Systems for Large Data Bases, P.C. Lockemann and E.J. Neuhold, (eds.)
North-Holland Publishing Company, 1976

A DATA MANIPULATION LANGUAGE FOR RELATIONAL DATA STRUCTURES

J.H. ter Bekke
Eindhoven University of Technology
Department of Mathematics
Eindhoven
The Netherlands

Abstract

Some queries on a relational database are analyzed. This analysis results in new concepts for a high-level data manipulation language. In the proposed language the cardinal number concept from set theory is applied, using semantic information on the contents of a database. Each query formulation in this language consists of a small number of easily manageable expressions. This means that the simple relational data structures also are reflected in the programs which operate on these structures.

Keywords and Phrases: relational data model, data manipulation language, database, data structure, set theory, cardinal number, subset invariant.

CR Categories: 3.70, 4.22, 4.33, 4.34.

0. Introduction

A database consists of a collection related data, considered to be of interest for different applications. In the relational model [1], all data are represented in relations. Each relation contains data for a set of similar entities, such that every tuple in a relation represents one entity. In a relation the ordering of tuples is considered to be irrelevant. The same applies to the ordering of attributes which together form a relation.
With this datamodel mathematically inspired high-level data manipulation languages, e.g. based on the relational calculus and the relational algebra, were proposed as languages for databases [2,3]. Other languages were also developed which do not make use of these mathematical concepts [4].
These languages have a disadvantage in common, in that they often lead to one single and therefore formidable expression for a query on a database. In contrast with these languages, our proposal for a data manipulation language leads to a problem formulation which also for complex queries consists of a small number of simple expressions.
The key idea in this approach is the use of "subset invariants", which exist between database relations and which embody semantic information on the database. They naturally lead to the use of the cardinal number concept from set theory.

This paper consists of three sections. First a description is given of a sample database, which will be used in all examples. The second section gives an informal introduction to some concepts of our data manipulation language. The examples in this section have been selected to illustrate the retrieval part of the proposed language. Finally, a short comparison is presented with other data manipulation languages for relational databases.

1. Database definition

A relational database consists of a collection time-varying relations. Each relation in a database is used to represent data about a set of similar entities or data about a set of similar associations between entities. In the relational data model there is no difference between entities and associations.

The database definition provides information about the entity-types which are of interest to a community of users. For all applications it is the base of operation. A data manipulation language is used for maintenance and retrieval of data. The latter enables the user to select data from existing entities, but also to derive new information (possibly new entities) from them.

All examples in this paper concern a database which consists of seven normalized relations (i.e. attribute values are not compound values). In each relation one attribute or a number of attributes (called the primary key of a relation) uniquely identify the tuples; in the definition below these attributes are enclosed by square brackets.

rel S.[*sn*], *sname*, *location*

rel P.[*pn*], *pname*, *stock*, *min stock*, *max stock*

rel A.[*sn*,*pn*], *price*

rel T.[*sn*,*pn*,*date*], *quantity*

rel C.[*pn*,*dn*], *quantity*

rel D.[*dn*], *dname*, *location*

rel E.[*en*], *ename*, *location*, *dn*

Semantics of these relations

- Relation *S* contains data about suppliers. For each supplier are recorded the attributes: *sn* (supplier number), *sname* (supplier name) and *location* (city of operations).
- Relation *P* contains data about parts. For each part are recorded the attributes: *pn* (part number), *pname* (part name), *stock* (actual stock), *min stock* (lower stock limit) and *max stock* (upper stock limit).
- Relation *A* contains data about assortments, which are associations between suppliers and parts. This implies that the combination (*sn*,*pn*) appears as the primary key of the relation. Furthermore each association provides the *price* at which supplier *sn* can supply part *pn*.
- Relation *T* contains data about parts ordered. Each tuple in this relation can be considered as an association between a tuple from relation *A* and a *date* (date of order). Each association provides also the attribute *quantity* (quantity ordered).
- Relation *C* contains data about associations between parts and departments. Each association provides the *quantity* of part *pn* allocated to department *dn*.
- Relation *D* contains data about departments. For each department are recorded the attributes: *dn* (department number), *dname* (department name) and *location* (city in which it is located).
- Relation *E* contains data about employees. For each employee are recorded the attributes: *en* (employee number), *ename* (employee name), *location* (residence) and *dn* (department).

