Using SQL on Nonrelational Databases

Barbara A. Barrett, SAS Institute Inc., Austin, TX
Jim Craig, SAS Institute Inc., Austin, TX

ABSTRACT

Structured Query Language (SQL) is a powerful tool for relational database query expression. That is what it was designed for. But many developers want to use SQL for nonrelational databases. Special interface software between the SQL code and the database is required because SQL has no knowledge of hidden information in the database. This paper describes, in SQL terms, some issues that can arise when SQL queries are made against a nonrelational database.

INTRODUCTION

SQL is a language for dealing with the relational data model. Many hardware and software vendors have developed SQL dialects, that is, implementations of SQL for their computers and software products. In 1986, SQL was ratified as an official standard by the American National Standards Institute (ANSI). Potential benefits to application developers include more portable applications, distributed applications, and reduced programmer training costs through standardization.

Nonrelational Systems Live On

For many reasons, there are still applications today that use data stored in nonrelational databases, such as hierarchies and networks. Perhaps they were written in prerelational days. Perhaps there was no relational database management system (DBMS) available to the developer. Perhaps the nonrelational system just fits the application better. Theoretically, although less flexible, networks and hierarchies can reduce storage and retrieval costs in applications with a lot of repetitious data. The ability to express relational queries against nonrelational databases is still desirable for the benefits listed above.

If you are an SQL application developer, you may use a vendor's SQL front end for a nonrelational database. The specific dialect of SQL that you use may vary from product to product. The implementation of that dialect also may vary. Two common alternatives are

- preprocess the input SQL query into one or more statements of the native DBMS Data Manipulation Language (DML)
- accept the input SQL query as is, and then surface output to it as though the underlying data were relational.

Behind the Scenes

As a developer, you expect that your query, however handled, accurately taps the stored information or accurately updates it without unwanted side effects.

In Version 6 of the SAS® System, applications can be written with the new SQL procedure. The database interface software will be provided by the SAS/ACCESS® database interface products. At SAS Institute, the developers of these database interfaces have been studying the best ways to materialize nonrelational databases in a relational way. This is an ongoing effort.

There are some tough issues involved. It has been helpful to describe the behind-the-scenes processing in SQL terms, as if the data were relational after all. If you are an SQL developer but not a database expert, this analysis should help you understand what you can expect to happen when you write an SQL application for nonrelational data. This paper identifies potential pitfalls that any interface software faces. You can evaluate the behavior of your interface in these areas.

An Example

Suppose you work for a hypothetical software company whose users gather annually to share information and ideas about the products. Being an avid follower of these conferences, you keep a database about the attendees. There is a set of information for any person who has ever attended. Here is a small sample from your general information file.

```
NAME            ADDRESS
------------    ---------------
Adams           New York
Baker           Chicago
Cox             Dallas
```

In addition, because attendance can vary year to year, you keep another file with records by year for each person.

```
( NAME        ) ATTENDED IN YEAR    COMPANY AFFILIATION
  ----------    ----------------    -------------------
( Adams       ) 1980                University
( Adams       ) 1985                ABC Inc.
( Baker       ) 1984                DEF Inc.
( Cox         ) 1987                GHI Inc.
```

Each year, a person may present a paper, lead a discussion group, or both, or none. A third file contains this information.

```
( NAME        ) (ATTENDED IN YEAR)  ACTIVITY          TITLE
  ----------    ----------------    ---------         -----
( Adams       ) ( 1980 )            Paper             How To..
( Baker       ) ( 1984 )            Round Table       When To..
( Baker       ) ( 1984 )            Birds of a Feather What To..
```

Note that Adams had no special activity in 1985, nor did Cox in 1987.

Finally, you keep a fourth file that is a running list of the backgrounds and interests of each person.

```
( NAME        ) INTEREST            LEVEL
  ----------    ----------------    ---------
( Adams       ) End User            Strong
( Adams       ) Graphics            Casual
( Baker       ) Programmer          Strong
( Baker       ) Statistics          Medium
( Baker       ) Graphics            Strong
( Cox         ) Programmer          Strong
( Cox         ) Databases           Medium
( Cox         ) Graphics            Casual
```

The four files are named GEN, ATT, ACT, and INT.

