Foreign and Primary Keys in RDM Embedded SQL: Efficiently Implemented Using the Network Model

By Randy Merilatt, Chief Architect - January 2012

This article is relative to the following versions of RDM:

- ✓ RDM Embedded 10.1

Network model databases have been viewed as strictly non-relational and outdated. But in the 1990s when the SQL standard incorporated support for declared foreign keys to represent inter-table relationships, the gap between the relational and the network model representation of a database narrowed. This was exploited by Raima in RDM Server (RDMs) with the introduction of the ***create join*** *statement that allowed an SQL foreign and primary key relationship to be explicitly mapped into a network model set. in 2009, when Raima decided to add SQL to RDM Embedded (RDMe) we decided to keep RDMe SQL as simple and as free of as many unnecessary, non-standard extensions as possible. Hence, network model sets are automatically (i.e., implicitly) used in the implementation of foreign and primary key relationships. This paper describes how network model sets are implemented in the RDMe core-level database engine, the syntax and semantics of SQL foreign and primary keys, and why the use of network model sets provides an efficient and high performance implementation for them.*

The Network Database Model: A Little History

The primary data organization mechanism that was provided in some very early database management systems (DBMS) was the ability to declare a one-to-many relationship between a parent record type (table) and a child record type. These systems were known as hierarchical model systems because the data was strictly hierarchically arranged—each child could have only one parent. The DBMS provided access methods where each child instance for a particular parent record instance could easily and quickly be accessed. Later database systems relaxed the strict hierarchical scheme so that a child could belong to more than one parent and were called network model systems. A standard database language came to be defined by a standards body known as CODASYL ("Conference of Data Systems Languages") that specified how a network model database could be defined (through a DDL—Data Definition Language—specification) and accessed and manipulated (through a DML—Data Manipulation Language—specification) designed to work with COBOL programs. The one-to-many parent-child inter-record type relationships in a network model database were specified by what is called a **set**. The example below shows the DDL for a CODASYL DBMS (Univac DMS 1100 circa 1973)[*] which defines two record types, author and book, and a set that defines the one-to-many relationship between the two.

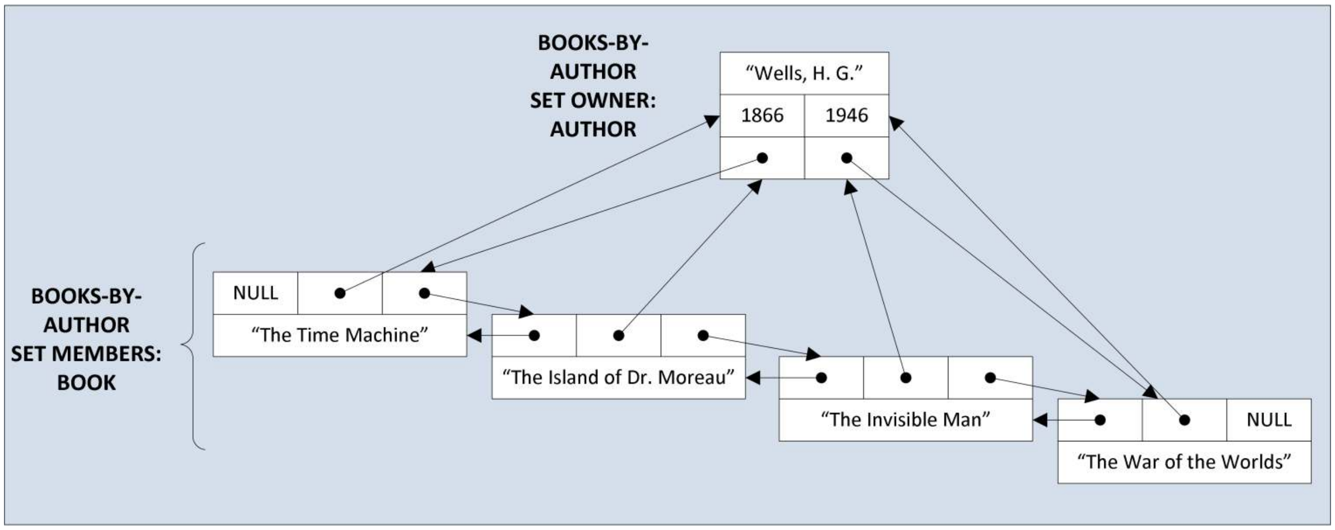
[*] Currently called DMS 2200 which Unisys continues to maintain and support.