A database is not an arbitrary collection of entities. On the contrary, time-independent relationships exist between different sets of entities. In our database, the following relationship should remain invariant in time:
"The set of part numbers appearing in relation *C* (denoted by C^{pn}) is a subset of the set of part numbers appearing in relation *P* (denoted by P^{pn})".

Mathematically: $C^{pn} \subseteq P^{pn}$.

Other invariants are: $A^{sn} \subseteq S^{sn}$, $A^{pn} \subseteq P^{pn}$, $T^{sn} \subseteq A^{sn}$, $T^{pn} \subseteq A^{pn}$, $E^{dn} \subseteq D^{dn}$, $C^{dn} \subseteq D^{dn}$.

In what follows these invariants will more specifically be called "subset invariants". These subset invariants represent in fact consistency requirements, which the contents of a database must satisfy. An operation on a database could violate a subset invariant $X \subseteq Y$:

- when an element is inserted in the set *X*.
- when an element is removed from the set *Y*.

The maintenance of these subset invariants is considered to be a function for the database management system.

2. Selection from the database

In this section an informal introduction is given to the concepts which may be used in the formulation of queries on a relational database. These concepts will be introduced by examples. The following symbols are part of the definition language:

- braces: enclose optional parts of expressions.
- angle braces: enclose metasymbols.

2.1. The selection expression

The selection expression occupies a central position in our data manipulation language; it appears in each formulation of a query on a database. The general format of a selection expression will be:

(<relation> { .<e list> } { | <predicate> })

Semantics: A selection expression will be used to obtain (usually new) tuples from a database. In this process only those tuples of *<relation>* must be considered which satisfy the *<predicate>*; in the absence of this *<predicate>* all tuples are to be considered. The expressions of the *<e list>* (i.e. expressions formed with at least one attribute of the relation) must be evaluated to yield the desired data in the new tuples. However, in the absence of *<e list>* all attributes of *<relation>* are provided. It should be noted that in a selection expression no elimination of duplicate (new) tuples is carried out (unless it is stated explicitly, see query 5). The *<relation>* { . *<e list>* } part of a selection expression is also called the "target", therefore *<relation>* will sometimes be called "target relation".

The next two simple queries illustrate the use of the selection expression. The first example is of the simplest form: selection of data using only one relation of the database.

Query 1. Get names of suppliers located in paris.

get (S.sname | location = "paris")

The verb *get* preceding a selection expression indicates that the tuples identified by the selection expression are to be output. From this formulation we see immediately that relation *S* must be considered; only values for attribute *sname* are to be output in this query. Moreover, because no projection is indicated duplicate values are not removed from this list.

We do not introduce the concept of a free variable in our manipulation language. That is why attributes in the target relation are indicated simply by their attribute name. For these attributes we will also use the term "primary attribute". Primary attributes in this example are *sname* and *location*.

In the next example we do not restrict ourselves to one relation of the database.

Query 2. Get names of employees who are not employed in their place of residence.

get (E.ename | location ≠ D[dn].location)

For each employee in relation *E*, we compare the value of the attribute *location* with the value of the attribute *location* in the corresponding tuple of relation *D*. The latter value can be found by using the value of attribute *dn*, which occurs in both relations. Because this attribute is the primary key of relation *D*, we use in our manipulation language the well-known notation of the subscripted variable.

An attribute that occurs in a relation associated with the target relation will be called "foreign attribute". The foreign attribute in this example is the attribute *D[dn].location*.

