DBMS TRANSACTION TRANSLATION

Yannis Vassiliou

and

F.H. Lochovsky

Computer Applications and Information Systems Area
Graduate School of Business Administration
New York University

Working Paper Series

CRIS #17

GBA #81-09 (CR)

Published in COMPSAC 80 Conference Proceedings, Chicago, October 1980.

DBMS TRANSACTION TRANSLATION

Yannis Vassiliou
Graduate School of Business Administration
New York University

F. H. Lochovsky
Computer Systems Research Group
University of Toronto

Abstract

Data translation and transaction translation are two major problems that have to be solved in order to achieve the coexistence of heterogeneous distributed databases. In this paper we discuss the problem of transaction translation. The nature of the problem is explored by developing direct translations of transactions between the relational, and hierarchical and network models. Methods for mapping a hierarchical or network schema to an equivalent relational schema are presented. The relational operators projection, selection, join, insertion, deletion and update are translated to equivalent hierarchical and network operations.

1. Introduction

Centralized systems have largely dominated the computer field for a number of years. While centralized systems are the appropriate choice for many organizations, there are some organizations such as banks, governments and large corporations that may find decentralized systems more suited to their processing requirements. Technological advances such as better network communications and cheaper processing power may also point to a distributed system solution to data management and data processing problems. However, even though the data and processing are distributed, we may still want to view the system as a whole [Deppe 1976, Rothnie 1977b, Adiba 1978]. There are several ways in which the data can be distributed in such a system.

One approach assumes that there exists a global database which is distributed in a controlled fashion to the different nodes of a network [Rothnie 1977a, Rothnie 1980]. In this case there is one data model and one data language which is used at every node. Access to any data in the system is done in a uniform way. This situation assumes that it is feasible to convert all the data of an organization to the DBMS of the distributed system or that this situation already exists. There may be situations where such a scenario is not realistic.

In any large organization that has existed for some time, the data may be distributed over many departments and perhaps even over several geographic locations. The different departments or locations may have computerized on an individual basis, choosing different DBMSs that met their particular needs. The trend in information systems is increasingly towards providing an integrated view of an organization's data. As these organizations move to integrate their data, they are faced with the need for communication and cooperation between heterogeneous, distributed databases [Adiba 1977].

Data translation and transaction translation are two major problems that have to be solved in order to achieve the coexistence of heterogeneous databases. For an on-line, fast response environment, data translation *per se* does not provide a complete solution. The restructuring and transmission of large amounts of data may not be economical. A more promising approach may be to translate transactions from one DBMS to another, and to process the transactions in the environment of the DBMSs where the data reside. In this way, only transactions and the retrieved data need to be translated and shipped over the network, not the entire databases, and database translation and restructuring is minimized or avoided entirely.

As a first step in the investigation of this approach, we assume the case where the language of database integration is relational-like. That is, whenever we want to perform global operations in a heterogeneous, distributed system, these operations are expressed in a relational-like language. We further assume that the databases in the system are structured according to only the hierarchical, network or relational data models. In this case, it is necessary to translate hierarchical and network schemas to relational schemas and transactions from the relational-like language to equivalent hierarchical or network operations. By specifying generic algorithms for these translations we will illustrate the nature of the transaction translation problem for heterogeneous, distributed databases. Not all algorithms are presented in this paper because of space limitations. For an extended version of this paper the reader is referred to [Lochovsky 1979].

2. Framework

We consider a database as consisting of a schema, a set of states and a set of operations [Klug 1978]. The schema has generally two parts, a naming part (structures) and a constraint part. The constraints (properties of the database) may be classified as *explicit* (declared in the schema), *inherent* (expressed by the structure) or *implicit* (implied by other existing constraints) [Brodie 1978]. A database state is determined by the values of a set of objects. For instance, a hierarchical database state is determined by the database proper, the position pointer, etc.

We say that a database is *schema-equivalent* to another database if there exists a mapping that maps the schema S_2 of the second database to the schema S_1 of the first database such that all constraints in S_2, that are essential in the context of the first database, can be preserved in S_1. There may be some properties in the schema of S_2 that are immaterial in the context

of S_1. For instance, the set ordering for a network schema is immaterial in the relational model. In this respect, the schema mapping is not reversible. We say that a database is *operation-equivalent* to another database if it is schema-equivalent, and every operation on the first database can be mapped to (a series of) operations on the second database such that the consistency of the database states is preserved. We show in this paper how, for any hierarchical or network database, we can generate an operation-equivalent relational database. First we show the schema equivalence, and then we map the basic relational operations (projection, selection, join, insertion, deletion and update) to hierarchical or network operations.

