Dissecting SQL Joins
By Dann Corbit, Sr. Software Engineer

Table of Contents

Overview

One of the marvelous things about the relational model used in SQL queries is that we only need to store business facts in a single location. There is no need to store redundant copies of information, greatly reducing the chance of inconsistent information. If we were to store two copies of a customer's mailing address, there is always the chance that one of the copies might not get updated. But this increased reliability and reduced storage space does have a cost. We need to be able to tie the information together, and we need to connect things in different ways, depending on what we want to accomplish. So let's take a look at some different ways to combine information using SQL join operations to achieve the results that we need.

For this article, we will use some sample basketball data collected from an interesting site named BasketballReference.com This site contains actual NBA data which you can download from this link (if you would like to repeat my experiments) at http://www.basketballreference.com/ stats_download.htm or you can use the sample database I created using the data. They will probably make some corrections to the data due to a few minor inconsistencies that I discovered and pointed out to them, so you may prefer to use the sample data along with the original errors which can be found at ftp:/cap.connx.com/pub/nba/

Inner Joins

First of all, we will consider inner joins. Inner joins are useful when we want to match things from different tables, and when we want to discard everything from both tables where a match is not found. Consider a database that contains information about NBA basketball sports teams. In that database, we might have a table called "Season" which contains facts about the regular season. We might also have a table called "Teams" which contains facts about teams. And we could have a table called "Players" which contains the player's facts. If we perform an inner join on these three tables using common keys, we can discard all the information except the part that matches all three tables.

In **Figure 1**, we see an area marked "Keep!" that is shaped a bit like the bottom of an electric iron. This is the area of intersection that is achieved with an inner join on those three tables. The other areas of overlap are also discarded, along with any areas with no common key element (or that are removed with WHERE clause restriction).

Figure 1

Inner Joins, Continued

So let's take a look at a sample query that works the same way as Figure 1. This query selects the teams, players, and scoring averages from the three tables for the year 1999.

Figure 2

Outer Joins

Inner joins are the most straightforward sort of joins. But the kind of filtering produced by an inner join is not always what is wanted. This brings up a second sort of join: the outer join. Sometimes, rather than throwing away things that do not match from both tables, we want to keep all the data in one table and enhance that information with additional information from another table. For instance, we might have a table of part information which has part number, part price, and part description, but we have another table with ad copy in it for some of those parts. To form a catalog, we could do an inner join on part number, but that would throw out all the entries for which we do not have ad copy yet. We still want to list those parts also in our catalog! What is needed is an outer join. There are two frequently used kinds of outer join: the left outer join and the right outer join. With a left outer join, we are embellishing the table on the left, and with a right outer join we are embellishing the table on the right.
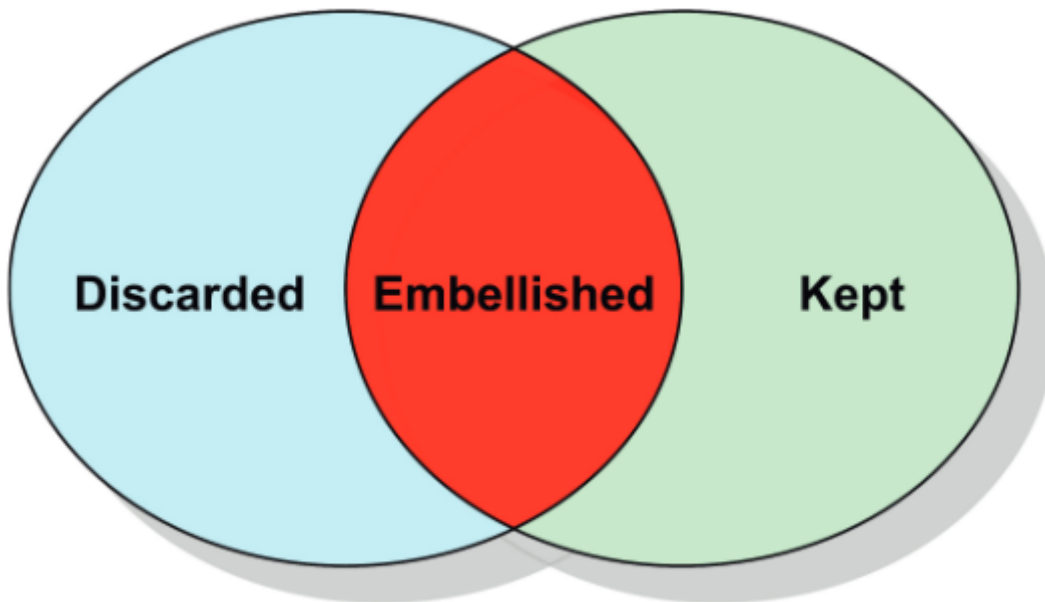