In the last example two (*location*) values from different relations are compared. This comparison can be performed because of the correspondence between the tuples from which the values are taken. This correspondence is provided by the attribute *dn*, which turns out to be the primary key in one of the relations considered. This kind of correspondence must always be present if attribute values of different relations have to be compared.
For the formulation of queries the restriction that only set elements can be referred to has no adverse consequences. On the contrary, it has a positive influence on the problem analysis (this is illustrated by example 5 of this section).

Henceforward we will distinguish the next four basic elements in a selection expression:

- Primary attribute: <*attribute*>
  A primary attribute is an attribute in the target relation.
- Membership test: <*relation*> [<*key attribute*>]
  The membership test *R[p]* is *true* if the tuple with key value equal to *p* occurs in relation *R*; otherwise it is *false*. This concept will be used in example 4.
- Foreign attribute: <*relation*> [<*key attribute*>] . <*attribute*>
  If the membership test part *R[p]* in the foreign attribute *R[p].q* is *true*, then *R[p].q* yields the attribute value *q* of the tuple in relation *R* with key value equal to *p*; otherwise it yields the value *nil*.
- Constants and variables of various types.

2.2. Use of subset invariants

Next to the primary key concept, the subset invariant is another important element in our formulation of queries. In all queries where otherwise sets would have to be compared, we aim at a description in which we can profit from a subset invariant. When using this approach, we are allowed to abstract ourselves from the actual contents of the sets; only cardinal numbers of sets (the number of elements in a set) are needed in the problem formulation. This approach is illustrated by the following examples.

Query 3. Get names of suppliers who are able to supply all parts.

The condition that has to be satisfied by the relevant suppliers is:

"All part numbers in relation *P* (denoted by P^{pn}) occur also in the set of part numbers *pn* in relation *A* that can be supplied by the supplier *sn* (denoted by A^{pn}_{sn})".
Or in other words, does

$$A^{pn}_{sn} \supseteq P^{pn} \tag{1}$$

hold.
Let A_{sn} denote the subset of relation *A* corresponding with supplier *sn*.
From $A_{sn} \subseteq A$ and $A^{pn} \subseteq P^{pn}$ we have

$$A^{pn}_{sn} \subseteq P^{pn} \tag{2}$$

Hence (1) can only hold if

$$A^{pn}_{sn} = P^{pn} \tag{3}$$

Since $A^{pn}_{sn} \subseteq P^{pn}$ we may also equate the cardinal numbers:

$$count(A_{sn}^{pn}) = count(P^{pn}) \quad (4)$$

Since the combination (*sn*,*pn*) is the primary key of relation *A*, any part number occurs at most once in a subset A_{sn}, so we may also investigate the following weaker condition:

$$count(A_{sn}) = count(P) \quad (5)$$

For each supplier in relation *S* we need the cardinal number of the corresponding subset A_{sn}. For that purpose we could create a temporary relation *S'* which, next to the attributes of relation *S*, also contains the cardinal numbers of the corresponding subsets in relation *A*. This is denoted by:

$$S' := S \oplus count(A_{sn})$$

Since relation *S'* is obtained from relation *S* by appending new information to entities in relation *S*, we will use the phrase: extending a database relation.

The notation introduced so far for the extension concept is not suitable to express more complex situations which can arise in queries. Instead we will use an expression similar to the selection expression. The general format of an extension statement is:

ext <*relation*$_1$> . <*attribute*> :
<*set function*> (<*relation*$_2$> {.<*e list*> } *per* <*a list*> { | <*predicate*> })

Semantics: An extension of <*relation*$_1$> with <*attribute*> is only defined if values for attributes in <*a list*> could occur as primary key values in <*relation*$_1$>. Then each tuple in <*relation*$_1$> corresponds with a (possibly empty) collection of tuples in <*relation*$_2$> which satisfy the <*predicate*>. From this collection of tuples another collection is obtained by selecting only those attributes that are defined in <*e list*>. The value for <*attribute*> in a tuple of <*relation*$_1$> is finally obtained by applying <*set function*> to the latter collection of tuples.
An extension of a relation is temporary, which means that it is only defined in the context of the query concerned.