Figure 1. Example of a CODASYL DDL Specification of a Parent and Child Record and a Set:

```
RECORD NAME IS AUTHOR
   RECORD CODE IS 1
   LOCATION MODE IS CALC NAMEHASH
      USING NAME
      DUPLICATES ARE NOT ALLOWED
   02 NAME PIC X(35)
   02 YEAR-OF-BIRTH PIC 9999
   02 YEAR-OF-DEATH PIC 9999
RECORD NAME IS BOOK
   RECORD CODE IS 2
   LOCATION MODE IS VIA BOOKS-BY-AUTHOR SET
   02 TITLE PIC X(60)
   02 YEAR-PUBLISHED PIC 9999
SET BOOKS-BY-AUTHOR
   MODE IS CHAIN LINKED PRIOR
   ORDER IS LAST
   OWNER IS AUTHOR
   MEMBER IS BOOK AUTOMATIC LINKED TO OWNER
      SET OCCURRENCE SELECTION IS THRU LOCATION
         MODE OF OWNER
```

Retrieving data is done in a COBOL DML program by navigating through sets to locate the desired records. For example, first a FETCH for a specific AUTHOR record occurrence would be executed followed by a FETCH of each BOOK record that is a member of the BOOKS-BY-AUTHOR set for which the fetched AUTHOR was the owner. Thus, multiple statements are executed in order to retrieve the desired data. In the example above, the BOOK occurrences that are all members of the same set are directly related together through a doubly linked list—no index is needed—providing optimal (i.e., maximum of 1 logical read) access. The following diagram shows how a typical BOOKS-BY-AUTHOR set instance may be stored in the database.

Figure 2. Example Set Instance Implementation:


BOOKS-BY-AUTHOR SET OWNER: AUTHOR
"Wells, H. G."
1866
1946
BOOKS-BY-AUTHOR SET MEMBERS: BOOK
NULL
"The Time Machine"
"The Island of Dr. Moreau"
"The Invisible Man"
NULL
"The War of the Worlds"

The DBMS maintains in the owner record pointers to the first and last member records in the linked list and for each member pointers to the prior and next member along with a pointer to the owner. These "pointers" are called database keys (or database addresses) and directly map into the actual location where the record is permanently stored on disk.

When a new book is stored along with its title (and whatever other data fields are included in the record's DDL declaration) the author name will also be provided. However, it is not actually stored with the book record but will be used to find the author record with that name so that when the book is stored it will be connected to the correct BOOKS-BY-AUTHOR set instance. In this way the author name functions like the primary key in the AUTHOR record type and a foreign key in the BOOK record type even though no such data field is declared in BOOK.

RDMe Core-Level DDL Set Declarations

From its original incarnation in 1983 as db_VISTA, RDM Embedded has been a network model DBMS but one that was designed for use with the C programming language rather than COBOL. Figure 3 below gives the RDMe Core (i.e. low-level) DDL specification corresponding to that given in Figure 1.

Figure 3. RDMe Core DDL Specification of Parent and Child Record and a Set:

```
record author {
   unique key char name[36];
   short  year_of_birth;
   short  year_of_death;
}
record book {
   char   title[61];
   short  year_published;
}
set books_by_author {
   order last;
   owner author;
   member book;
}
```

Data retrieval is performed by the C program by making calls to the RDMe Core-level C API functions that lookup records based on a specified key value (e.g., the author name) and then locating the related book records by calling a function that retrieves the members of the set. So, like CODASYL. retrieval is navigational.

