AN IMPLEMENTATION OF HYPOTHETICAL RELATIONS

John Woodfill and Michael Stonebraker

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

ABSTRACT

In this paper we present a new approach to implementing hypothetical relations. Our design borrows ideas from techniques used in processing views and differential files and offers several advantages over other schemes. A working implementation is described and performance statistics are presented.

1. INTRODUCTION

Hypothetical relations [STONB0, STONB1, AGRA92] have been suggested as a mechanism to allow users to generate alternate versions of real relations. Each version can be updated as if it were a real relation; however, only differences between the hypothetical relation (HR) and the real relation on which it is defined are actually stored. Previous papers have concentrated on data structures for representing these differences and algorithms for processing data manipulation commands addressed to HR's.

HR's correspond closely to the notion of versions [BONA77] used in systems which manage iterations of computer programs. On the other hand, HR's differ fundamentally from views in that updates to HR's should not cause changes to the relation on which they are defined. An update to a view is reflected through to the base relation(s) underneath it.

Consider, for example, a university financial officer who is in charge of research assistant's salaries. Suppose she is trying to balance her budget and wants to know whether her accounts would balance under the hypothetical scenario that salaries of senior employees were cut by 20%. She could make a new copy of her database, actually perform the appropriate updates, and then survey the results. This procedure would be slow and require considerable disk space. Alternatively she could define an HR on the employee relation, perform her updates on the HR, and then survey the results.

Another use of database HR's might be in debugging a database application program. The programmer might not want to test his program on "live" data because a logical error could corrupt the database. He could define an HR on the live data and test the program on this HR.

The implementation suggested in [STONB0] involves a single differential file [SEVR76]. A more elegant solution [STONB1] suggests supporting HR's by using the view mechanism [STON75] already present in many relational systems. A hypothetical relation, W, for a real relation R would be defined as a view of the form W = (R UNION S) DIFFERENCE T. To support this implementation of HR's, one need only extend a relational DBMS and its associated view mechanism with the UNION and DIFFERENCE operators. A possible advantage of this implementation is that R can be a read-only relation while S and T are append-only. This leads to the possibility of implementation on an optical disk.

Unfortunately, there are problems with defining HR's as views. We first examine these problems and show general solutions in Section 2. Then in Section 3 we combine these solutions into a new mechanism for supporting HR's. Our implementation of this solution is described in Section 4. Finally, we present performance statistics from our running prototype in Section 5.

2. PROBLEMS AND SOLUTIONS

Proposals for implementing hypothetical relations as views contain various flaws which must be removed before a realistic implementation can be attempted.

2.1. The Re-insertion Problem

[STONB1] points out that the implementation of hypothetical relations as W = (R UNION S) DIFFERENCE T is flawed in the case where one wants to re-insert a tuple which has been deleted, as shown by the example in Figure 1. Initially there is a tuple in relation R corresponding to Eric. Following the algorithm in [STONB1], the tuple can be deleted by inserting it into relation T. Lastly a user re-embeds Eric and an appropriate tuple is inserted into S. Unfortunately, the resulting hypothetical relation, W, does not contain the re-inserted tuple since (R UNION S) is the same as R, and R DIFFERENCE T is empty.

<table>
<thead>
<tr>
<th>R name</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>eric</td>
<td>10000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S name</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>eric</td>
<td>10000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T name</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>eric</td>
<td>10000</td>
</tr>
</tbody>
</table>

Figure 1.

This research was supported by the Advanced Research Project Agency under contract #N00039-C-0235.
2.2. A Solution Using Timestamps

This problem can be solved by adding a timestamp field to the relations S and T, and modifying the semantics of the DIFFERENCE operator. Tuples in R do not require a timestamp field and can be thought of as having a timestamp of zero. [AGRA82] also proposes a timestamp solution to this problem.

The timestamp field is set to the current time (from a system clock, or any other monotonically increasing source of timestamps) whenever a tuple is appended to S or T. For any relations A and B with timestamps as described, A DIFFERENCE B is defined as all tuples a in A for which there is no tuple b in B such that:

1. DATA(a) = DATA(b)

and

2. TIMESTAMP(a) < TIMESTAMP(b)

The definition of R UNION S is unchanged, except that a timestamp field must be added to the result, which contains either the timestamp of a tuple in S, or a zero timestamp for a tuple in R. If tuples with identical DATA appear in both R and S, the newer timestamp (from S) is chosen for the result tuple.