How Your Database Is Seen

Notice that some of the columns are enclosed in parentheses. What does this mean?

In SQL terms, each of the four files above is a base table, meaning that the data are stored that way. In a nonrelational system, the four tables are not stored as shown. In particular, the data in parentheses are not stored. Instead, internal pointers represent the connections that are implied between the tables. The DBMS software maintains the pointers.

In SQL, if you want to connect two tables, you create a join view (also called a derived table).

```
CREATE VIEW PEOPLE AS
  SELECT GEN.NAME, GEN.ADDRESS, INT.INTEREST, INT.LEVEL
  FROM GEN, INT WHERE GEN.NAME = INT.NAME;
```

This view guarantees that you get the right names with the right interests. The NAME fields in both tables contain essential information to tie the two tables together. In relational terms, GEN.NAME is the primary key of GEN, and INT.NAME is the foreign key of INT. A primary key uniquely identifies one record. A foreign key is a field whose values only take on the values of a primary key in some other table.

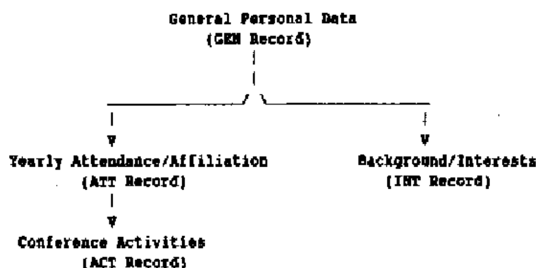
Hierarchies and networks already know which GEN records go with which INT records. The DBMS pointers take the place of the primary and foreign keys. But the join is *frozen*; that is, it is part of the database structure. In SQL terms, a nonrelational schema is a frozen set of join views. The relationships that primary and foreign keys provide in SQL are represented in nonrelational schemas by pointers instead. Because you do not see the pointers, these schemas are said to contain hidden essential information.

Apples and Oranges?

Hierarchies and networks are alike in that both hide essential information about relationships. They do so to minimize the number of times a given item of data physically must occur. Generally, the performance of the software when it processes internal pointers is very fast. In a relational database, there are never any hidden pointers connecting records, though it is still possible to minimize redundancy through a discipline called normalization.

Normalization rules help you arrive at the best arrangement of relational tables. You attempt to minimize redundancy by rearranging a table into multiple smaller tables. You carry field(s) in each table with enough information to join tables back together properly. But some redundancy is inevitable. Hierarchies and networks also can contain redundant data. But they can completely eliminate the redundant data supporting relationships, though at the cost and added complexity of hidden pointers.

Hierarchies are simpler than networks because fewer kinds of relationships are allowed. A hierarchy can be easily represented by the following tree diagram.



A network is more complex. It supports more kinds of relationships by allowing a record to have a pointer to it from more than one other kind of record. It also allows a record to have nothing pointing to it. For example, your database may have a paper with two authors or an interest field with no one interested in it.

Think of the relationships in a network as a collection of many small hierarchies, often called sets. Unlike a hierarchy, the records in a network can have many relationships or none. It is more difficult to draw a picture of the potential connections.

Hierarchies and networks share many of the same problems for relational access. But the issues are usually magnified in a network because the potential for numerous relationships creates a potential for much more hidden essential information. To simplify the examples, assume the sample database is a hierarchy.

Why Problems Arise

There are problems in surfacing nonrelational data to a relational language like SQL because

- nonrelational models consist of sets of records in which there is hidden information; relational models do not.
- nonrelational models contain repeating groups of varying dimension; relational models do not.

The rest of this paper describes some interesting mapping issues and explains in SQL terms how you should expect your application to behave.

When You Qualify with Repeating Values

When you code an SQL query like

```
SELECT NAME FROM PEOPLE
  WHERE INTEREST = 'Graphics';
```

the DBMS can answer it fairly easily. That is because the SQL emulation is not very difficult. A list of related, or joined, GEN and INT records is derived using indexes and relationship pointers.

