Craig S. Mullins

Return to Home Page

October 1998

SQL Server update

Referential Integrity in Microsoft SQL Server

By Craig S. Mullins

This article analyzes the various aspects that should be considered when implementing referential integrity (RI) in a Microsoft SQL Server database environment. RI is examined from a conceptual level first, and then from a practical, implementation oriented viewpoint. The DDL necessary to create tables using declarative RI is discussed as is using triggers to implement complete referential integrity.

After reading this article the reader will:

- know what referential integrity is and how to implement referential integrity between logically related tables
- be able to describe the intent of each RI rule (insert, update, delete) and its various options
- be able to code efficient SQL data definition language (DDL) **create table** statements using declarative RI constraints
- code efficient triggers to implement all aspects of RI

What Is Referential Integrity?

Referential integrity is a method for ensuring the "correctness" of data within a DBMS. Many people tend to over-simplify RI stating that it is merely the identification of relationships between relational tables. It is actually much more than this. Of course, the identification of the primary and foreign keys that constitutes a relationship between tables is a component of defining referential integrity.

RI appropriately embodies the integrity and usability of a relationship by establishing rules that govern that relationship. The combination of the primary and foreign key columns and the rules that dictate the data that can be housed in those columns is the very beginning of understanding and utilizing RI to ensure correct and useful relational databases.

The set of RI rules, applied to each relationship, determines the status of foreign key columns when inserted or updated, and of dependent rows when a primary key row is deleted or updated. In general, a foreign key must always either contain a value within the domain of foreign key values (values currently in the primary key column), or be set to null.

The concept of RI is summarized in the following "quick and dirty" definition: RI is a guarantee that an acceptable value is *always* in the foreign key column. Acceptable is defined in terms of an appropriate value as housed in the corresponding primary key, or a null.

The combination of the relationship and the rules attached to that relationship is referred to as a referential constraint. The rules that accompany the RI definition are just as important as the relationship.

Two other important RI terms are parent and child tables. For any given referential constraint, the parent table is the table that contains the primary key and the child table is the table that contains the foreign key. Examine Figure 1.

The parent table in the employed-by relationship is the DEPT table. The child table is the EMP table. So the primary key (say DEPT-NO) resides in the DEPT table and a corresponding foreign key of the same data type and length, but not necessarily the with same column name, exists in the EMP table.

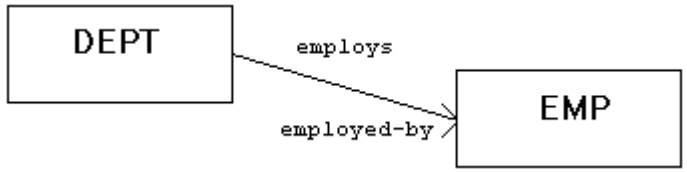

Figure 1. Parent and Child Tables.

Theoretically, there are three types of rules that can be attached to each referential constraint: an insert rule, an update rule, and a delete rule. Let's see how these rules govern a referential constraint.

Insert Rule

The insert rule indicates what happens when attempting to insert a value into a foreign key column without a corresponding primary key value in the parent table. There are two aspects to the RI insert rule:

1. It is **never** permissible to insert a row into a dependent table with a foreign key value that does not correspond to a primary key value. This is known as the RESTRICT insert rule.
2. The other aspect of the insert rule is whether or not actual values **must** be specified instead of NULLs.

For each relationship, the implementer must decide whether the foreign key value(s) must be specified when the row is initially inserted into the table. To determine this, ask the following question: "Does it make sense, in business terms, to know the primary key value in the parent table when adding a dependent row?"

If a foreign key value is specified, it must be equal to one of the values currently in the primary key column of the parent table. This implements the RESTRICT insert rule. If a foreign key value is optional, it must be set to null.

SQL Server's declarative RI supports both optional and required foreign key specification when a dependent row is to be inserted.

Update Rule
The basic purpose of the update rule is to control updates such that a foreign key value cannot be updated to a value that does not correspond to a primary key value in the parent table. There are, however, two perspective of the update rule: that of the foreign key and that of the primary key.

Foreign Key Perspective
Once a foreign key value has been assigned to a row, either at insertion or afterwards, it must be decided whether that value can be changed. Again, this is determined by looking at the business definition of the relationship and the tables it connects.

However, if a foreign key value is permitted to be updated, the new value must either be equal to a primary key value currently in the parent table or be null.