In the above example, the timestamp of Eric’s tuple in T would be newer than that of Eric’s tuple in R (zero), but would be older than the timestamp of Eric’s tuple in S; hence, (R UNION S) DIFFERENCE T would be equivalent to S, and W would contain the re-appended tuple.

[KATZ82] suggests solving the problem of re-appended tuples by adding a unique identifier (termed a Surrogate) to each tuple. Thus if a tuple is deleted from an HR, the appended tuple in T has the same Surrogate as the tuple to be deleted. If a tuple with the same DATA is subsequently appended, it will have a new Surrogate, and hence be distinct. Neither [AGRA82] nor [KATZ82] deals with the multi-level HR’s to be discussed next.

2.3. The Multi-level Problem

The addition of timestamps solves the re-insertion problem. However, this solution does not work for multi-level HR’s. Multi-level HR’s would be useful in many applications where several people are working on a project. One designer might want to safely test his own embellishments to another’s proposed modifications. Consider the case of a second level hypothetical relation, W = (W UNION S’) DIFFERENCE T’, as shown in Figure 2. Suppose Eric has been given a 20 percent raise in W at timestamp 20, which results in the entries for S and T shown in Figure 3. According to the algorithm above, W’ would contain two tuples for Eric, one with salary 15,000, and one with salary 12,000. The problem is that the tuple in T’ no longer functions to exclude Eric from W UNION S’ and hence an unwanted Eric tuple is present.

3. According to the algorithm above, W’ would contain two tuples for Eric, one with salary 15,000, and one with salary 12,000. The problem is that the tuple in T’ no longer functions to exclude Eric from W UNION S’ and hence an unwanted Eric tuple is present.

There are at least two choices for the proper semantics for W’ under this update pattern:

1. Eric’s salary is set to the latest value, in this case the 15,000 from W.
2. Eric’s salary is set to 12,000, corresponding to the original update of W.

We make the latter choice, and specify the following semantics:

Once a tuple has been changed at level N, changes to this tuple at levels < N cannot affect tuples at levels > N.

As a result further modifications to the HR algorithms are required.

2.4. A Solution With Identifiers

These semantics can be guaranteed by the addition of a tuple identifier, and modification of the DIFFERENCE operator. A tuple identifier, TNAMES, must be added to each tuple in R. Any inserts to S or T, which are used to replace or delete a tuple in W, must be marked with the identifier for the original tuple in R or S which they replace or delete. Each tuple inserted into W (and thereby added to S) must be given a new identifier. For any relations A and B with timestamps and TNAMES as described, A DIFFERENCE B is defined to be all tuples a in A for which there is no tuple b in B such that:

1. TNAMES(a) = TNAMES(b)

and

2. TIMESTAMP(a) < TIMESTAMP(b)

To guarantee that our chosen update semantics hold, tuples in A DIFFERENCE B must be given timestamps of zero. Hence, at a second level, each tuple in S’ and T’ will have a newer timestamp than its corresponding tuple in W.

In our example the identifier of all of the five Eric tuples from Figure 3 will be identical. Since the timestamp of the tuple in W is treated as being older than that of the tuple in T’, only Eric’s tuple from S’ will be contained in W.

Timestamps alone have been shown to be insufficient for solving the problems of multi-level HR’s. Identifiers alone are also insufficient, since in multi-level HR’s, identifiers must remain constant in REPLACE3 and hence multiple tuples with the same identifier must be distinguishable (timestamps can distinguish the multiple tuples).
3. A MECHANISM

Given these modifications to the composition of S, T and the meaning of DIFFERENCE, an HR of the form W = (R UNION S) DIFFERENCE T no longer has its original conceptual simplicity. Moreover, support for HR’s becomes considerably more complex than simply implementing UNION and DIFFERENCE as valid operators in a relational system. Consequently, we have designed a mechanism based on differential file techniques. The goal is to provide a single-pass algorithm with proper semantics that will support arbitrary cascading of HR’s. The next two sections describe our data structure and algorithm in detail.

3.1. The Differential Relation

Each hypothetical relation W, built on top of a real or hypothetical relation B, has S and T merged into an associated differential file D, which contains all columns from B plus plus five additional fields. For example, the differential relation D for the base relation R from Section 2 is shown in Figure 4. Name and salary are the attributes from R. The fields min-

name | salary |
-------|--------|
fred | 4000 |
sally | 6000 |

Figure 4.