The extension of relation *S* with attribute *asm* using this concept would be:

ext *S.asm : count(A per sn)*

Relation *S* may now temporarily be considered to read:

rel *S.[sn], sname, location, asm*

The following set functions may be applied on tuples in normalized relations:

count: delivers the number of elements in the corresponding collection.
any : delivers the value *true* if the corresponding collection is not empty, else the value *false*.
empty: delivers the value *true* if the corresponding collection is empty, else the value *false*.
max : selects the maximum of the expression <*e list*> in the corresponding collection; for an empty collection the value *nil* will be delivered. It is obvious that it can only be applied for attributes with a defined ordering.
min : selects the minimum value of the expression <*e list*> in the corresponding collection; for an empty collection the value *nil* will be delivered. It can only be applied for attributes with a defined ordering.
total: evaluates the sum of the expression <*e list*> of the corresponding collection; for an empty collection the value *nil* will be delivered. It is obvious that addition must be defined for the attributes in <*e list*>.

The following set function may also be applied on tuples in unnormalized relations (obtained from normalized relations):

list: selects the attribute values of <*e list*> attributes from the corresponding collection; for an empty collection the value *nil* will be delivered. The result is a compound attribute.

Next to an extension, we will need in the formulation of this query a variable which contains the cardinal number of a set. This can be realized by using the following assignment statement:

var <*variable*> : <*set function*> (<*relation*> { .<*e list*> } { | <*predicate*> })

Semantics: The result of this statement is that the value of the <*set function*>, applied to the selection expression, is assigned to a temporary <*variable*>. Excepting the function *list* all afore named functions may be used in this statement.

After these preparations the formulation of the third query now becomes self-explaining:

ext S.asm: count(A *per* sn)
var p: count(P)
get (S.sname | asm = p)

The execution order of the retrieval statements is by this formulation only partial ly fixed. Because the first two statements are independent of each other, they migh be executed in parallel. However, the last statement is dependent on the results of other statements, therefore it must be executed last.

At this moment we can explain why set functions can not be used in the predicate or target of a selection expression. If a set function would occur in a selection expression, this would mean that the function must be evaluated over and over again for each tuple in the target relation. We can distinguish the following cases:

- The set function must be evaluated for each tuple in the target relation because the set function result is dependent on the values in the tuple. Before the selection of information can take place the creation of derived information is done separately by means of an extension. Within this creation process a temporary attribute is created which contains the set function result.
- The result of the set function is dependent on values in an associated relation of the database. In this case the associated relation should be extended with the desired attribute. The foreign attribute can be used to refer to this desired attribute.
- The set function provides a value which is constant for all tuples in the target relation. In this case it is obvious that a temporary variable should be created; the set function result will be assigned to this variable.

Using subset invariants, the formulation for the following query is rather obvious.

Query 4. Get names of suppliers who have at least the assortment of supplier s2.

The set of part numbers in the assortment of supplier *s2* (i.e. A^{pn}_{s2}) plays an important role in this query.
The condition that has to be satisfied by the relevant suppliers is:

$$A^{pn}_{sn} \supseteq A^{pn}_{s2} \qquad (6)$$

There is no subset invariant between these sets. Consequently, we may not use the cardinal numbers for verifying this condition; however, condition (6) is equivalent to the following condition:

$$(A^{pn}_{sn} \cap A^{pn}_{s2}) = A^{pn}_{s2} \qquad (7)$$

A trivial subset invariant exists between the sets in this condition, so that we may use cardinal numbers:

$$count(A^{pn}_{sn} \cap A^{pn}_{s2}) = count(A_{s2}) \qquad (8)$$

For the formulation of this query we will create a temporary relation. In our manipulation language this can be done by using the following statement:

rel <*relation*$_1$> . <*a list*> : (<*relation*$_2$> . <*e list*> { | <*predicate*> })

Semantics: By means of this statement a temporary <*relation*$_1$> is defined with attributes <*a list*> which are taken from <*e list*> of <*relation*$_2$>. In <*a list*> there must be at least one attribute enclosed by square brackets to denote the primary key of the new relation. On the right hand side of the colon a selection expression must be given.
(Note that whenever a key value occurs in more than one tuple as a result of the selection expression, then it is not determined which tuple will be included in the temporary relation.)