We define the network and hierarchical schemas by simple inductive rules applied to the schema structures. The structures in a hierarchical schema are segment types and hierarchical links (parent-child relationships) [IBM 1975]. The structures in a network schema are record types and network links (set types) [CODASYL 1971].

To map network and hierarchical schemas to relational schemas we need to specify constraints in the relational schema [Zaniolo 1978, 1979]. Constraints are specified procedurally. A basic constraint that will be used extensively in the mappings is the *foreign-key dependency* constraint or simply *foreign-key* constraint. Our notation for this constraint is:

FC_{ij} := *value* $FK(R_i)$ *in* R_j *dependent on value* $K(R_i)$

The term $FK(R_i)$ means that the key of R_i is a foreign key in R_j. The term $K(R_i)$ refers to the key of R_i. The consequences of this constraint are:

1.- at all times, *values* $FK(R_i)$ *in* R_j $\subseteq$*values* $K(R_i)$;

2.- if we insert an R_j tuple, then the R_i tuple with key value $FK(R_i)$ in R_j must exist or the insertion is not allowed;

3.- if we delete an R_i tuple, then we must also delete all R_j tuples where $K(R_i)=FK(R_i)$ in R_j.

The foreign-key constraint does not allow for null values in $FK(R_i)$. If null values are to be used, a new constraint, denoted by NFC_{ij} is imposed. This new constraint only implies that if the value for the foreign key specified in an insertion is not null, then it must appear as a key value in the relation R_i. It has no other implicit consequences. We elaborate more on this constraint in section 4.

Since we are mapping generic operations, we do not need to be confined to a particular relational data manipulation language. We assume a language in which we are able to specify a target list (attributes whose values are to be retrieved) and a qualification term.

3. Hierarchical Mappings

3.1. Schema Transforms

The hierarchical to relational schema mapping is based on the process of normalization introduced by Codd [Codd 1970]. We assume that all the fields in the hierarchical schema are named uniquely and that each segment type in the hierarchical schema has a *hierarchical key*. A hierarchical key is a key, in the relational sense, the values of which are unique within a parent segment in the database. This corresponds to an IMS-like database [IBM 1975]. The schema mapping algorithm transforms each segment type into a relation and propagates the hierarchical keys of all the ancestor segment types of any particular segment into the relation generated from that segment. The hierarchical keys propagated into a relation correspond to foreign keys of the relation. More formally:

Let S be a hierarchical schema with k segment types and m hierarchical links. The schema mapping is:

1.- For each root segment type H_i define a relation R_i such that

  1.1 R_i contains one attribute for each field of H_i;

  1.2 the key of R_i is equal to the hierarchical key of H_i.

2.- Recursively, for each dependent segment type H_j, for which a relation R_i has been generated for its parent segment type H_i, and the hierarchical link in which it is a child, define a relation R_j such that

  2.1 R_j contains one attribute for each field of H_j, and the attributes of the key of R_i;

  2.2 the key of R_j is equal to the hierarchical key of H_j plus the key of R_i;

  2.3 the constraint FC_{ij} is introduced.

3.2. Operation Transforms

For the hierarchical system, we will assume an IMS-like record-at-a-time navigation language. The basic commands are **get-next** and **get-next-within-parent**. We use a simple syntax in our query language to avoid the intricacies of IMS. We assume an **output** command that makes available for further processing specified field values of an accessed segment occurrence. For simplicity we make the assumption that for all algorithms the traversal of the hierarchical database starts from the first hierarchical database record. This condition can be easily guaranteed by means of a suitable **get-unique** command in IMS. The algorithms assume a global enumeration of the segment types in the hierarchical definition tree. Furthermore, for each retrieval involving a target list and/or a qualification term, a new enumeration of the referenced segment types, according to their position in the hierarchy, is assumed. With these enumerations we are able to present the algorithms in a concise form.

Projection

Projection of more than one attribute of a relation based on a segment type generally requires a recursive algorithm with extensive sequential search of the database. This is because a relation can be generated from the attributes contained in more than one segment type due to hierarchical key propagation. To form the projection, first the segment types which contain the fields corresponding to the projected attributes are determined. By construction of the relations the segment types must be in the same hierarchical path. Suppose that these segment types are $H_1, H_2, \cdots, H_k$. The segment type H_1 is the one at the highest level in the tree.