Declared Foreign and Primary Keys in SQL

One-to-many relationships between tables (= record types) in SQL are explicitly defined in the **create table** statement through primary and foreign key column declarations. A *primary key* is a column (or columns) in a (parent) table which is used to uniquely identify each row (= record occurrence) of the table. A *foreign key* is a column (or columns) in a (child) table that is used to identify the primary key value of the related row from the parent table. Foreign key column(s) must be declared to have the identical data type as its referenced primary key column(s). The SQL DDL that corresponds to the DDL given earlier in and Figure 3 is shown in Figure 4 below.

Figure 4. SQL DDL for Parent and Child Tables:

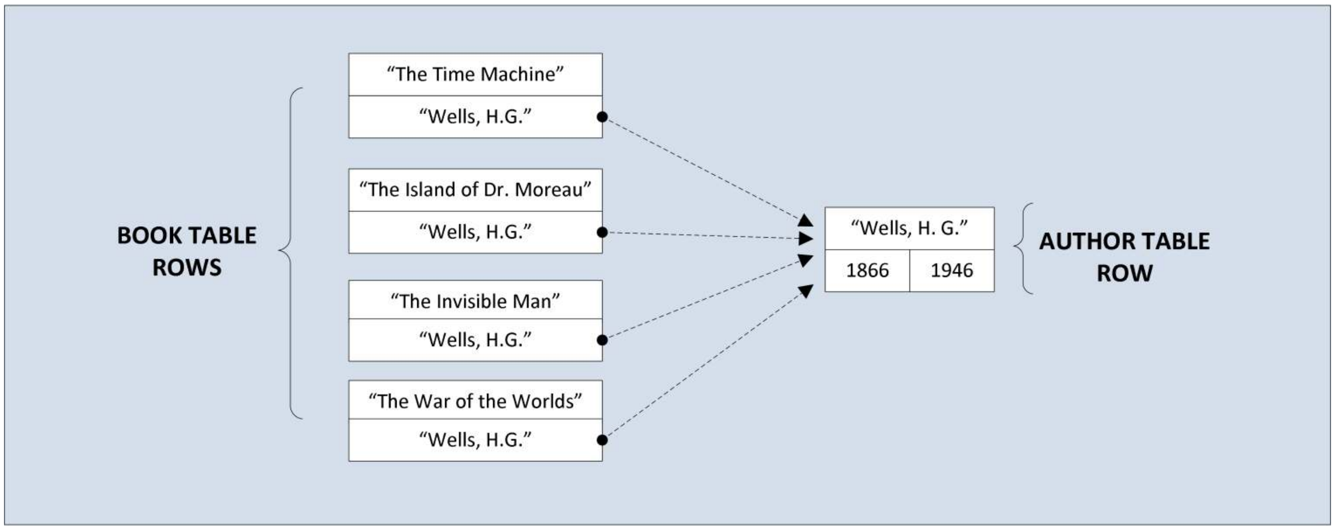
```
create table author
(
   name              char(35) primary key,
   year_of_birth     smallint,
   year_of_death     smallint
);
create table book
(
   title             char(60),
   year_published    smallint,
   name               char(35) not null references author(name)
      on delete cascade on update cascade
);
```

The foreign key is declared on the name column in the book table with the **references** clause. The **on delete cascade** and **on update cascade** clauses specify what is to happen when a row of the author table is deleted or updated. When **cascade** is specified then on a **delete** all of the related book rows will automatically be deleted as well. On an **update** of the author's primary key column (e.g., fixing a misspelled author name) all of the related book rows will automatically updated. In a standard relational DBMS, in order for these operations to be performed in a reasonable amount of time the name column in the book table would need to be indexed (something that the DBMS might automatically create based on the **cascade** specification).

