Group to Group Relations

by Lincoln Stoller, Ph.D.
11/7/91

Introduction

This article explains the use and properties of a fundamental relation I call a group-to-group (G-to-G) relation. A G-to-G relation exists between two groups of records, each of which are in different files. For example, if a file of customers has a field that records state and there's also a separate state file, then a group of state records can be related to customers located in that state. The relationship between the states and the customers is a group-to-group relationship.

G-to-G relations differ from the familiar one-to-many relations established in 4D's structure editor. Before describing G-to-G relations any further I will review 4D's basic tools for building file relations.

4D File Relation

4D relates files using one-to-many relations are established by drawing lines between files in the structure editor. A relating line has an arrowhead pointing to the record in the file that is RELATED TO with the tail of the arrow leading from the record in the file that's RELATED FROM. The field in the file that is related to stores the value of the field of the file related from. These one-to-many relations are the basic building blocks used to form other file relations.

The line that connects two files in the structure editor is somewhat deceptive since it implies that some additional feature is added by its presence. In fact a 4D relation is little more than a copy of a value stored in a record of one file kept in a field of a record in another file.

When you ask 4D to fetch a related record or records you're asking 4D to execute a file search. RELATE ONE or RELATE MANY commands initiate this search on records in the related to or related from file respectively.

The immediate consequences of these file relationships are:

1) In order to locate a record, the value assigned to that record should be unique. When the value is unique it can be considered a "primary key." In correctly normalized relational database structures every file has one and only one primary key.

2) To make sure that relating values are not changed these fields are not user modifiable. As a consequence the fields used to relate files are managed by the program itself and not by the user. This means that relating values are *extrinsic* to the data entered by the

user; relating fields are created to maintain the relational file structure and are both assigned and used in a manner invisible to the user.

Relating fields are subject to uniqueness constraints and are internal to the database structure. The G-to-G relationship that we now consider involves fields not subject to either of these conditions.

Group-to-Group Relationship

A group-to-group relation occurs when a number of records in one file have something in common with records in another file. The group of records in the first file, which I'll call group A, don't need to have anything in common with each other (except that they have the same structure and are in the same file). Similarly the records in the group in the second file, which I'll call group B, also don't need to share any intragroup quality. It's only necessary that for every group A record there is at least one group B record with a shared feature. This is actually overly general and I'll make it more precise in a moment.

Considering the previous example, Group A is made up of customers and Group B is made up of states. All the customers in Group A may be in different states and all the states in Group B may be different. The customer records and the state records are in a group-to-group relation if for each of the customer's states in the customer group there is a matching state in the state group.

The related groups concept can be applied in any DB and in most DBs it can play a central role. However, 4D provides no tools to implement such relation so we hve to create these tools ourselves.

1-to-Many and Group Relationships

A parent child relationship can be written as:

[Mothers]<-----[Sons]

It makes an equal amount of sense to consider a group of mothers and ask for the group consisting of their sons as it does to start with the sons and trace back to their mothers. However, the elements found in the related group depend on the group you choose to begin with; the related group will be differend depending on the direction you trace the relation.

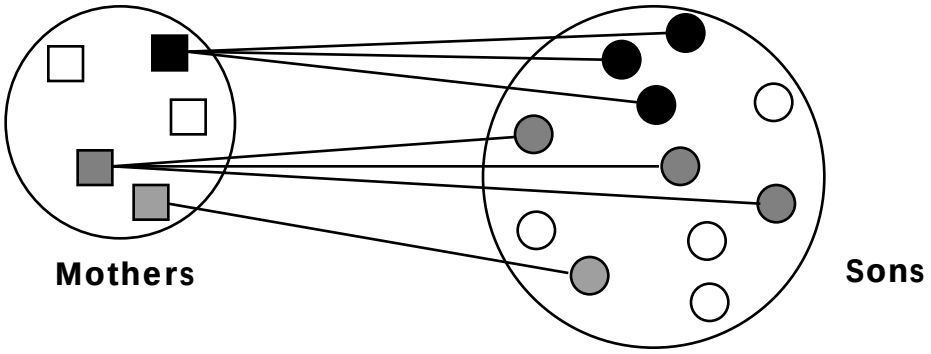

Figure 1: Mother and son records in a one-to-many relation.

In Figure 1 you can see that different groups of children many share the same parents but different groups of parents are not, generally, related to the same children . To put it more simply: in a 1-to-many relationship lineage is not symmetric.

In G-to-G relations lineage is symetric. Consider the G-to-G diagram in Figure 2.