Informally, the algorithm for projection is: For each highest level referenced segment (of type H_1) in the database, retrieve the second highest-level referenced descendant segments (of type H_2) and continue likewise down the hierarchy until the lowest level referenced segments are retrieved. Because we assume an IMS-like hierarchical language, this retrieval will be according to a preorder traversal of the database.

```
PROJECTION

loop
  get-next H1 segment
  exit if none
  output referenced field values
  R-C-U (1,k)
end loop

recursive procedure R-C-U (i,k)
  i ← i+1
  exit if i>k
  loop
    get-next-within-parent Hi segment
    exit if no more children
    output referenced field values
    R-C-U (i,k)
  end loop
```

The algorithm for projection recursively nests **get-next-within-parent** commands. Some hierarchical systems may not support such a feature. For these systems, we may need selections on the maximum path length that can be referenced in a query. IMS/VS offers a code option (D-option) that allows path calls, i.e., multiple segments in a hierarchical path are retrieved in a single call. The retrieved segments are concatenated in the user's I/O area. Repeated path calls and a masking of the unspecified fields would produce the desired result of projection.

Selection

We want to retrieve tuples of a relation according to a qualification which is a Boolean *(AND/OR)* of simple conditions. We assume that the qualification t can be partitioned into separate t_i terms where each t_i applies to only one segment type H_i. We have in the target list reference to one or more segment types H_i and in the qualification one or more terms t_i. There are three cases.

The first case is when the Boolean operator between t_i terms is only *AND*. That is, the qualification is t_1 *AND* t_2 *AND* $\cdots$ *AND* t_k, where the t_i's are a series of simple conditions on respectively the segment type H_i. It may be that for a segment H_j no qualification is present. In this case the corresponding t_j term evaluates trivially to *true* for *AND* processing. The algorithm for selection is similar to the one for projection. The strategy is to first visit the highest level segment and evaluate the condition. If the evaluation results in a *true* value, the descendants are accessed in turn for further condition evaluation.

```
SELECTION    (only AND terms)

loop
  get-next H1 segment where t1
  exit if none qualifies
  R-C-Q (1,k)
end loop

recursive procedure R-C-Q (i,k)
  i ← i+1
  exit if i>k
  loop
    get-next-within-parent Hi segment where ti
    exit if no more children qualify
    if i=k then
      output referenced field values
    R-C-Q (i,k)
  end loop
```

The second case arises when the Boolean operator between t_i terms is only *OR*. In this case it is necessary to evaluate the condition on each H_i until one that is *true* is found. When a t_i which is *true* is found, then all descendants qualify regardless of the truth of their t_js. Note that if no qualification is given for a segment H_i, then the corresponding t_i evaluates trivially to *false* for *OR* processing.

```
SELECTION    (only OR terms)

loop
  get-next H1 segment
  exit if none
  if t1
    then R-C-U (1,k)
    else RETRIEVE-CHILDREN (1,k)
end loop

recursive-procedure RETRIEVE-CHILDREN (i,k)
  i ← i+1
  exit if i>k
  loop
    get-next-within-parent Hi segment
    exit if no more children
    if ti
      then R-C-U (i,k)
      else RETRIEVE-CHILDREN (i,k)
  end loop
```

The final case arises when *AND*s and *OR*s are mixed between t_i terms. In this case the query can be transformed to a normal form (conjunctive/disjunctive) and the terms processed independently for each hierarchical path. The results of each path evaluation are merged to form the response to the query. This of course is a very expensive algorithm which may require several passes over the entire database. We conjecture that the semantics for such a query are very ambiguous and thus it would rarely be asked.

Join

We will give algorithms that translate natural joins between two relations R_i and R_k. The Booleans between join terms will be restricted to *AND*'s since the appearance of an *OR* connective will generally require a sequential search. For simplicity, assume that the target list may include domains from H_i and H_k only. We consider two cases for our join algorithm:

Case (a) Both H_i and H_k appear in the same branch of the hierarchical definition tree. In this case we have a branch $H_1, \cdots, H_i, \cdots, H_k$ where $0=\text{level}(H_1) \leq \text{level}(H_i) < \text{level}(H_k)$. By construction of the relations

we have: $K(R_i) \subseteq K(R_k)$

We call a *key−join−term* the join of the whole key in R_i with the equivalent part in R_k. The key-join-term may or may not be included in the join qualification by the user. There are, of course, different semantics in each case. In addition, the processing of the query is drastically different. The hierarchical organization of the database is geared very well for the case where the key-join-term is included. In such a case we join a segment occurrence with all its descendants. By contrast, the absence of the key-join-term leads to an extremely expensive algorithm for the join.