Explicit declaration of foreign keys allows the DBMS to enforce *referential integrity* in which the system ensures that all rows from the primary key table that are referenced by foreign keys always exist. For example, besides the **cascade** option, the **on** clause can specify **restrict** which means that the **update** or **delete** on the author row is not allowed when a book row that references it exists.

Retrieval is performed through an SQL **select** statement that specifies a *join* between the two tables that conditionally equates the two tables through the foreign and primary keys. The diagram below shows how the rows from the two tables are related through the common foreign and primary key column values.

Figure 5. Relational Database Representation of a Parent-Child Table Join:



The dotted-line arrows represent the primary key row being referenced by the foreign key column values—they are not actually pointers. The **select** statement that can be used to retrieve the data is shown below along with its *result set*.

Figure 6. SELECT Statement to Retrieve Author-Book Data:

```
SELECT name, title FROM author natural join book where author.name = "Wells, H. G."

name            title
Wells, H. G.    The Time Machine
Wells, H. G.    The Island of Dr. Moreau
Wells, H. G.    The Invisible Man
Wells, H. G.    The War of the Worlds
```

Foreign Key Implementation with Sets

An SQL DBMS needs to be able to efficiently manage all of the properties associated with foreign key declarations such as:

1. fast access from the primary key row to all rows with matching foreign key values,
2. fast access from the foreign key row to the referenced primary key row,
3. ability to quickly check to ensure a referenced primary key row exists,
4. ability to quickly determine if a primary key row has references

Relational database systems usually do this by creating a unique index on the primary key column values and a non-unique index on the foreign key column values. By having indexes on both the primary and foreign keys all four of the above requirements can be supported.

The cost, however, comes in the amount of redundant data introduced by the duplicate foreign key column values in which not only is each stored in a row of the foreign key table but also in the index. Moreover, the cost of accessing a foreign key row using the index, while usually acceptable, may still involve 2 or 3 additional disk accesses.

Foreign and primary key relationships can easily and efficiently be implemented using network model sets and that is exactly what RDM Embedded SQL does.

By using sets, all of the rows of the foreign key table that reference the same primary key row are maintained in a link list much as is shown in Figure 2. This means that the related rows can be retrieved by direct access (guaranteed to require no more than a single disk read).

Use of sets to implement the foreign and primary key relationships also means that the foreign key columns can be *virtual*—that is, their values do not actually need to be stored in the foreign key table as those values can be easily retrieved from the referenced row through the set's owner pointer. That, coupled with the fact that the foreign keys do not need to be indexed, can significantly reduce the amount of redundant data. It also means that the work needed to be done in order to process a cascade update (where the foreign keys are all instantly updated whenever the primary key value changes) and delete (the related rows to be deleted are all directly connected to the referenced row) is minimized.

So, by using sets to implement SQL foreign and primary key relationships, RDM Embedded SQL not only provides optimal performance but does so with less storage requirements than needed in a relational-only implementation.

Relational Model Versus Network Model

The differences between relational and network model databases are often characterized as being mutually exclusive. But as shown above, from a database organization standpoint, they are compatible because a network model implementation using sets can efficiently be used to implement relational model foreign and primary key relationships.

However, the real difference between the two models is in how the data is *accessed* not in how it is *defined* (or implemented).

Access to database data in a network model database requires that the application program navigate through set occurrences, one member record occurrence (i.e. row) at a time to locate the desired record occurrence(s). Using a relational database model language (e.g., SQL), a single **select** statement will perform the needed navigation and, thus, is much simpler. The advantages of the relational model are most significant in performing queries. For storing, modifying, deleting and retrieving individual record occurrences (table rows), however, there is no significant advantage. But only with RDM can you choose which approach is best for your application requirements.

Want to know more?

Please call us to discuss your database needs or email us at info@raima.com. You may also visit our website for the latest news, product downloads and documentation:

www.raima.com