a previous value.

It is possible to successfully cope with this situation but only by performing a DELETE operation to T. The mappings for W = R - S, in the case of APPENDs and DELETEs would require an additional statement. In both cases it is:

```
RANGE OF S2 IS S
DELETE S2 WHERE
  {S2.col-i = Hi(S)}
```

Inclusion of this statement would degrade the performance of hypothetical relations and destroy the append-only nature of differential relations. In Section 3.4 we indicate that append-only differential relations have a strong resemblance to the data base logs used for crash recovery. This connection can be usefully exploited, and we are reluctant to destroy it by updating T.

The second problem update concerns APPENDs with no qualification, i.e.

```
APPEND TO W(name = "Smith", age = 30, salary = 1000)
```

This can be accomplished by:

```
APPEND TO S(list-of-constants)
```

which is faster than the normal algorithm.

3.2 Performance

The algorithms in the preceding section suggest that relations of the form W = (R UNION S) - T may be rather slow to update. This subject is explored further in Table 1. There we indicate four rows, one for each type of QUEL command and in column one we suggest that each be given unit cost (in some arbitrary units).

The second column suggests the cost of these unit commands when applied to a hypothetical relation. For example, note that a DELETE command is turned into two APPEND commands each with the original qualification to be evaluated. If the storage structures of all relations involved are the same, it is safe to assume that the cost of two APPENDS is twice the cost of one DELETE. The rest of the rows in column two are obtained in a similar fashion. A cursory glance at column two suggests that hypothetical relations are 3.25 times as costly as normal relations if all four QUEL operations are equally likely.

	Normal	Hypothetical	Batched	Main Memory	3 plus 4
COMMAND	INGRES	relations	commands	S and T	
RETRIEVE	1	3	3	1	1
DELETE	1	2	2	1	1
APPEND	1	3	3	1	1
REPLACE	1	5	3	2	1

Costs of Hypothetical Relations

Table 1

Column three suggests one possible speed-up technique. In the previous section it can be noted that two of the five commands into which a REPLACE operation is mapped evaluate Q(R1) and two evaluate Q(S1). Then, they do slightly different APPEND's to different relations. If these two pairs of commands are "batched", it should be clear that each pair will cost only slightly more than one of them. Column three results by assuming such "batching" takes place and that the cost of two commands with the identical qualification is indeed equal to the cost of either one alone.

Column four indicates another possible assumption. If S and T are reasonably small, they can be safely assumed to reside in main memory. This might be the case if a utility to merge S and T into R is run periodically, say once per day. Hence, S and T have at most one day's changes. If S and T are in fast memory, column four is obtained by assuming that access to such relations comes at no cost. (This would only be the case if I/O accesses were the major bottleneck). Consequently, all commands access R once except REPLACE's which access it twice.

The final column suggests the case where both "batching" and main memory S and T are present. The bottom line is that the cost of hypothetical relations may not be substantially more expensive than real relations.

3.3 Faster Commands

Some of the commands can be "special cased" in certain circumstances at faster performance. One situation with this property is when operations always specify a unique key

Suppose name in the example from Section 2.5 is a unique key, i.e. no two people have the same name. Moreover, suppose all commands have a qualification of the form "name = constant". In this case, one can access S and T which may be in main memory and consequently inexpensive. The following table indicates the resulting situation concerning R. For example, the first row indicates the case that a particular name is in both S and T, in which case it has assuredly been updated. As such, it exists in R but there is no need to access it because the tuple is identical to the one in T. The remaining rows are arrived at in a similar fashion.

Present in S	Present in T	Present in R
Yes	Yes	Yes, but don't need to access
Yes	No	No

No	Yes	Yes, but don’t need to access
No	No	Yes, and need to access

Access Needed to R

Table 2

Consequently, after the accesses to S and T, one can avoid the access to R in all but the last situation.

Moreover, access to S and T can be accelerated by the addition of a Bloom filter [SEVR76]. One would access such a filter to ensure that the desired tuple was in neither S nor T. If so, one only needs to access R. Alternately, one only needs to access S and T. Consequently, the cost of an access to a hypothetical relation may be near the cost of an access to a non hypothetical relation when a unique key is present and always used in the qualification.

3.4 Crash Recovery

Suppose a unique time stamp (TS) is assigned to each transaction. Moreover, consider widening S and T to have a field for such a time stamp. In addition, suppose all APPENDS to S or T include the time stamp of the transaction making the change. Lastly, a transaction reaching its commit point [GRAY78] would execute the following QUEL command:

```
 APPEND TO DONE-XACT(TS = my_time_stamp)
```

Here DONE-XACTS is a relation containing only time stamps for completed transaction.