When the key-join-term is included in the qualification, the translation algorithm is similar to those given for projection and selection. By construction, we know that all descendants will qualify according to the key-join-term. In addition, because of the uniqueness of hierarchical keys along a hierarchical path, only the descendants of a segment can be joined to a given parent according to the key-join-term. Therefore, we sequentially retrieve all higher level segments and join each of them with their descendants according to the join terms that are not part of the key-join-term. Suppose that we have a join between H_i and H_k with m join terms in addition to the key-join-term.

```
JOIN (a)        Key-join-term included

loop
  get-next H_i segment
  exit if none
  loop
    get-next-within-parent H_k segment
      where (F_k1=value(F_i1)) AND, ···,
                  ···,AND (F_km=value(F_im))
    exit if no more children
    output referenced field values
  end loop
end loop
```

Suppose now, that the key-join-term is not included in the qualification. If the join is between H_i and H_k we basically need to sequentially retrieve each H_i segment and join it with all H_k segments.

```
JOIN (b)        Key-join-term not included

loop
  get-next H_i segment
  exit if none
  reset currency to start of database
  loop
    get-next H_k segment
      where (F_k1=value(F_i1)) AND, ···
                  ···,AND (F_km=value(F_im))
    exit if none qualifies
    output referenced field values
  end loop
  reset currency to last H_i segment
end loop
```

We note that suitable mechanisms must exist for setting and resetting the currency indicators as required. We do not show these operations in detail in our algorithm.

The two previous algorithms are for extreme cases. In the first case we have all the hierarchical keys from the root to the highest level segment type H_i, appearing in the qualification. In the second case we have no keys. Different algorithms are required for intermediate cases. For instance, if the equality of the root-key is included in the qualification, then we only join segments that are in the same database record. The last algorithm can also handle the case where we have H_i and H_k appearing in different definition trees.

```
JOIN (c)        Common-join-term included

loop
  get-next H_int segment
  exit if none
  loop
    get-next-within-parent H_i segment
    exit if no more children
    loop
      get-next-within-parent H_k segment
        where (F_k1=value(F_i1)) AND, ···,
                    AND (F_km=value(F_im))
      exit if no more children qualify
      output referenced field values
    end loop
    reset currency to last H_i segment
  end loop
end loop
```

Case (b) H_i and H_k appear in different branches of the same definition tree.

Let $H_1, \cdots, H_{int}, \cdots, H_i$ and $H_1, \cdots, H_{int}, \cdots, H_k$ be the two branches. By construction, both R_i and R_k will have the hierarchical keys of $H_1, \cdots, H_{int}$ as common attributes (part of their relational key). If a join term for these common attributes (*common−join−term*) does not appear in the qualification, then we use sequential search (like algorithm *JOIN* (*b*)). Otherwise, we join segments that appear in the same hierarchical subtree, i.e., are descendants of the same H_{int} segment. We can easily determine H_{int} by tracing up the hierarchical defintion tree from H_i and H_k until we reach a common ancestor.

Algorithms for intermediate cases, where the equality of only part of the common hierarchical key attributes is included are not presented here. These algorithms depend on which part of the common-join-term is present.

Deletion

In order to delete a tuple or tuples in a relation, we first have to select the tuple(s). If we qualify on the key of a relation, we are guaranteed to select a tuple uniquely. Otherwise, several tuples may qualify. To select the segment(s) to be deleted, we can use the algorithms for selection given earlier. However, now at the final level of selection, a get-hold retrieval command is used. The tuple(s) is (are) then deleted by the command:

delete H segment

The hierarchical deletion of H triggers the deletion of all descendants of H. However, this is expected by the relational user because of the foreign-key constraints for the relations.

Update

Like deletion, update of a relation requires that the tuple(s) to be changed first be selected. The update is then trivial.

replace field-value in H **with** given value

Updates of key values in a relational database are usually not allowed and this selection is also important for a hierarchical database using hierarchical keys.

Insertion

The insert operation in a relational system is conceptually simple. Tuples are specified and inserted directly in a relation. There is no ordering among the tuples. Inserting a segment in a hierarchical database is a more complicated operation. It usually involves a search for the appropriate place where the new segment is to be placed. The search, in an IMS-like language, is based on the hierarchical key values of ancestor segments. A hierarchical path is established and the segment is inserted at the end of the path.