But suppose you want the names of people interested in both Graphics and End Users. Then either you or the DBMS must go to some extra trouble. Consider the following view PEOPLE:

VIEW PEOPLE

NAME	ADDRESS	INTEREST	LEVEL
Adams	New York	End User	Strong
Adams	New York	Graphics	Casual
Baker	Chicago	Programmer	Strong
...	...	...	...

Note that the view PEOPLE has no single row that meets the condition

```
WHERE INTEREST = 'Graphics' AND INTEREST = 'End User'
```

Using SQL, you need to code a reflexive join to handle repeating values properly. This operation joins a view to itself, in this case PEOPLE to PEOPLE, producing a crossproduct from which the WHERE clause can be evaluated.

```
SELECT DISTINCT * FROM PEOPLE FIRST, PEOPLE SECOND
  WHERE FIRST.NAME = SECOND.NAME AND
     FIRST.INTEREST = 'End User' AND SECOND.INTEREST = 'Graphics';
```

FIRST.PEOPLE				SECOND.PEOPLE			
NAME	ADDRESS	INTEREST	LEVEL	NAME	ADDRESS	INTEREST	LEVEL
Adams	New York	End User	Strong	Adams	New York	Graphics	Casual

Your DBMS may not have to do a reflexive join to satisfy the query. It may be able to take advantage of special nonrelational operators to materialize the same data. But the result should be the same as if a reflexive join were done.

When Values Are Missing

In most databases, it is possible to insert records without providing a value for every field. The DBMS chooses a value for any missing field, for example blank for character fields and zero for numeric fields. Some DBMSs select a special value, called a null value, because they think it is important to distinguish between data you set to zero or blank, and data you did not provide. Unfortunately, this is one area where DBMSs tend to differ a lot.

SQL supports 'IS NULL' and 'IS NOT NULL' syntax in the WHERE clause. If you use it, then you should look carefully at how your DBMS interface supports missing values. If your DBMS does not distinguish nulls from other values, then your result may contain rows in which the values are not really null but happen to have the same value as null, such as zero.

Other missing-value anomalies result when you use negation operators, such as NOT or NE. Insert new GEN and ATT records for Dixon, a consultant who attended the conference in 1988. Then insert new INT records as follows:

(NAME	) INTEREST	LEVEL
(Dixon	) End User	Strong
(Dixon	) Databases	<missing>

Dixon did not indicate an interest level for Databases. To find the names of people whose interest in databases is anything but casual, you ask

```
SELECT NAME FROM PEOPLE
  WHERE INTEREST = 'Databases' AND LEVEL NE 'Casual';
```

Your result may or may not include Dixon depending on how your DBMS handles missing values. The distinction may be important to you. A potential twist can arise if your DBMS does not include null values in its indexes. You may determine that your query gives you Dixon in the result, only to have that negated because the index neglected to include nulls.

When Records Are Missing

Just as values can be missing, so can records. For example, a person may attend a conference but not present a paper. Note that Baker attended the conference in 1985 but did not lead an activity. Let us create a new view so we can talk about this case.

```
CREATE VIEW HISTORY AS
  SELECT GEN.NAME, GEN.ADDRESS, ATT.YEAR, ATT.COMPANY, ACT.ACTIVITY
  FROM GEN, ATT, ACT
  WHERE GEN.NAME = ATT.NAME AND GEN.NAME = ACT.NAME
    AND ATT.YEAR = ACT.YEAR;
```

A given occurrence of a hierarchical structure is said to be *incomplete* if it has a record that could have children but does not. In a network, a set with absent related records is called an empty set. These are quite common. In the sample database, there are several examples of missing records. The structures for Cox and Dixon are incomplete because there are no ACT records. The structure for Adams is incomplete because the 1985 ATT record has no corresponding ACT record.

The analogy in SQL for an incomplete structure or empty set is a row in one table with no match in the other. In a standard SQL implementation, these mismatched rows are not present in the HISTORY view because they fail the test GEN.NAME = ACT.NAME.