Figure 3

Figure 3 shows a right outer join. To envision a left outer join, simply reverse the kept and discarded areas. The kept data gathers additional information from the other participating table within the area labeled "Embellished".
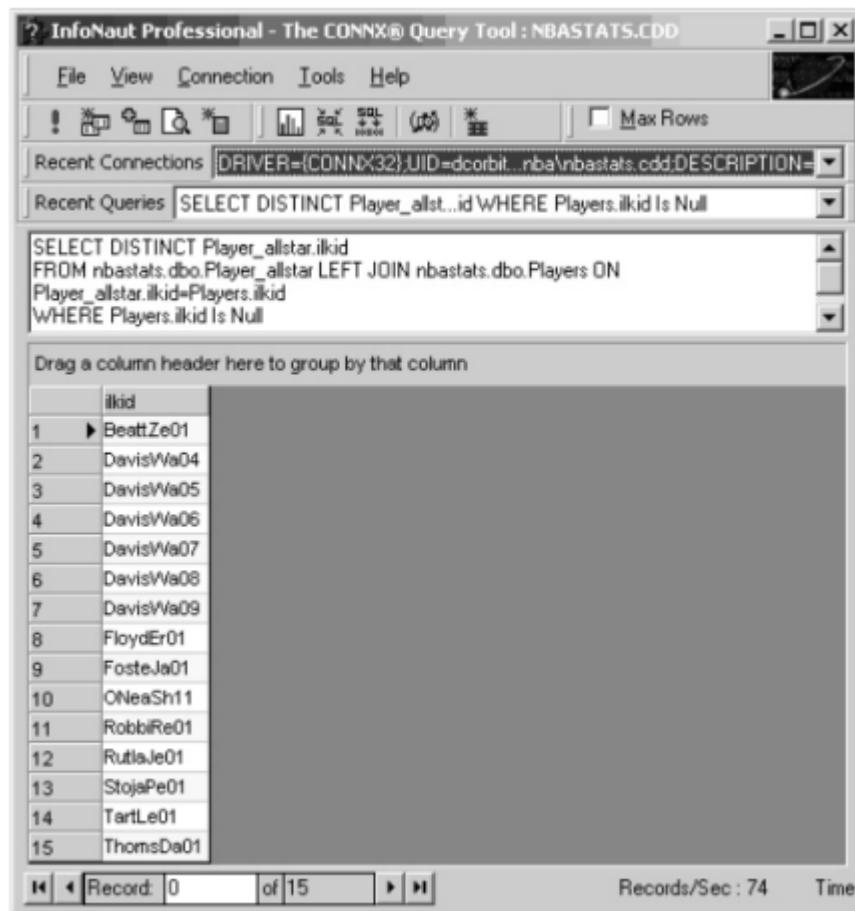
Other Joins

In addition to the left and right outer joins, there is a join called the full outer join. It is similar to left and right outer joins except that we do not throw away any data from either table and we also create matches wherever possible. For those rows where no matches exist, we create null values, just like the left and right outer joins. We could use a full outer join to discover errors in both tables at once using an approach like the one described above. In reality, it's not such a good idea, because for each row, we will have to carefully determine whether the problem is in the left or the right table. It is usually preferable to perform left and right outer joins instead, since the result is simpler to interpret.

There is another sort of join that usually happens by accident. It is called a Cartesian product. The Cartesian product happens when we forget to add join column criteria to limit rows in the result set. The result of a Cartesian product is a multiplication of every row in both tables. So if you have two tables, each with five thousand rows, the Cartesian product has twenty five million rows in it. And if both tables should have one million rows, then the Cartesian product has one trillion rows in it. Despite what some textbooks say about how useful the Cartesian product can be, it's probably not a good idea to form one.

There is a special classification of join types that is not really a new sort of join but (rather) a special case of one of the other join types called a self-join. A self-join occurs when a table is joined against itself. A typical example found in textbooks is when a table of employees contains also the supervisor data. A self-join can be formed against such a table in order to identify who works for a given employee or for whom a given employee works. Self-joins are also useful for many other things. We will use a self-join against our NBA data to ferret out more data problems.