The formulation of this query is now:

rel R.[pn] : (A.pn | sn = "s2")
ext S.asm : count(A per sn | R[pn])
var asm s2 : count(R)
get (S.sname | asm = asm s2)

(Note: It is also possible to formulate this query without the creation of a temporary relation, namely by extension of relation *P*. The alternative formulation is left as an exercise for the reader.)

2.3. Derived entities

The foregoing queries were all straightforward, in the sense that attributes of existing entities were asked. In the next example we ask for attributes of entities that still have to be created since they are not present in the database.

Query 5. Get locations where at least three suppliers operate.

In this query we ask for locations of suppliers. Because there is no relation in the database with locations as entities, we must create such a temporary relation. This is done by a projection (denoted by [and]) of relation *S* on the attribute *location*. Next, we must extend this temporary relation with the number of suppliers in each specific location. The locations for which this number is at least three, must be selected. Therefore, our solution to the problem will be:

rel L.[location] : (S.location)
ext L.count s : count(S per location)
get (L.location | count s ≥ 3)

2.4. Some simple queries

The following simple queries illustrate the use of other set functions.

Query 6. Get name and assortment for each supplier in paris.

rel R.[sn], sname : (S.sn, sname | location = "paris")
ext R.asm : list(A.pn per sn)
get (R.sname, asm)

Query 7. Get part numbers and names for unused parts.

ext P.unused : empty(O per pn)

get (P.pn, pname | unused)

Query 8. For each part get part number, name and total volume of parts on order.

ext P.vol : total(P.quantity per pn)

get (P.pn, pname, vol)

The following declaration can be very useful for the formulation of queries:

def <relation>.<attribute> : <expression on attributes>

Semantics: This form enables us to split up complex expressions on attributes into simpler expressions. The result of this declaration is the apparent extension of *<relation>* with *<attribute>*, the attribute value being equal to the evaluation of *<expression on attributes>*.

2.5. Stock control problem

We conclude this series of examples with a more complex problem which illustrates how a stock control problem can be formulated using the concepts of our data manipulation language.

The attribute *stock* in relation *P* denotes the actual stock of the parts. Parts for which the *stock* is below the minimum stock, must be ordered from one of the suppliers tendering the minimum price. The ordersize is equal to the difference betwee maximum stock and actual stock. Furthermore, a list must be given specifying for each order the supplier name, location and orderlines. The orders must also be adde to relation *T* (the attribute *date* gets the value *"7603"*).
It is possible that a part to be ordered can not be supplied by any supplier. For these parts we want a list of part numbers and names.

We solve this problem by splitting it up into several simpler problems: We need the minimum price at which each part can be supplied. This is obtained by:

ext P.min price : min(A.price per pn)

Further, we have to know where the orders must be placed: A supplier is candidate to supply a certain part, if he is able to supply it at the minimum price for that part. Hence, a supplier number in an *A*-tuple is candidate if the price in the *A*-tuple is equal to the minimum price for that part in the corresponding *P*-tuple:

def A.sn cand : price = P[pn].min price

A part number in an *A*-tuple is candidate if the actual stock is below the minimum stock in the corresponding *P*-tuple, so we define:

def A.pn cand : P[pn].stock < P[pn].min stock

Next we define the ordersize:

def A.supply : P[pn].max stock - P[pn].stock

Now we can create a temporary relation *O* with all orderlines:

rel O.[pn], sn, quantity : (A.pn, sn, supply | sn cand and pn cand)

The insertion in relation *T* is obtained by the (herewith introduced) statement:

ins T.sn, pn, date, quantity : (O.sn, pn, "7603", quantity)

The list of orderlines is obtained by the following statements:

ext S.orders : any(O per sn)

ext S.orderlines : list(O.pn, quantity per sn)

get (S.sname, location, orderlines | orders)

Finally the list of undeliverable parts:

ext P.no supplier : empty(A per pn)

get (P.pn, pname | no supplier and stock < min stock)

This completes the formulation for this problem in our language.