STANDARD VIEW HISTORY

NAME	ADDRESS	YEAR	COMPANY	ACTIVITY
Adams	New York	1980	University	Paper
Baker	Chicago	1984	DEF Inc.	Round Table
Baker	Chicago	1984	DEF Inc.	Birds of a Feather

Consider the query, "Tell me the names of people who attended the conference in every year except 1980," expressed as

```
SELECT NAME FROM HISTORY
  WHERE YEAR NE 1980;
```

The standard SQL answer would be Baker. But this seems incorrect. You know there are more people who attended before and after 1980 than just Baker.

Some SQL dialects, such as that of the SQL procedure in the Version 6 SAS System, provide a way of dealing with missing records. In PROC SQL, a special kind of join is provided, called an outer join. The outer join identifies a LEFT or RIGHT table to identify whether mismatched rows from either table should be included in the result.

```
CREATE VIEW HISTORY AS
  SELECT ATT.NAME, ATT.YEAR, ATT.COMPANY, ACT.ACTIVITY
  FROM ATT LEFT JOIN ACT
  ON ATT.NAME = ACT.NAME AND ATT.YEAR = ACT.YEAR;
```

VIEW HISTORY USING OUTER JOIN

NAME	YEAR	COMPANY	ACTIVITY
Adams	1980	University	Paper
Baker	1984	DEF Inc.	Round Table
Baker	1984	DEF Inc.	Birds of a Feather
Adams	1985	ABC Inc.	<missing>
Cox	1987	GHI Inc.	<missing>
Dixon	1988	Consultant	<missing>

Now the query produces the expected result.

Another outer join operator, FULL, includes mismatched rows from both tables. This is useful in joining networks with empty sets, because records on either side of the join can be missing. However, precaution is advised in systems whose WHERE clauses equate missing values with other values. Otherwise, you may wind up joining the wrong records, for example people who never gave papers with people having no interests. One solution is to assign special missing values by table yourself, before attempting the join. PROC SQL in the SAS System provides a special option on the outer join that helps prevent unwanted matches.

When Records Are Not Directly Related

Not every record in a hierarchy or network is directly related to every other record. Some are not related; others are only indirectly related. For example, there is nothing in the ATT record that ties it to an INT record. ATT and INT are said to be disjoint. In a network, a set with disjoint records is called a multimember set.

Dealing with disjoint records in SQL can be a difficult problem for a nonrelational DBMS. Since the records are not related there are no internal DBMS pointers to tie the records together in any meaningful way. (And, since the schema is nonrelational, there are no redundant values to use either.) A useful class of applications to study for this issue is the various DBMS report writers. Report writers commonly present unrelated records in compact tabular rows.

Suppose you want to produce a data sheet for each person that summarizes attendance and interest information. In SQL, the presence of redundant data lets you do this fairly easily. Create a report view as follows:

```
CREATE VIEW SHEET AS
   SELECT GEN.NAME, ATT.YEAR, INT.INTEREST
   FROM GEN, ATT, INT
   WHERE GEN.NAME = ATT.NAME AND GEN.NAME = INT.NAME;
```

The view SHEET looks very awkward for a report. There are too many rows.

```
NAME     YEAR   INTEREST
------   ----   ----------
Adams    1980   End User
Adams    1980   Graphics
Adams    1985   End User
Adams    1985   Graphics
Baker    1984   Programmer
 ...      ...     ...
```

The report is more readable if rows 2 and 3 are eliminated.

```
NAME     YEAR   INTEREST
------   ----   ----------
Adams    1980   End User
Adams    1985   Graphics
Baker    1984   Programmer
 ...      ...     ...
```

This more compact form shows an arbitrary association of records according to ordinal information in the database. That is, the first ATT is shown next to the first INT, and so on.

Your DBMS SQL interface may not even allow a view to refer to disjoint records or multimember sets. If it does, many alternatives can be imagined, none perfect.

1. Produce a fully relational, fully redundant result. Through postprocessing, handle the compaction yourself.

2. Accept an arbitrary compaction from the DBMS. Some WHERE clauses that work in the fully redundant result may not work in the compact one because the rows are different.