Other Joins, Continued

Notice in the results from the self-join query in Figure 5 that there are exactly three players who have been entered with last name and first name reversed in one of the tables. One simple way to correct this problem is to normalize the database so that the first name and last name facts are carried only in the "players" table
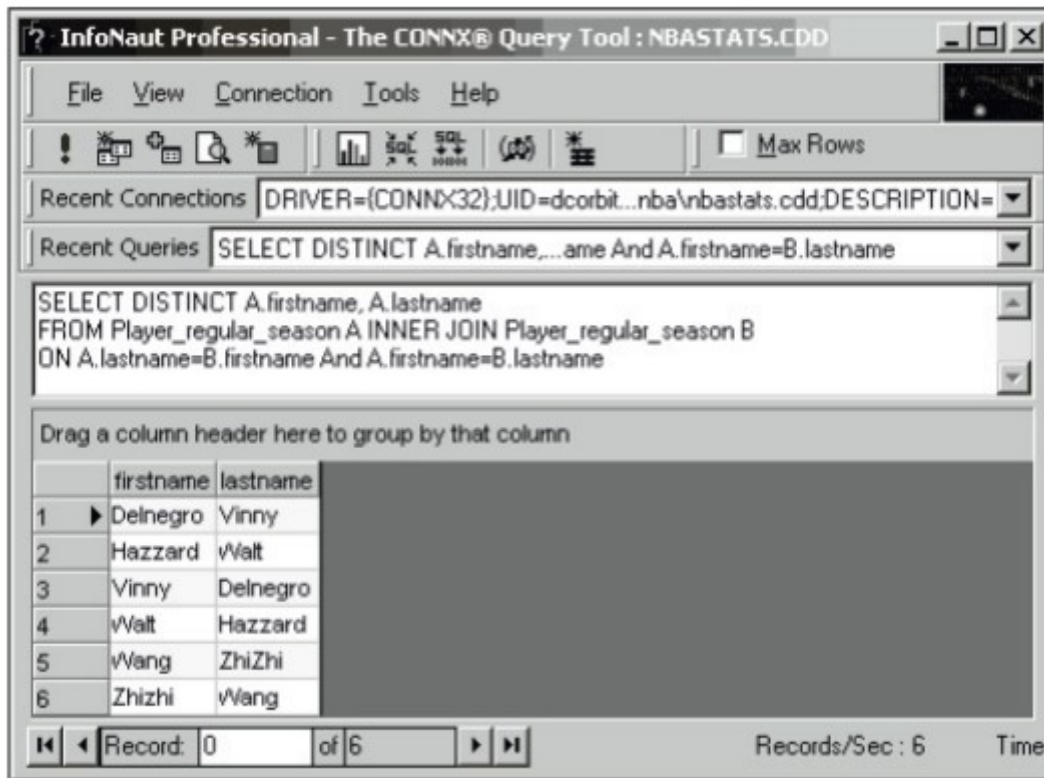

Figure 5

Problems with Using Joins

Well, now that we have the various join types fresh in our minds; let's think about problems that can arise from using joins. One obvious problem is choosing the wrong sort of join or leaving out the join criteria altogether and thereby creating a Cartesian product. When the wrong join type is chosen, the query results are usually not as expected (content is missing that should be there or content is there that should be missing).

It is obvious if a Cartesian product is created because our computer will hang for a while with the disk light blinking like mad and the CPU meter pegged at 100% as we rapidly run out of disk space (or worse yet – we are responsible for similar resource problems occurring on the main database server).

Consider a more common and insidious problem – the problem of poor join performance. We might formulate a join and discover that it does get the right answer, but it takes a very long time to get it. We might be joining on a column or columns that lack an index. When that happens, a database server looks at all the data in the table. Worse yet, it has to process the join by creating order in the data via hashing, nested loops, or some other workaround for the lack of an index file.

Solutions

One possible solution is to create an index. Unfortunately, most of the time, creating an index is not possible. Adding additional indexes to an existing database can slow update performance and is often not allowed, even if performance is not an issue, because of the danger of making changes to an existing, debugged, production system. One possible workaround for this sort of thing on joins with equality tests against a column is to use a special CONNX escape clause called {forcetempkey} which asks CONNX to process the join.

CONNX has a special join algorithm which is very fast. We have recorded times for a large data set that outperforms a native join on a unique clustered index.

Another possibility is to develop a reporting server using CONNX Data Synchronization, and create new indexes on the target server to your heart's content. Since the original tables do not need any sort of schema changes, you can make any modifications that you need to get the job done.