3. Comparison with some other data manipulation languages

Rather than undertaking the formidable task of formulating the foregoing stock control problem with two other data manipulation languages for n-ary relational database systems, i.e. Alpha [2] and Sequel [4], we will present a nonexhaustive comparison of simpler queries. The Alpha language is a high level language based on the relational calculus; Sequel is based on so-called mappings.

The difference with these languages will be shown by some examples from the foregoing section. Firstly, we compare our language with Alpha:

Query. Get supplier names for suppliers who are able to supply all parts.

$$\{\ s.sname \mid s \in S \wedge \forall_{p \in P} \exists_{a \in A}\ (s.sn = a.sn \wedge a.pn = p.pn)\ \}$$

The relational calculus description is equivalent with the verification of the condition

$$A^{pn}_{sn} \supseteq P^{pn}$$

for each supplier sn, since in a relational calculus description subset invariants do not play any role. Hence, sets have to be compared instead of cardinal numbers of sets.
The single statement formulations of queries in the relational calculus leads, because of the nesting of quantifiers, to unmanageable expressions. (Even if the formulation could be split up into simpler expressions, we would obtain a description which differs completely from our proposal.)

The concept of primary key, which plays an important role in verifying the database consistency, is used heavily in our data manipulation language. On the other hand in Sequel no such concept exists. In the next example we see that such a concept facilitates a query formulation substantially.

Query. Get names of employees who are not employed in their place of residence.
In Sequel there are two possible (single statement) formulations for this query. Overlooking the primary key concept one might get a formulation analogous to Q 7 in reference [4].

```
select ename
from e in E
where dn in select dn
           from D
           where location ≠ e.location
```

For each employee we have to create a subset of department numbers (this subset may be pretty large). Furthermore, we must verify whether a given department number is included in this set.
This formulation may lead to a very time-consuming implementation. It is caused by the order in which the user has given the two "jointerms". This together with the overlooking of the primary key concept leads to the description above.
In our language the user must specify the order in which the jointerms have to be evaluated and lean heavily on the primary key concept.
In this example we see that a formulation using Sequel has a network structure. For simple queries this has no unpleasant consequences, but complex queries are almost unmanageable.

4. Conclusion

The relational datamodel is often associated with high-level data manipulation languages based on the relational calculus or relational algebra. In this paper it is indicated that these languages lead to unmanageable problem formulations. The use of semantic information about relations in the database and the possibility of introducing temporary relations or temporary extensions of existing relations leads to an approach in which this disadvantage does not exist. Each formulation in the proposed language follows naturally from the problem specification. It consists of a small number of simple statements; each statement representing a necessary step in the derivation of the desired result. These statements form a description which is easily adapted to variations in the problem specification.

Acknowledgment

The author is indebted to professor R.J. Lunbeck and to F.J. Peters and F. Remmen for their valuable remarks during many fruitful discussions.

References

1. Codd, E.F. A Relational Model of Data for Large Shared Data Banks. Comm. ACM 13, 6 (June 1970), 377-387.
2. Codd, E.F. A Data Base Sublanguage Founded on the Relational Calculus. Proc. 1971 ACM SIGFIDET Workshop, San Diego, Calif, Nov. 1971, 35-68.
3. Codd, E.F. Relational Completeness of Data Base Sublanguages, Courant Computer Science Symposia. Vol. 6: Data Base Systems, Prentice Hall, Englewood Cliffs, N.Y. 1971.
4. Astrahan, M.M., Chamberlin, D.D. Implementation of a Structured English Query Language, Comm. ACM 18, 10 (Oct. 1975), 580-588.