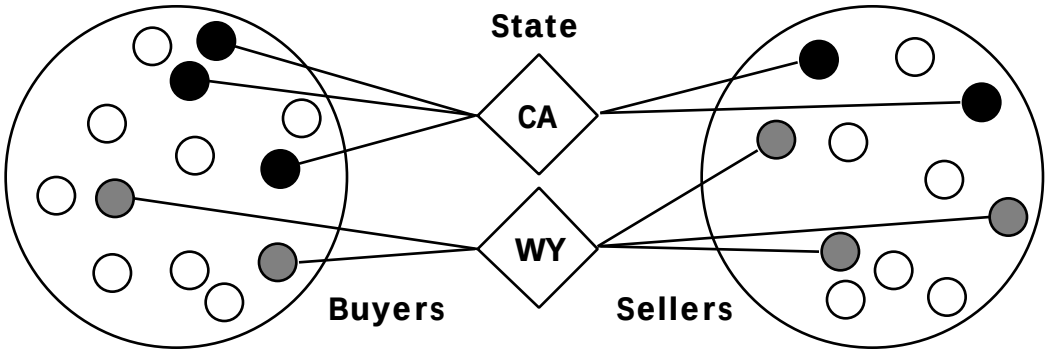

Figure 2: Buyer and seller records in a group-to-group relation.

Just as different groups of buyers can be related to the same group of sellers, so can different groups of sellers be related to the same group of buyers. As the diagram shows there are different groups of buyers that share these same states with different groups of sellers.

It's important to note that we're finding *all* the buyers (or sellers) that share their states with a *select* group of sellers (or buyers). We are not picking a state and then finding all the buyers and all the sellers in these states. This is a different problem.

Structures that Support Group-to-Group Relations

The G-to-G relation can be established between any two files that have a property in common. It doesn't require any special fields or one-many relations. In fact it doesn't require there be any relations at all!

As an example consider a BUYERS and a SELLERS file and say that records in both files specify the STATE they're located in. This alone is enough for us to ask: "Given a diverse group of BUYERS, what group of SELLERS are located in the same states?"

We can represent this question as:

[Buyers] -----> states <-----[Sellers]

The representation emphasises an important difference between related groups and related records, namely the group relation has no "direction." It doesn't matter which group you know first, the *process* of finding the members of the two groups is symmetrical.

The relation between groups is not a formal "many-to-many" relation as represented by,

[Cars] -----> cars / parts <---- [Parts]

and illustrated as in Figure 3.

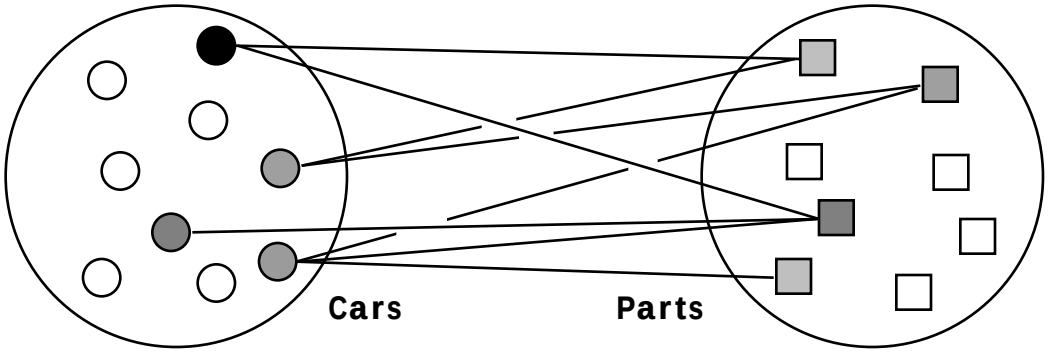

Figure 3: Cars and Parts records in a many-to-many relation.

The term many-to-many describes a structural relation requiring an auxiliary file that records each pairing of records in the related files. This auxiliary file is shown in Figure 4. The CarsAndParts file contains records that store the ID of a car along with the ID of a part. If many cars have the same part there will be many CarsAndParts records with different car ID's and the same part ID In a many-to-many relation the relations are specific to, and separately maintained for each record.

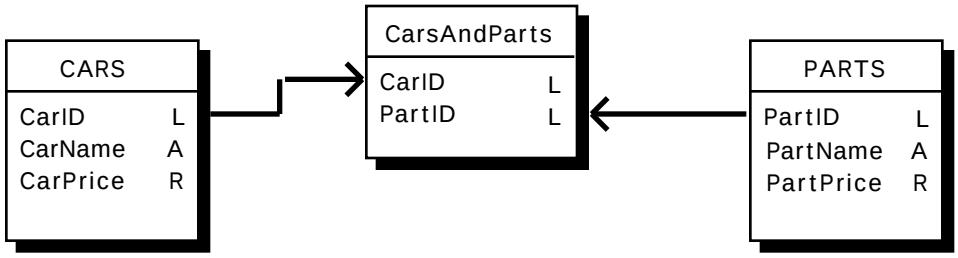

Figure 4: Cars and Parts files in a many-to-many file relation.

In contrast G-to-G relations can exist between files that are also related one to many, one to one, many to many or are not formally related at all. Knowing that a G-to-G relation exists tells us almost nothing about the file structure and this is the reason that G-to-G is so important: it is a simple relationship supported by all file structures.

Effecting the Group-to-Group Relationship

We want a tool to establish G-to-G relationships that is simple to use and that requires no knowledge as to how it operates on the part of the programmer. Our tool will have to specify the two groups that are related and the properties of the first group that are to be found in members of the second group. This means we need to tell 4D which two files are being related and the fields that store the properties that relate the two files.