Primary Key Perspective
If a primary key value is updated, three options exist for how to handle foreign key values:

- Restricted Update — the modification of the primary key column(s) is not allowed if foreign key values exist.
- Neutralizing Update — all foreign key values equal to the primary key value(s) being modified are set to null. Of course, neutralizing delete requires that NULLs are permitted on the foreign key column(s).

- Cascading Update — all foreign key columns with a value equal to the primary key value(s) being modified are modified as well.

Microsoft's declarative RI enforces updated foreign key values to be either null or a current primary key value from the parent table. Restricted updates are enforced when a primary key column is updated. Neutralizing and cascading update is not supported by SQL Server.

Delete Rules
Referential integrity rules for deletion define what happens when an attempt is made to delete a row from the parent table. Three options exist:

- Restricted Delete — the deletion of the primary key row is not allowed if a foreign key value exists.
- Neutralizing Delete — all foreign key values equal to the primary key value of the row being deleted are set to null.
- Cascading Delete — all foreign key rows with a value equal to the primary key of the row about to be deleted are deleted as well.

The declarative RI provided by Microsoft SQL Server enforces only restricted deletes when a primary key column is updated. If there are rows in the dependent table with foreign key values equal to the primary key value of a parent row being deleted, the deletion of the primary key row is disallowed.

Declarative RI Constraints
A declarative referential constraint is added by coding the primary key in the parent table and one or more foreign keys in dependent tables. Constraints can be added using the **create table** and **alter table** statements. When implementing declarative referential integrity between a parent and a dependent table, the following rules must be followed:

For the parent table:

- A primary key must be identified in the **create table** or **alter table** statement.
- A unique clustered index is automatically defined by SQL Server for the primary key when the primary key constraint is specified in the **create table** or **alter table** statement.

For the dependent table:

- A foreign key that references the parent table must be identified in the **create table** or **alter table** statement.
- Although not explicitly required, it is strongly recommended that an index be defined for the foreign key for performance reasons. The index should not be unique (unless possibly if you are defining a one-to-one relationship).

All constraints will be named, whether explicitly or implicitly. It is better to explicitly name each referential constraint in the DDL. Failure to do so will cause SQL Server to assign a system-generated name, making future referencing of the constraint more difficult.

Declarative RI Implementation Concerns
Microsoft SQL Server provides two methods of defining referential integrity: declarative constraints and triggers. Before deciding on whether to use declarative constraints or triggers to support referential integrity, careful examination of the integrity requirements of each referential constraint should be performed. For certain types of constraints, declarative RI will not be an option. Remember that declarative RI can be used to support only the RESTRICT rule. However, regardless of the type of RI being implemented, certain standard rules of thumb apply:

- Sometimes a constraint needs to be set up within a single table. For example, a table of departments may need to record the management

structure of which department is subordinate to which other department. A managed_by_dept column may need to be a foreign key of the dept_no primary key-all within the same single table. A table is able to reference itself in a declarative RI constraint. This is referred to as a self-referencing constraint.

- It is not possible to drop a parent table until the constraint is removed or the dependent table is dropped.
- Proper authority is required to declare a foreign key. The table creator (or owner) is not allowed to create a foreign key reference to another table, unless the owner of the table being referenced has granted the REFERENCES privilege to the table creator.
- Constraints are checked before triggers are fired. If declarative RI constraints and triggers co-exist on the same tables, be sure that they are compatible with one another. For example, you should not code a delete trigger to delete foreign keys in conjunction with a declarative constraint because the declarative constraint will prohibit the trigger from ever firing.
- SQL Server provides a system procedure to retrieve information on constraints. It operates at the table name level. For example, the following statement will provide a report of all constraints that exist on the named table:
  sp_helpconstraint *table_name*
- Tables can be altered to add or drop declarative RI.

Referential Integrity Using Triggers

Triggers can be coded, in lieu of declarative RI, to support all of the RI rules. Of course, when you use triggers, it necessitates writing procedural code for each rule for each constraint. Complete referential integrity can be implemented using four types of triggers for each referential constraint:

1. A delete trigger on the parent table can be used to code:
   - delete restrict
   - delete cascade

    - delete neutralize
2. An update trigger on the parent table can be used to code:
    - update restrict
    - update cascade
    - update neutralize
3. An insert trigger on the dependent table can be used to code:
    - insert restrict
4. An update trigger on the dependent table can be used to code the restriction that a foreign key cannot be updated to a non-PK value.