For our transforms, the hierarchical path for insertion is implicitly provided by construction of the relations (propagation of hierarchical keys). Let $H_1, H_2, \cdots, H_k$ be segments in the same hierarchical path, and $R_1, R_2, \cdots, R_k$ be the corresponding relations. Let $V_1, V_2, \cdots, V_{k-1}$ be the values of all the attributes corresponding to foreign keys in R_k (those correspond to the hierarchical keys of $H_1, H_2, \cdots, H_{k-1}$). Suppose we insert a tuple in R_k. Unless all values for the foreign-keys in R_k exist in a hierarchical path, the insertion will be invalid. The reason is the presence of the foreign-key constraints for the relations $R_2, R_3, \cdots, R_k$ in the relational schema.

```
INSERTION

if k = 1 then    (root segment)
   begin
     insert H_k segment
     terminate
   end

i ← 1
if k = i + 1 then    (second level segment)
  begin
    get-hold-next H_i segment where (K(H_i) = V_i)
    terminate if none qualifies (integrity error)
    insert H_k segment
    terminate
  end

get-next H_i segment where (K(H_i) = V_i)
terminate if none qualifies (integrity error)
loop
   i ← i + 1
   exit if i = k - 1
   get-next-within-parent H_i segment
        where (K(H_i) = V_i)
   terminate if no more children qualify
        (integrity error)
end loop

get-hold-next-within-parent H_i segment
        where K(H_i) = V_i)
terminate if no more children qualify
        (integrity error)
insert H_k segment
```

4. Network Mappings

4.1. Schema Transforms

The mapping from a network to a relational schema is based on a one to one correspondence between record types and relations, and between data items and attributes. For record types that are members in a network link (set type), the corresponding relation has additional attributes. These additional attributes express the relationships implied by the network links and correspond to the data items that constitute the key of the owner record type. Network links differ from hierarchical links in that they have flexible set membership requirements. Set membership may be either *automatic* or *manual*. In addition, the connection may be *optional*, *mandatory* or *fixed*. The reader is reminded of the foreign-key constraint FC_{ij} and its implicit consequences. In contrast to the hierarchical transforms, values for the foreign-keys need not necessarily be specified in the network case. Therefore, the constraint NFC_{ij} which allows for null values in the foreign-key and has no implicit consequences may apply.

More formally, let S be a network schema with k record types and m network links. The schema mapping is:

1.- For each record type N_i define a relation R_i such that:

  1.1 R_i contains one attribute for each data-item of N_i;

  1.2 if N_i has a key, then the key of R_i is equal to the key of N_i; otherwise the key of R_i is equal to the database key of N_i, which appears as an explicit attribute of R_i.

2. For each network link L_{ij}, with owner record N_i and member record N_j, define relational integrity constraints and change the existing relations such that:

  2.1 the key of N_i appears as a foreign key of N_j;

  2.2 one of the following sets of constraints applies depending on the type of set membership (R_i and R_j are the relations corresponding to N_i and N_j respectively):

i. fixed automatic
FC_{ij}
Additional consequence:
(a) if we update an R_j tuple, then the value of $FK(R_i)$ in R_j *cannot* be changed.

ii. fixed manual
NFC_{ij}
Explicit consequences:
(a) if we delete an R_i tuple, then we must also delete all R_j tuples where $K(R_i)=FK(R_i)$;
(b) if we update an R_j tuple, then the value of $FK(R_i)$ in R_j *cannot* be changed, *unless* it is null.

iii. mandatory automatic
FC_{ij}

iv. mandatory manual
NFC_{ij}
Explicit consequences:
(a) if we delete an R_i tuple, then we must also delete all R_j tuples where $K(R_i)=FK(R_i)$;
(b) if we update an R_j tuple, then the value of $FK(R_i)$ in R_j *cannot* be changed *to* null.

v. optional automatic
NFC_{ij}
Explicit consequences:
(a) if we delete an R_i tuple, then we must change all R_j tuples, such that $FK(R_i)$ in R_j becomes null;
(b) if we insert an R_j tuple, then the value of $FK(R_i)$ in R_j *cannot* be null.

vi. optional manual
NFC_{ij}
Explicit consequence:
(a) if we delete an R_i tuple, then we must change all

R_j tuples, such that $FK(R_i)$ in R_j becomes null.

4.2. Operation Transforms

The network language that we use for the translations is loosely based on the facilities of IDMS which is a subset of the DBTG specifications [IDMS 1975]. For simplicity of illustration, we assume that, in addition to the different forms of the **find** command, we have a **locate** command. This command is the combination of a **find** command and the programming language statements for evaluation of a qualification for the "found" record. The qualification is a Boolean combination of simple conditions on data-item values. The retrieval algorithms are only discussed here.