We can do this by using pointers to the fields in two files that carry the relating data. The pointers tell us the fields that we need to compare; these fields might store, for example, a two-character state code. If the first field pointer is associated with the file whose current selection is the source group then we've specified all we need: we know where to look (the field pointed to by the second pointer) and what to look for (all values of the current selection in the field pointed to by first pointer).

In our example we'll set the first pointer to the State field of the Buyers file and the second pointer to the State field of the Sellers file. If we're relating some group of Sellers to a known group of Buyers we'll limit the current selection of the Buyers file to those Buyers that we want to relate from. We'll then scan the values in the Seller's State field looking for Sellers in the Buyer's states.

The tool I'll construct will be a two-argument procedure called *GROUPtoGROUP*(a;b), where "a" points to the field of group A that is to be matched with the field pointed to by "b." In our source code we will call group-to-group procedure as:

GROUPtoGROUP(»[BUYERS]state;»[SELLERS]state).

If we knew the group of Sellers first and had to find the group of Buyers we'd reverse the order of the arguements.

As an example of the flexibility of G-to-G relations we'll construct a G-to-G relation that applies to the Mothers and Sons structure. First we need to know the quantity that relates them. A common sense criterion might be their last names but this might not work if

there are several mothers with the same last name. Instead we'll store a unique value in a field called "M_key" that distinguishes one mother record from another. The file structure that would appear in the structure editor is:

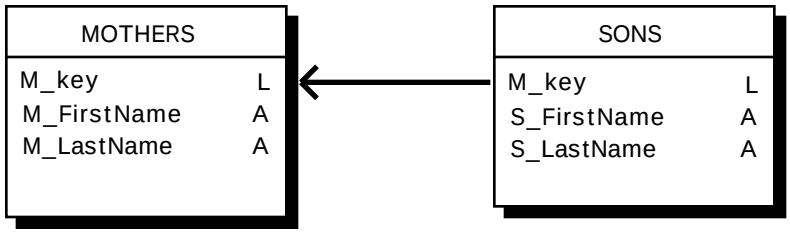

Figure 5: Mother and son files in a one-to-many file relation.

To operate as intended we make sure that the current selection of Mothers records is the group we want to relate from and then we call the G-to-G procedure as:

GROUPtoGROUP(»[Mothers]M_key;»[Sons]M_key)

Inside the Black Box

I'll give the simplest realization of a G-to-G procedure. More complicated versions using arrays and built searchs can be written which may run up to 50% faster.

The G-to-G procedure shown below uses the current selection in the first file pointed to as the first group and gathers the related records in the second group by adding them to a set. The procedure performs repetitive searches on the field pointed to by the second argument. The form of the call is:

GROUPtoGROUP(»[File1]fieldX;»[File2]fieldY)

In the following procedure code the two local variables, $1 and $2, are of pointer type.

```
`core code:
$1stFilePtr := File(File($1))
$2ndFilePtr := File(File($2))
CREATE EMPTY SET($2ndFilePtr»;"RelatedGroup")
FIRST RECORD($1stFilePtr»)
For($j;1;Records in selection($1stFilePtr»))
  SEARCH($2ndFilePtr»;$2»=$1»)
  CREATE SET($2ndFilePtr»;"RelatedSubgroup")
  UNION("RelatedGroup";"RelatedSubgroup";"RelatedSubgroup")
  NEXT RECORD($1stFilePtr»)
End For
USE SET("RelatedGroup")
```

This searches the related file (pointed to by $2ndFilePtr) for records that match the criterion in the current record of the relating file (that pointed to by $1stFilePtr). It stores the related records that it locates in the "RelatedSubgroup" set. At each iteration of the

For loop it finds records related to the next current record in the relating file and adds these to the “RelatedGroup” set.

Some relating records in the relating file may have the same relating values and the search on these values will locate the same related records. When this occurs these cycles of the loop will be redundant, however duplicates will not appear in the final group of related records because we’re using sets. When the procedure finishes the For loop the “RelatedGroup” set has all the related records and the **USE SET** command makes these records the current selection of the related file.

In addition to the core code we should type the variables before using them and dispose of the set to free memory after we’re finished. To complete the GROUPtoGROUP code we add the following:

```
`GROUPtoGROUP
`$1=pointer to field in relating file,
`$2=pointer to field in related file,
`Requires relating records to be current selection of relating file,
`Results in related group as the current selection of the related file.
C_POINTER($1;$2)
C_LONGINT($j)
`----------------
   { core code }
`----------------
CLEAR SET(“RelatedGroup“)
CLEAR SET(“RelatedSubgroup“)
```

To establish the relation in the opposite direction you simply switch the argument’s order of appearance. This symmetry means that either field may be searched. If you expect you might be searching in either direction then both fields should be indexed for faster searching.

This G-to-G procedure satisfies all our requirements. It’s simple and foolproof. As long as the two pointers that we pass relate to the correct fields the procedure will produce a current selection in file 2 that’s related to that in file 1.

The G-to-G concept is tremendously useful and packaging it into a simple procedure enables you to program and think more broadly. It can expand both what your program does and what you allow your users to do.