Consider recovery from a soft crash, i.e. one for which the data on the disk is intact after the crash. We can simply define the following hypothetical data base and resume normal operation on it.

```
 RANGE OF R1 IS R
 RANGE OF S1 IS S
 RANGE OF T1 IS T
 RANGE OF X IS DONE-XACTS
 DEFINE VIEW W-CRASH (R1.all)
       UNION
  (S1.all) WHERE S1.TS = X.TS

         -
  (T1.all) WHERE T1.TS = X.TS
```

Notice that crash recovery is instantaneous and that all we have to do is switch users of W to W-CRASH. At some later time S and T can be purged of offending tuples by a background task which runs the following update:

```
 DELETE T1 WHERE COUNT
 (T1.TS WHERE T1.TS = X.TS) = 0

 DELETE S1 WHERE COUNT
 (S1.TS WHERE S1.TS = X.TS) = 0
```

Note that this update is somewhat syntactically unappealing because QUEL does not contain a "there does not exist" operator.

Another point to be noted is that conventional DBMSs recover from soft crashes either by writing a log [GRAY78] or by using a deferred update mechanism [STON76]. Both tactics amount to less structured versions of S, T and DONE-XACTS stored as normal files. Hence, overhead must be included to write these extra structures anyway; the extra overhead paid by hypothetical relations is that these extra objects are structured as relations (which are slower than normal files) and qualifications must be evaluated on them.

Also, if the log is stored as relations, it can be accessed by normal users via the query language, in stark contrast to current logs. Notice finally that the code to implement soft crash recovery is very simple, a feature not found in current recovery systems.

Recovery from hard crashes, i.e. ones for which data on the disk may be lost, amounts to periodically dumping S, T and DONE-XACTS to an alternate medium and occasionally dumping R.

3.5 Clean Up of S and T

It should be noted that a tuple which is updated multiple times will appear several times in S and T. Hence, it will enlarge the collection of tuples examined by subsequent commands and presumably slow down the execution of them. However, a background task can run the following update to clean up S and T:

```
DELETE S1 WHERE S1.all = T1.all
DELETE T1 WHERE T1.all = S1.all
```

Although it is obvious the effect which is intended, the second DELETE will never do anything because the matching tuple in S has already disappeared.

Moreover, it is equally unworkable to define a view W = S UNION T and then issue the command:

```
RANGE OF W1 IS W
DELETE W1 WHERE S1.all = T1.all
```

This will be expanded into the two incorrect DELETES above.

To obtain the correct effect we would have to extend QUEL to have the notion of a TRACK VARIABLE as follows:

```
RANGE OF TRACK-VAR IS TRACK (S,T)
DELETE TRACK-VAR WHERE S1.all = T1.all
```

The semantics of track variables are that they can appear in DELETE and REPLACE statements in place of the normal tuple variable indicating the relation to be affected. Track variables cannot appear elsewhere in a command. One implementation of track variables would be to map the command as if it were defined on a view consisting of the union of all tracked relations and in addition to put all the resulting updates inside a (begin-transaction, end-transaction) pair. Such a transaction must also have the property that updates inside the transaction CANNOT be visible to the transaction making the changes. This is in contrast to the normal mechanism where the opposite assumption is made.

3.6 Incremental Reorganization

It appears not uncommon to have relations in real applications which are so large that physically reorganizing them is extremely costly. Such reorganizations occur when the access structure for a relation is changed, i.e. from indexed to hashed or when a relation is rehashed to achieve better storage utilization. Moreover, it is possible that the cost is so large that the relation cannot be quiesced for an appropriate period of time. In such cases, incremental reorganization becomes attractive. In fact B-trees have exploited this situation fully [BAYE70].

However, HDB's are easily reorganized as follows:

```
RANGE OF R1 IS R
RANGE OF S1 IS S
RANGE OF T1 IS T
APPEND TO R(S1.all) WHERE Q(S1)
DELETE S1 WHERE Q(S1)
DELETE R1 WHERE
   R1.all = T1.all AND Q(T1)
DELETE T1 WHERE Q(T1)
```

An appropriate choice of Q will allow any desired portion of the two differential files to be merged into R. Moreover, with suitable locking this merging can be done as a background task. Hence, an appropriate Q and a suitable interval between running the above transaction will yield any desired reorganization characteristics.

3.7 Update Rules

Consider three hypothetical relations:

```
W = (R UNION S) - T

X = (R UNION S') - T'

Y = (W UNION S'') - T''
```

Conceptually, suppose W is the "real" relation, implemented as a hypothetical relation consisting of R, S and T. Hence, all "real" updates are directed to W and mapped according to the algorithms of Section 2.4. Now X is a hypothetical relation based on R. Here, X is a "what if" relation based on a "snapshot" of the real relation W. This snapshot is as of the time when S and T were last merged into R. Normal updates applied to W do not affect X and vica-versa. The result is a hypothetical relation with an alternate assumption about the state of W at a fixed point in the past.