Nested and recursive triggers are permitted. This supports a robust RI implementation. Triggers are the required method of implementing system-managed RI under the following circumstances:

- When deleted, inserted, and updated information needs to be explicitly referenced in order to determine the action to take. Triggers provide a method of doing this that will be discussed shortly.
- When an RI rule other than restrict is necessary. Declarative RI only supports restricted deletes and updates.
- When pendant delete processing is required. This is sometimes referred to as "reverse" RI. Pendant delete processing is when the parent row must be deleted when the last dependent row that references it is deleted. The only way to implement this type of constraint is with a trigger.

The Inserted and Deleted Tables
In order to use triggers to support RI rules, it is sometimes necessary to know the values impacted by the action that fired the trigger. For example, consider the case where a trigger is fired because a row was deleted. The row, and all of its values, has already been deleted because the trigger is executed after its firing action occurs. But if this is the case, how can we ascertain if referentially-connected rows exist with those values? We may need to access it in its original, non-modified format.

SQL Server provides two specialized tables with each trigger for just this purpose:

- the INSERTED table
- the DELETED table

Each trigger has one INSERTED table and one DELETED table available. These tables are accessible only from triggers. They provide access to the modified data by viewing the transaction log. These two tables operate as follows:

- When an insert occurs, the INSERTED table contains the rows that were just inserted into the table to which the trigger is attached.
- When a delete occurs, the DELETED table contains the rows that were just deleted from the table to which the trigger is attached.
- When an update occurs, the INSERTED table contains the new values for the rows that were just updated in the table to which the trigger is attached. The DELETED table contains the old values (before they were updated) for the updated rows.

It might not be readily apparent how these tables can be used. Let's examine some sample triggers to clarify the usage of the INSERTED and DELETED tables. Consider the following code that presents an implementation of the cascading delete RI rule:

```
create trigger title_del
on titles for delete
as
if @@rowcount = 0
        return
delete titleauthor
        from titleauthor, deleted, title
```

```
        where titles.title_id = deleted.title_id
return
```

When a row in the parent table (titles) is deleted, the delete is cascaded to the dependent table (titleauthor). This code implements the cascading delete RI rule.

Consider another example:

```
create trigger title_ins
on titleauthor for insert
as
declare @rc int
select @rc = @@rowcount
if @rc = 0
        return
if (select count(*)
        from titles, inserted
        where titles.title_id = inserted.title_id)!=@rc
        begin
                raiserror 20001 "Invalid title: title_id
                 does not exist on titles table"
                rollback transaction
                return
        end
return
```

This code implements the restricted insert RI rule. When a row in the dependent table (titleauthor) is inserted, we must first check to see if a viable primary key exists in the parent table (titles).

A final example depicts neutralizing updates:

```
create trigger title_upd
on titles for update
as
if update (title_id)
        if (select count(*)
                 from deleted, titles
                 where deleted.title_id = title.titleid) = 0
        begin
                update titleauthor
                        set titleauthor.titleid = NULL
                from titleauthor, deleted
                where titleauthor.titleid = deleted.title_id
        end
return
```

The first check is to see if the title_id was actually updated. Following that, the code checks to make sure that the title_id was not updated to the same value as it previously held. If it was, the neutralizing update should not occur.

If these two checks are passed, the update occurs. When a row in the parent table (titles) is updated, we check to see if any corresponding rows exist in the dependent table (titleauthor). If so, the foreign key columns must be set to null.

Trigger-Based RI Rules of Thumb

- SQL Server optionally enables the user to code a single trigger with update, insert, and delete logic embedded within it, or to code separate triggers (one for update, one for insert, and one for delete). It is better to use three independent triggers instead of combining insert/delete/update actions into one trigger. This makes debugging and maintenance easier to perform.
- Always count rows at the beginning of the trigger. A trigger is fired when the firing activity is performed, regardless of the actual number of rows

impacted (even if no rows are impacted). If the count of rows impacted is zero, the logic can be skipped and performance will be enhanced.
- Always use raiserror (or print) to return information about the success or failure of a trigger. Do not use select statements to send information, as the resulting output can be difficult to read and interpret.

Additional Referential Integrity Considerations

User- vs. System-Managed RI

Since system-managed, declarative referential integrity has not always been an option with SQL Server, your installation may have applications with user-managed RI already in place. It may be necessary to support both user- and system-managed RI in this situation.