Figure 4, attributes of the differential relation date and maxdate are both timestamps. mindate is the timestamp as defined in Section 2, while maxdate is another timestamp to be explained in Section 4.2. The fields level and tupnum are used to identify the tuple which this tuple replaces or causes the deletion of. Each hypothetical relation is assigned a level number as indicated in Figure 5. All real relations are at level zero, and an HR built from a real relation is assigned a level of one. Then an HR built on top of level 3

level 2

level 1

level 0

Figure 5.

a level one HR is given a level of two. Hence the column level identifies the level number of a particular tuple, while the column tupnum is a unique identifier at a particular level. Together tupnum and level comprise the unique identifier, TNAME, of a tuple. Values for tupnum are sequentially allocated integers. The last field in D, type, marks what form of update the tuple represents; thus, it has three values, APPEND, REPLACE, and DELETE.

The following examples will illustrate the use of these extra fields. A precise algorithm is presented in Section 3.2. Suppose the relation R has the data shown in Figure 6.

range of w is W

replace w (salary = 8000) where w.name = "sally"

After this update, D looks like the table in Figure 8. mindate is 40, the current timestamp. The tuple which we are replacing in R has an identifier of ( level = 0, tupnum = 1) (see Figure 6).

The resulting form of D is Figure 9. Since this operation is a delete and name and salary are no longer important, they are set to null. Tupnum and level are the same as in Figure 8, since they refer to the same tuple.

Suppose we now replace the tuple appended above; eg:

<table>
<thead>
<tr>
<th>name</th>
<th>salary</th>
<th>tupnum of this tuple is 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>salary</td>
<td>tupnum of this tuple is 1</td>
</tr>
</tbody>
</table>

Figure 6.

Figure 7.

Figure 8.

Figure 9.
replace w (name = "billy") where w.name = "nancy"

Figure 10 shows D in its final form. T tupnum and level
identify the original "nancy" tuple (see Figure 7 above). At this point, R is unchanged, and W looks like:

<table>
<thead>
<tr>
<th>name</th>
<th>salary</th>
<th>mindate</th>
<th>maxdate</th>
<th>level</th>
<th>tupnum</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>nancy</td>
<td>5000</td>
<td>30</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>APPEND</td>
</tr>
<tr>
<td>sally</td>
<td>8000</td>
<td>40</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>REPLACE</td>
</tr>
<tr>
<td>billy</td>
<td>5000</td>
<td>60</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>REPLACE</td>
</tr>
</tbody>
</table>

Figure 10.

3.2. The Algorithm

There are two parts to the algorithm for supporting hypothetical relations: accessing an HR, and updating an HR.

3.2.1. Accessing Hypothetical Relations

The algorithm for deriving a level N hypothetical relation W from a base relation R and a collection of differential relations D1, ..., DN is a one pass algorithm which starts with the highest level differential relation and proceeds by examining all tuples at each level, passing through lower levels, and finally scanning through the level 0 base relation. Figure 11 shows this processing order more clearly. MaxLevel is the level N of the relation W.

For physlevel := MaxLevel DOWN TO 0 DO
BEGIN
WHILE (there are tuples at level physlevel) DO
BEGIN
    tuple := get-next-tuple(physlevel);
    examine-and-process-tuple(tuple, physlevel);
END
END.

Figure 11. HR processing order.

<table>
<thead>
<tr>
<th>Row</th>
<th>level0</th>
<th>newest</th>
<th>seen</th>
<th>type</th>
<th>samelevel</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>accept</td>
</tr>
<tr>
<td>2</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>3</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>4</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>5</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>6</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>7</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>8</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>9</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Table 1. Processing criteria for HR's.
Figure 13. Initial structures for processing $W$.

3.2.2. Updating Hypothetical Relations

All updates to an HR of level $N$ require appending tuples to the differential relation $D_N$ at level $N$. The contents of the different fields in the appended tuple are specified as follows:

(A) For APPENDS and REPLACES, the data columns of $D_N$ are filled with new data. For DELETES, the data fields are empty.

(B) $\text{mindate}$ is assigned the current timestamp. ($\text{mindate}$ is discussed in Section 4.2.)

(C) For APPENDS, $\text{tupnum}$ and $\text{level}$ are set to self-identify the inserted tuple. For DELETES and REPLACES $\text{tupnum}$ and $\text{level}$ identify the tuple which we will call the "Affected Tuple". The Affected Tuple is the tuple in $R$ or in one of the differential relations $D_M$ ($M \leq N$) which is being deleted or replaced.

(D) $\text{type}$ is the type of the update, APPEND, DELETE or REPLACE.