Projection

We discuss the general algorithm for projection translation where all attributes in a relation R_k which corresponds to a record type N_k are projected. Assuming that in general N_k will participate as a member in several sets, we sequentially retrieve all N_k records. For each of them we also retrieve all owners N_i in the set types L_{ik} in which the N_k records participate as members.

If an N_k record occurrence is not currently a member in a set type L_{ik}, then the *null* value is output for the foreign key. There may be other versions of this algorithm depending on which attribute is projected. For instance, if a foreign key is projected, it may be better to access all owner records N_i sequentially and then, if they have an N_k member in the set they own, output their key value. In general, even though the projection and its translation are conceptually very simple, the network operations may prove to be very expensive. The main reason is that, for at least one record type, all occurrences in the database must be accessed sequentially.

Selection

Basically, two strategies may be used in the translation of the selection operation. In the first strategy, the record type N_k (corresponding to R_k) is used as an "anchor" record type. In the second strategy the record type N_i which is the owner in a network link L_{ik} where N_k participates as a member is the anchor record type. It may not always be clear which strategy is the best for a particular situation. However, as we will show, there are some clear-cut cases.

Consider a network schema with two record types N_i and N_k, a network link L_{ik} and the corresponding relational schema. Assume that we have a selection with the qualification term "t_i and t_k", where t_i is a simple condition on the foreign key in R_k and t_k is a qualification on any other attributes of R_k. Clearly, if t_i is of the form $(FK(R_i)=v)$ it may be better to use the owner as the anchor record. On the other hand, if t_k is of the form $(K(R_k)=v_1)$ and t_i is of the form $(FK(R_i)\neq v_2)$, it is very likely that the other approach is less expensive. We also note that if t_i is of the form $(FK(R_i)=null)$, then neither of the above approaches works. A new algorithm is needed which proceeds along the lines of the algorithm which uses the member as anchor, except it outputs referenced field values when N_k is not a member in a set L_{ik}.

Qualifications with *OR* Booleans complicate the translation algorithms. Consider again the previous simple network and relational schemas. Assume that we have the selection on R_k with the qualification term "t_i *OR* t_k". For each N_k record, we have to retrieve the owner record irrespectively of the truth value of t_k.

We have considered selections on relations that involve only one foreign key. The general case, where many foreign keys appear, is similar. Suppose that we perform a selection on R_k. We retrieve sequentially all N_k records in the database. For each of the retrieved records, we access the owners R_i in every set L_{ik} in which they participate as members collecting all key values. We may now evaluate the qualification and decide whether to keep the retrieved values.

Join

In the relational model a join operation is allowed between two relations if the joined attributes are compatible. Because of the freedom that the user has in forming joins, he may form joins which are not meaningful. Behind any meaningful join there is an inherent relationship. According to current specification of the network model (DBTG), the only inter-record relationships are the one's that are explicitly expressed with a set type declaration. The network DML is designed around the set concept so that it can take advantage of the underlying relationships. Hence, the only relational joins whose translation would be efficient (where the network DML commands can take advantage of the set construct) are those that have the foreign-key equality in the qualification. In this case we join only the records that are in the same set occurrence (owner with its members). Thus, we first retrieve an owner record N_i and then sequentially all of its member records N_k for all owner records of type N_i. Every other join that involves two or more record types will require sequential searches in the entire database.

Insertion

To insert a tuple into a relation R_k we need to specify values for attributes of R_k. The attributes of R_k are composed of data-items from the record type N_k plus data-items, corresponding to keys, of record types N_i, $i=1, 2, \cdots, m$, which are owners of the N_k record type in set type L_{ik}. We denote by $V_1, V_2, \cdots, V_m$ the values of these latter attributes which correspond to the foreign keys in R_k. The constraints for insertion are those given previously for the different types of set membership.

In translating a tuple insertion, we first store the record N_k. We then manually insert it into all sets in which its membership is manual and a value has been specified for the foreign key in N_k of the owner N_i. If no such owner exists, then an integrity error has occurred. The record is automatically inserted into all set types for which its membership is automatic according to the set occurrence selection specified in the schema. Procedurally, this latter process will be similar to that for manual set membership except that a value *must* be supplied for the foreign key in N_k of the owner record types.

```
INSERTION

store N_k record
exit if integrity error
for each set L_ik in which N_k is a manual member
        and V_i ≠ null
   locate next N_i record where (K(N_i)=V_i)
   if none then
      begin
         delete N_k record
         terminate (integrity error)
      end
   insert N_k record into L_ik set
end for
```

If N_k never participates as a member in a network link, only the **store** operation will be executed.