Furthermore, even though system-managed RI is now available, sometimes user-managed RI is a more appropriate solution. One such instance is when it is always necessary for applications to access the parent and dependent tables (even when system-managed RI is implemented). For example, one application program always inserts the order row into the ORDR_TAB (parent) table before inserting the order item rows into the ORDR_ITEM_TAB (dependent) table and another application always accesses the rows in the ORDR_ITEM_TAB table for historical information before deleting them and then deleting the parent row from the ORDR_TAB table. Since these applications already access both tables, the additional overhead of system-implemented RI may not be worthwhile.

However, the added benefit of system-managed RI is that the integrity of the data is also enforced during ad hoc access (interactive SQL, data warehouse queries, etc.). When RI is maintained only in programs, data integrity violations can occur if data modification is permitted outside the scope of the application programs that control RI.

It is usually a wise move to implement system-managed referential integrity

instead of user-managed. But remember, SQL Server provides two methods of implementing system-managed RI: declarative constraints and triggers.

RI in Stored Procedures
Referential integrity can also be coded into stored procedures. Using stored procedures enables the programmer to influence when the RI code will be executed. This can enhance overall performance.

However, stored procedures are not generally recommended for supporting RI because:

- Stored procedures are not event-driven, so programmers must remember to execute them. Programmers are human, and errors can result in data integrity violations. Triggers are always automatically fired and will catch all RI violations (if properly coded).

- Waiting to check all RI at once with a stored procedure may enhance performance, but it may also cause a performance problem. Consider the following situation:
  1. PK updated
  2. FK inserted
  3. FK updated
  4. PK deleted
  5. execute RI stored procedure

  What if the first PK update is restrict? Then the stored procedure must prohibit the PK update (if FK rows exist) and roll back all of the subsequent processing. This will negatively impact performance.

- Finally, stored procedures carry with them all the negatives of user-managed RI with regard to ad hoc access. If ad hoc data modification is to be permitted, RI using stored procedures is not an option.

Microsoft SQL Server Considerations
Be aware that there are certain situations in which referential integrity can be bypassed. This can cause severe data integrity problems as well as significant confusion. The bcp utility can be used to load data into SQL Server tables without checking foreign key references. This makes the bcp load run faster because constraints are not checked. However, it also means that integrity problems will most likely exist.

Another way to bypass RI is using the WITH NOCHECK clause when adding a foreign key to a table that already has data in it. Without the WITH NOCHECK clause SQL Server will automatically check all current data for constraint violations when adding the new foreign key. Using WITH NOCHECK, however, SQL Server will not check existing data; it will only check subsequent INSERT and UPDATE statements.

A final way to bypass constraint checking is by altering the table with the clause NOCHECK CONSTRAINT. When the NOCHECK CONSTRAINT clause is specified no constraints defined for the table are checked. If there is a period of time when you do not want data integrity to be enforced you can alter the table to specify NOCHECK CONSTRAINT, and then when data integrity is to be automatically applied again alter the table back specifying CHECK CONSTRAINT. This method is not recommended because it inevitably results in data integrity problems that must be sought out and corrected later.

General RI Rules of Thumb
Regardless of the type of RI you plan to implement in your databases, there are several rules of thumb that should be heeded:

- Primary and foreign key columns can have different names, null attribute qualifiers (e.g., NULL vs. NOT NULL), and default values. The column attributes (e.g., CHAR(5)) must be the same. It is not possible to create a declarative constraint between two columns with different attributes.

Likewise, though possible, it is not wise to create a trigger-based constraint between columns with differing attributes.

- As a guideline, when the foreign key also participates in the primary key of the dependent table, the insert rule must be 'required', the update rule should not be 'allowed', and the delete rule must not be 'neutralize'. This will provide the proper entity integrity required of relational primary keys.
- If multiple relationships exist for the dependent row, they must all be verified before the row is inserted.
- When composite keys are used for the primary/foreign key relationship, a single row must exist in the parent table with key values that match all the columns of the foreign key for the row being inserted into the dependent table.

Synopsis

Microsoft SQL Server provides a wealth of features supporting referential integrity. Because one of the major problems plaguing production systems today is data quality, it is imperative that SQL Server DBAs understand, implement, and administer referential integrity in their database designs. Failure to do so can be a prescription for disaster.

From SQL Server Update (Xephon) October 1998.

Home