Let us now turn to Y which is a hypothetical relation defined on top of W. It can be updated according to the normal rules and its changes will not be reflected into W. Hence, it is a hypothetical relation based on the present state of W. However, what if W is updated? We consider the following illustrative sequence:

Initially Smith earns $1000 and appears in R. Next we run the following update

```
RANGE OF Y1 IS Y
REPLACE Y1(salary = 1.1*Y1.salary)
       WHERE Y1.name = "Smith"
```

The effect will be to add (Smith, 1000) to T'' and (Smith, 1100) to S''. Next, Smith is given a 20 percent raise in W, i.e.

```
RANGE OF W1 IS W
REPLACE W1(salary = 1.2*W1.salary)
  WHERE W1.name = "Smith"
```

As a result (Smith, 1000) is added to T and (Smith, 1200) to S. The last command is to retrieve Smith's salary, i.e:

```
RETRIEVE (Y1.salary)
   WHERE Y1.name = "Smith"
```

We will obtain two answers, 1100 and 1200, which is obviously not the semantically desired result.

The curious result which we are left with is summarized in Table 3.

	Normal views	Hypothetical relations
updates on views	o.k. with certain restrictions	o.k. always

views on top of views	o.k.	o.k.
updates to views on top of views	o.k. if composite mapping invertible	o.k.
updates to views on top of which there are other views	o.k. if view is updatable	not always o.k.

Updates to Views

Table 3

Normal views (as in [STON75]) can be updated if the mapping describing the view is invertible. As pointed out in [DAYA78] this is not always the case. On the other hand, hypothetical relations can always be updated evn though the mapping described in Section 2.3 is not invertible.

Next, both types of views allow cascading of views on top of views and allow such cascaded views to be updated. However, updates to views on top of which there are other views behave differently in the two situations. Normal views can be updated without concern for whether there are cascaded views on top of them. Such updates are automatically reflected correctly into cascaded views. On the other hand, hypothetical relations do not have this freedom. Cascaded views will be invalidated if a tuple is updated that the cascaded view has modified or deleted or if a tuple is deleted which the cascaded view has modified.

The fundamental problem is that the update rules for (R UNION S) - T depend for their correct operation on R being read-only. In the case of Y = (W UNION S’’) - T’’, W is NOT read-only and correct operation is not assured. The update rules for normal views do not have any requirements that an underlying relation be read-only.

IV CONCLUSIONS

We have examined views of the form (R UNION S) - T and indicated algorithms to support them. They appear to be appropriate for hypothetical data bases, for assistance in crash recovery, for efficiently providing snapshots and for generating an automatic audit trail. Certainly, there may be efficiency questions; however, it is suspected that a novel DBMS organization may be able to help overcome them.

REFERENCES

[ASTR76]	Astrahan, M. et. al., "System R: A Relational Approach to Data," TODS, 1, 2, June 1976.
[BLAS79]	Blasgen, M. et. al., "System R: An Architectural Update," IBM Research, San Jose, Ca., Report RJ2581, July 1979.
[CHAM75]	Chamberlin, D. et. al., "Views, Authorization and Locking in a Relational Data Base management System," Proc. 1975 National Computer Conference, Anaheim, Ca., June 1975.
[CODD72]	Codd, E., "Relational Completeness of Data Sublanguages," Courant Computer Science Symposium, 1972.
[DAYA78]	Dayal, U. and Bernstein, P., "On the Updatability of Relational Views," Aiken Computer laboratory, Harvard University, March 1978.
[GRAY78]	Gray, J., "Notes on Operating Systems," IBM Research, San Jose, Ca., Report RJ 3120, October 1978.
[SEVR76]	Severance, D. and Lohman, G., "Differential Files: Their Application to the Maintenance of Large Databases," TODS, June 1976.
[STON75]	Stonebraker, M., "Implementation of Integrity Constraints and Views by Query Modification," Proc. 1975 ACM-SIGMOD Conference on Management of Data, San Jose, Ca., June 1975.
[STON76]	Stonebraker, M. et. al., "The Design and Implementation of INGRES," TODS 2, 3, September 1976.
[STON80]	Stonebraker, M. and Keller, K., "Embedding Expert Knowledge and Hypothetical Data Bases Into a Data Base System," Proc. 1980 ACM-SIGMOD Conference on

Management of Data, Santa Monica, Ca., May 1980. September 1980.