Deletion

A simple **delete only** statement in the network system corresponds to the relational delete on the given relational schema. In addition, a selection operation may also have to be translated to select the desired records to be deleted. Note that the **delete N_k record only** statement has the following properties:

p1. Removes an N_k record from all set occurrences in which it participates as a member.

p2. Removes, but does not delete, all optional members for each set where N_k participates as an owner.

p3. Deletes all fixed and mandatory members for each set where N_k participates as an owner. If any of the deleted fixed or mandatory members is itself the owner of any set occurrences, then the delete statement is executed on that record as if it was the object record (N_i) of a **delete only** statement (i.e. triggered deletion).

There is an analogy with the expectations of the relational user when he deletes a tuple of a relation R_k.

p1'. When a tuple is deleted, all attribute values are deleted, including the foreign keys (removal of N_k from all sets as a member).

p2'. For optional membership in sets, where N_i is the owner, we have in the relational schema an assertion that foreign key values in R_i are updated to *null* (optional members are removed).

p3'. For mandatory or fixed membership in sets, where N_k is the owner, we have an assertion in the relational schema that the deletion is triggered.

Update

Depending on the particular attribute that we want to update, a relational **update** command will be translated to a simple **modify** command or to a **remove** command or to an **insert** command or a combination of **remove** and **insert** commands.

Suppose we want to update the value of an attribute A in the relation R_k with the value V. Basically, we consider two cases. In the first case A corresponds to a data item in the corresponding record type N_k and we need a **modify** network command. In the second case A is a foreign key. This translation to network operations does not include changes of physical values but removals and/or insertions in set occurrences.

```
UPDATE (a)    A is not a foreign-key

if A=K(R)
   then drop the update
   else modify N_i record with V

UPDATE (b)    A is a foreign-key

case 1: fixed-automatic membership
   drop the update

case 2: fixed-manual membership
   if V=null or value(A)≠null
      then drop the update
      else begin
           locate next N_i record where (A=V)
           if none then
                terminate    (integrity error)
           insert N_i record into L_ki set
           end

case 3: mandatory-automatic membership
   if V=null
      then drop the update
      else switch sets for N_i using V

case 4: mandatory-manual membership
   if V=null
      then drop the update
      else if value(A)=null
         then begin
              locate next N_k record where (A=V)
              if none then
                   terminate    (integrity error)
              insert N_i record into L_ki set
              end
         else switch sets for N_i using V

case 5: optional-automatic or manual membership
   if V=null
      then begin
           locate next N_k record where (A=FK(R_k))
           remove N_i record from L_ki set
           end
      else begin
           locate next N_k record where (A=FK(R_k))
           if located
              then remove N_i record from L_ki set
           locate next N_k record where (A=V)
           if located
              then insert N_i record into L_ki set
           end
```

The **switch sets** statement transfers the record from one owner to another owner with the same set type. This facility is available explicitly in some network systems; in others it would have to be implemented by a procedure.

The easiest way to deal with composite keys is to treat them as a whole. For instance, a foreign composite key in a relation is updated only if all attributes in the key are updated. In this way, we can avoid situations where a composite foreign key has some attributes with null values and others with non-null values.

5. Concluding Remarks

A general framework for transforming schemas and mapping operations between relational and hierarchical/network models is outlined. Rather than translating specific schemas and operations, we present generic translation algorithms. The algorithms presented are by no means complete or optimal. Optimization of the transaction translation algorithms is a very important problem if they are to be useful.

There is a strong emphasis in the framework on the concept of constraints on the database and their

preservation in a schema or operation transform. The model to be used for the expression of the mappings must be very powerful in its treatment of constraints. We had to extend the relational model for the description of just the inherent constraints of the hierarchical/network structure.

It seems that null values will inevitably appear in the mappings. However, the reader may have noticed that in our approach no actual physical storage of null values in the hierarchical/network database was necessary. We assumed that in our "virtual" relational interface we could use null values in modifications, e.g., inserting a relation tuple corresponding to a record which is a member in an optional-manual set. Also, answers from the hierarchical/network database could have null values. This does not imply though that a mechanism exists for treating null values. In particular, there is no indication that a database with null values is queried or modified.