4. IMPLEMENTATION

An implementation of HR's was done within the INGRES DBMS [STONY76]. This implementation took approximately 2 man months and 2000 lines of code. In order to create an HR, the following addition to QUEL was made:

**DEFINE HYPREL newHR ON baserel**

Once an HR has been defined, it can be updated and accessed just like an ordinary relation. Since, $baserel$ can be either a regular relation, or an HR, a practically unlimited number of levels is allowed.

4.1. DBMS Modifications

Within the INGRES access methods, a relation is accessed first by a call to "find" which sets the range for a scan of tuples, and then "get" is called repeatedly to access each tuple in this range. It is within "get" that most of the HR algorithm is implemented. "Get" returns accepted tuples from each differential relation, and finally the accepted tuples from the base relation. The routines which perform REPLACES, DELETES, and APPENDS have also been modified to initialize and append the appropriate tuples to the differential relation.
4.2. Implementation of Newest

If tuples were appended to a differential relation and the relation were scanned in the same direction, it would be possible to tell when a tuple was the newest for a particular identifier by the fact that it was the first one encountered. Unfortunately, the INGRES access methods append tuples at the end of a relation and scan relations from the beginning. In order to be able to tell from a single pass whether a tuple is newest, an additional timestamp field maxdate was added. When a tuple is appended, maxdate is set to infinity. When the tuple is REPLACED or DELETED at the same level, maxdate in the Affected Tuple is updated. Thus a tuple is the newest if the time of the current scan is between mindate and maxdate. If access methods which appended and scanned starting at the same end of a relation were implemented, the field maxdate could be eliminated and the differential relation could be append-only.

4.3. Implementation of Seen-ids

The data structure, seen-ids, can be stored either in a series of main memory bitmaps, one for each level, or as a hashed table. For small relations or ones with many changes, the bitmap representation makes sense. Thus to see a tuple with tupnum Y at level L, bit Y in bitmap L is set. The boolean function "seen(L, Y)" tests whether the corresponding bit is set. For large relations, with few changes a hash table is more efficient.

4.4. Performance Enhancement

If the base relation is organized as either a random hash structure or an ISAM structure, the differential relations can be given a similar structure and a sequential scan of the differential relation can be avoided. To accomplish this, a correspondence must be established between the pages in a differential relation and those in the base relation. If a tuple would be placed on a certain page of the base relation, then the tuple in the hypothetical relation must be placed on the corresponding page in the differential relation.

If the base relation is structured so that the number of pages which must be scanned can be restricted for a particular query, then only the corresponding section of the structured differential relation need be scanned. For example, suppose the relation R(name, salary) is stored hashed on name and the differential relation D is stored likewise.

Then, the query

range of w is W
retrieve (w.all) where w.name = "billy"

only requires accessing the hash buckets corresponding to the hash value of "billy" in both R and D.

There is one complication with this performance enhancement, which stems from the fact that a replace can create a new tuple which differs from the Affected Tuple in the value of the column (or set of columns) which determine where a tuple is located. We will call this column (or set of columns) the Access Key of the relation. If the new version of a tuple has a different Access Key than the Affected Tuple it may go on a different page than it would if it had the same Access Key. This possible movement of the newest version of a tuple could have dire consequences. For example, consider the contents of R and D shown in Figure 14. Then suppose we do the following REPLACE:

range of w is W
replace w (name = "kelly") where w.name = "suzy"

As a result, R and D would look like Figure 15.

The query:

retrieve (w.all) where w.name = "suzy"

would generate the result:

<table>
<thead>
<tr>
<th>name</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>suzy</td>
<td>3000</td>
</tr>
</tbody>
</table>

Despite the fact that we changed Suzy's name, she appears in the result because the algorithm indicates searching the hashbucket corresponding to "suzy" hashbucket 1 of D, where there are no tuples, then searching hashbucket 1 of R, where Suzy's tuple
Figure 16 appears. This tuple in hashbucket 1 of R is accepted, because no tuples have been seen. Unfortunately, the algorithm never searches hashbucket 0 of D to discover the replacement tuple.

This problem can be solved by the addition of a fourth type of differential tuple, FORWARD. A FORWARD tuple is inserted in hashed and ISAM differential relations in the location indicated by the Access Key of the Affected Tuple whenever a REPLACE is done which changes the Access Key so that the new REPLACE tuple will go in a different hashbucket (or ISAM data page) than that of the Affected Tuple. With this correction, D of Figure 15 would look like Figure 16. The processing of the query would then start in hashbucket 1 of D in Figure 16, where a FORWARD tuple would be found, and the ordered pair <0, 0> would be added to seen-ids. Next, hashbucket 1 of R would be scanned, but since <0, 0> is in seen-ids, Suzy's tuple, tuple 0 of R, would not be accepted.