3. Use ordinal information from the DBMS, if available, in your query.

```
CREATE VIEW SHEET AS
  SELECT GEN.NAME, ATT.YEAR, INT.INTEREST
  FROM GEN, ATT, INT
  WHERE GEN.NAME = ATT.NAME AND GEN.NAME = INT.NAME
      AND ATT.ORD = INT.ORD;
```

When You Need to Do Updates

In the relational model, updates are a special topic. There are many ways to get into trouble, especially when dealing with join views. Many SQL products disallow join view updates, as does the current ANSI standard. What makes updates so problematic?

The main issue is the question: should an update be allowed to produce side effects? That is, should updating a row ever cause another row to change simultaneously? Relational gurus say no. But people used to dealing with data presented by existing nonrelational applications are used to updating what they see, regardless of where it lives. In many cases, the side effects are understood and even wanted. This is especially true in interactive applications, where someone witnesses the side effect. Side effects may be less permissable in a batch application.

An example of a desirable side effect can be seen with the view PEOPLE. Suppose Adams calls you on the phone. You pull up an application screen showing row 1 of PEOPLE. He tells you he has just moved from New York to Albany, so you key in the update. A side effect occurs: row 2 changes also! This does not bother you. You know that because there is really only one copy of Adams in the hierarchy, the DBMS is doing you a fevor by not requiring you to update every Adams row.

Although relationally invalid, this may be an example of a desirable side effect. What about an undesirable one? Consider the following SQL update command:

```
INSERT INTO PEOPLE (NAME, ADDRESS, INTEREST, LEVEL)
  VALUES ('Cox', 'Houston', 'Languages', 'Medium')
```

PEOPLE already has several Cox rows.

```
NAME   ADDRESS    INTEREST     LEVEL
----   --------   ----------   --------
Cox    Dallas     Programmer   Strong
Cox    Dallas     Databases    Medium
Cox    Dallas     Graphics     Casual
```

How do you want this insert to be handled in the hierarchy? Do you mean to add another person named Cox who lives in Houston instead of Dallas? Or has Cox moved, and all you really want to do is add another interest? Odds are that whichever way you go someone will want it the other way. The potential for side effects is much greater in networks because much more essential information is hidden.

A Rule of Thumb for Updates

There are many other problems with updates beyond the scope of this paper. The flexibility of your SQL interface product determines how many of them you must understand and deal with. An easy rule of thumb to avoid problems is: if you are allowed to update join views, then restrict your updates to the file on the foreign key side of the join. In a hierarchy, this is the bottom-most record in your tree. While more restrictive than nonrelational users are accustomed to, this rule keeps you out of trouble. Then, as you understand your environment more, you can experiment with the side effects your implementation allows.

When SQL Is Smarter Than Your DBMS

This paper has dwelled on those areas in which the features of the nonrelational DBMS overwhelm the intended uses of SQL. There are other situations in which SQL capabilities overwhelm the DBMS capabilities. Examples of this occur with the WHERE clause. For example, your nonrelational DBMS may not support

pattern matching, built-in functions, ordering, or qualification by group, all handy SQL features. Records may need to be post-processed after the DBMS returns them to evaluate conditions the DBMS cannot handle. Version 6 of the SAS System provides for WHERE postprocessing automatically.

Postprocessing can be helpful, but to work properly some rules or extra information may be required. Postprocessing executes in a layer of software normally ignorant of hidden information. If your WHERE clause conditions refer to fields in different files, then you are exercising this hidden information. Two ways that the SQL interface software can help are listed below.

1. Forego any row compaction processing, such as described under "When Records Are Not Directly Related." Redundant results are the only way SQL can evaluate some conditions properly.

2. Provide DBMS internal information as additional variables in the view, such as ordinal variables or record numbers.

CONCLUSION

These studies identify mapping problems common to SQL dialects for nonrelational databases. Most of the specialized capabilities of nonrelational databases can be reasonably represented in SQL terms. Some extensions to standard SQL make the job easier. Likewise, some restrictions may be necessary to standardize the behavior of the interface software. Developers at SAS Institute are examining these issues early to serve future users of PROC SQL and the Version 6 SAS/ACCESS database interfaces better.

SAS and SAS/ACCESS are registered trademarks of SAS Institute, Cary, NC, USA.