The capability to deal with nulls is necessary in some cases. Consider, for instance, that there is no "virtual" interface but that the relational user has an actual underlying relational database with the same description as a network database (operation-equivalent). It is expected that the user does not distinguish between the two databases and can ask for the same transactions in either one. In such a case, null values may be introduced in the underlying relational database.

Ways for dealing with null values range from rudimentary ones, e.g., zeros, values like any other, to more sophisticated ones [Codd 1979, Lipski 1979, Vassiliou 1979, Zaniolo 1979]. We note that the above work considers only the retrieval aspects of the problem. That is, how queries are evaluated in the presence of null values. More work is required along the lines of examining how modifications and semantic constraints behave in the presence of null values [Vassiliou 1980].

Our direct mappings of transactions between DBMSs is the first step in determining the nature of the interface problem for transaction translation among heterogeneous DBMSs. Because organizations have a high investment in current DBMS software, application programs and data storage, we do not foresee a wholesale conversion to a common DBMS for distributed applications. Instead, organizations will allow local installations to retain their data model and data language. However, at the same time, they would also like to permit some cooperation with other databases for purposes of data integration and sharing. We believe that transaction translation can provide the facilities to support the integration of heterogeneous, distributed DBMSs.

References

[Adiba 1977]

Adiba, M., and Delobel, C., "The Problem of Cooperation Between Different DBMS," in *Architecture and Models in Data Base Management Systems*, (Nijssen, G.M., ed.), pp. 165-186.

[Adiba 1978]

Adiba, M., Chupin, J.C., Demolombe, R., Gardarin, G., and Le Bihan, J., "Issues in Distributed Data Base Management Systems: A Technical Overview," *Proc. ACM Int. Conf. Very Large Data Bases*, pp. 89-110.

[Brodie 1978]

Brodie, M.L., *Specification and Verification of Data Base Semantic Integrity*, Ph.D. thesis, Department of Computer Science, University of Toronto.

[CODASYL 1971]

CODASYL Data Base Task Group Report, Conference on Data Systems Languages, ACM, New York.

[Codd 1970]

Codd, E.F., "A Relational Model of Data for Large Shared Data Banks," *CACM* 13, pp. 377-387.

[Codd 1979]

Codd, E.F., "Extending the Database Relational Model to Capture More Meaning," *ACM TODS* 4 (4), pp. 397-434.

[Deppe 1976]

Deppe, M.E., and Fry, J.P., "Distributed Data Bases A Summary of Research," *Computer Networks* 1, pp. 130-138

[IBM 1975]

IMS/VS, General Information Manual, IBM Publication No GH20-1260, IBM Corp., White Plains, New York.

[IDMS 1975]

IDMS – General Information Manual, Cullinane Corporation.

[Klug 1978]

Klug, T., *Theory of Data Base Mappings*, Ph.D. thesis, Department of Computer Science, University of Toronto.

[Lipski 1979]

Lipski, W.Jr., "On Semantic Issues Connected with Incomplete Information Databases," *ACM TODS* 4 (3), pp. 262-296.

[Lochovsky 1979]

Lochovsky, F.H., (editor), "A Panache of DBMS Ideas II", *CSRG Tech. Report 101*, pp. 92-122.

[Rothnie 1977a]

Rothnie, J.B., and Goodman, N., "An Overview of the Preliminary Design of SDD-1: A System for Distributed Databases," *Proc. Berkeley Workshop on Distributed Data Management and Computer Networks*, pp. 39-57.

[Rothnie 1977b]

Rothnie, J.B., and Goodman, N., "A Survey of Research and Development in Distributed Database Management," *Proc. ACM Int. Conf. Very Large Data Bases*, pp. 48-62.

[Rothnie 1980]

Rothnie, J.B., *et al.*, "Introduction to a System for Distributed Databases (SDD-1)," *ACM TODS* 5 (1), pp. 1-17.

[Vassiliou 1979]

Vassiliou, Y., "Null Values in Database Management - A Denotational Semantics Approach," *Proc. ACM SIGMOD*, pp. 162-169

[Vassiliou 1980]

Vassiliou, Y., "Functional Dependencies and Incomplete Information," *Proc. ACM Int. Conf. Very Large Data Bases*.

[Zaniolo 1978]

Zaniolo, C., *Relational Views in a Data Base System: Support for Queries*, Sperry Research Center, Sudbury, Mass.

[Zaniolo 1979]

Zaniolo, C., "Design of Relational Views Over Network Schemas," *Proc. ACM SIGMOD*, pp. 179-190.