4.5. Functionality
With these enhancements, all QUEL commands have been made operational on HR's. Moreover, a type of "snap-shot" of the state of an HR at any point in the past can be accessed by setting the scan time (see Section 4.2) to a time prior to the current time. In this way the algorithm is run with some earlier timestamp than the usual current one.

Because differential relations are stored as standard relations, they can be restructured and manipulated using the full power of QUEL. Hence, a user could easily purge the differential relation of records which became invalid as of a certain date using the following QUEL command:

```
range of d is D
delete d where d.maxdate < "a given date"
```

Alternatively, if a user wanted to merge the HR back into the base relation, he can use a series of QUEL statements to update the base relation using the information in the differential relations. A simple utility could also be constructed to perform the same function.

4.6. Space Utilization
The secondary storage requirement for an unstructured HR is one differential tuple per updated tuple. For a structured relation, there is a significant initial overhead for the hashed or ISAM differential relation. This cost can be minimized by constructing the differential relation so that one of its pages corresponds to several in the base relation. If it is expected that 25% of the tuples in the HR will be updated, one page in the differential relation might correspond to four pages in the base relation.

The only primary storage requirement is temporary space for bitmaps or hash tables for the duration of a scan. Bitmaps, require roughly one bit per unique identifier. For example a million record relation would require a corresponding bitmap of 1 million bits, i.e. 125k bytes. Hash tables require more bits per identifier, but should have far fewer entries, and hence should be fairly small.

5. PERFORMANCE MEASUREMENT
Our performance study is aimed at comparing the performance of QUEL commands on standard relations versus the same commands on HR's. The tests were run on a single user VAX 11/780. The three benchmarks in Figure 17 are used to measure update performance for a relation parts5000(pnum, pname, pweight, pcolor) of 5000 tuples stored as a heap. Moreparts5000 is a source of 5000 additional parts tuples. At the beginning of each benchmark parts5000 is returned to its initial state. Table 2 indicates the results of running benchmarks (a) - (c) first with parts5000 as a real relation stored as a heap and then for parts5000 as an HR stored as a heap. In the latter case parts5000 consists of an empty differential relation, D and a 5000 tuple real relation, R stored as a heap. Notice that real and hypothetical relations perform comparably. Table 3 shows the same update tests run against parts5000 hashed on pnum. Benchmark (d) (shown in Figure 18) is added in order to test HR performance when Access Keys are changed and FORWARD tuples are inserted in the differential relation (see Section 4.4).
To test retrieval performance, we also ran benchmark (e) (shown in Figure 19) for 10000 tuple real relation and a 10000 tuple HR. The hypothetical relations had sizes of differential relations, D, varying from 0 to 100% of the size of the R. (If every tuple has been replaced once in an HR, D is 100% of the size of R.) Figure 20 shows the results of these tests.

Benchmark (e) was also run against a second level HR based on a first level HR with 50% of its tuples replaced. The results of this test are in Figure 21.
Lastly, we ran benchmark (f) (shown in Figure 22) against a parts relation hashed on pnum. Table 4 compares performance where parts5000 is either a 5000 tuple real relation hashed on pnum, or a 5000 tuple HR hashed on pnum. with 50% of its tuples replaced. Moreparts5000 is an unstructured 5000 tuple relation.

<table>
<thead>
<tr>
<th>type</th>
<th>cpu time secs</th>
<th>elapsed time minutes</th>
</tr>
</thead>
<tbody>
<tr>
<td>real relation</td>
<td>131</td>
<td>6.86</td>
</tr>
<tr>
<td>HR</td>
<td>165</td>
<td>9.88</td>
</tr>
</tbody>
</table>

Table 4, hashed access performance.

We can see that the performance of INGRES using hypothetical relations is comparable to its performance on real relations for a variety of commands including restrictions, aggregates, and joins. In some cases HR's cause additional overhead, however this penalty is usually small. Only in extreme cases is it more than a factor of the HR level number.

6. CONCLUSIONS

We have described a mechanism for supporting HR's which is shown to overcome the problems of previous proposals. We have described an implementation of this mechanism and provided data to show that performance of HR's is surprisingly good and has modest space requirements. Moreover using our HR mechanisms, it is possible to make inquiries about HR's as of a particular time in the past. Hence, an additional benefit is an efficient implementation of "snapshots".

Benchmark (f)

range of m is moreparts5000
range of p is parts5000

retrieve (m.weight, p.weight) where m.pnum = p.pnum

Figure 22, hashed retrieval benchmark.
REFERENCES


