Database Subsetting: What You Need to Know

Creating Small Copies of Large Databases

A Net 2000 Ltd. White Paper

Abstract

Small subset versions of large databases are often required for development, testing and training. Such databases must be far smaller than the original and yet still form a self-contained system composed of data that is relevant, interrelated and complete.

This paper examines the many factors associated with the creation of such referentially correct subset databases and is, in most respects, a generic survey of the issues. However it must be noted that Net 2000 Ltd., the authors of this paper, sell a software data subsetting tool called DataBee (http://www.DataBee.com) and that this tool is referenced briefly towards the end of this paper..

Some keywords which may assist you in finding this document online are:

Database Subsetting, Data Subset, Cut Down Database, Data Archiving, Small Test Databases

Net 2000 Ltd.
http://www.Net2000Ltd.com
Info@Net2000Ltd.com

Table of Contents

Disclaimer

The contents of this document are for general information purposes only and are not intended to constitute professional advice of any description. The provision of this information does not create a business or professional services relationship. Net 2000 Ltd. makes no claim, representation, promise, undertaking or warranty regarding the accuracy, timeliness, completeness, suitability or fitness for any purpose, merchantability, up-to-datedness or any other aspect of the information contained in this paper, all of which is provided "as is" and "as available" without any warranty of any kind.

The information content of databases varies widely and each has a unique configuration. Readers should take appropriate professional advice prior to performing any actions.

Database Subsetting: What You Need to Know

Introduction

The demands of business today are such that the size and complexity of databases is continually increasing. Coupled with this ever increasing size is a continuing need for new applications to process the data. Therein lies the problem. Once a database exceeds a certain size it becomes very expensive in terms of both money and time to provide full size copies of the original database for the purposes of development, testing and training. Consider the cost - just in time - to replicate, test against, backup or recover a full sized copy of your production system. Add to this the costs of storage and backup media.

It is usually impractical to give every developer or tester a full size copy of the production database.

Many organizations resolve this problem by creating far fewer copies (often only one) of the production database than are really needed and then ask the development and testing teams to share its usage. Needless to say, this approach quickly becomes an exercise in diplomacy. The trouble soon starts: the developers collide with one another, the testers overwrite each others data and everybody squabbles because they have to wait for a time slot.

Using full size test and development databases can be a disadvantage.

Databases required by training or application development teams rarely need to be full size - in fact full size databases can be a drawback. Often, all that is required is a smaller version of the database that faithfully replicates the structure and content of the larger database. Truth be known - developers, testers and trainers don't like to work on full size copies of production databases as it slows them down. They also hate sharing.

What development and test teams really need are referentially correct subsets of the production database.

Development and testing teams need small copies of the production database. Creating these small databases manually can be complex - it is not good enough to populate a smaller database with a sample of data taken randomly from each table. To do so might provide a database smaller than the original by the desired amount, but the data would be meaningless. For example, such a database might have invoices without customers, customers with no addresses and so on.

Manually extracting a set of data that is meaningfully interrelated without taking too much, leaving any out or having duplicate records is surprisingly hard to do. The reasons for this will be discussed in detail later in this paper but a typical experience is that unless the schema is extraordinarily simple,

Manually creating referentially correct subset databases can be a "black hole" which consumes unlimited DBA hours.

manual procedures for creating referentially correct subset databases quickly become a descent into SQL hell. Due to the complexity of the task, manually created subsets are almost always a compromise with some foreign key constraints left disabled or some tables containing rows that do not have supporting rows in other tables.

Automated tools can create the subset databases.

Creating referentially correct database subsets manually is not the only option. There are software applications which can automate the creation process. Probably the most sophisticated example of such a tool is the DataBee software distributed by Net2000 Ltd.

Summary of Development and Test Database Creation Options

For most organizations the available options for creating development and testing databases are:

- Make full size copies and suffer the resource costs or accept the reality of potential conflicts between the database users.
- Manually create referentially correct database subsets and expend large amounts of DBA effort in creating what will probably be a "best effort" compromise, using a combination of SQL scripts.
- Use a tool.

Before You Begin: Plan Your Subset Database

Before you begin, find out who wants the subset database and what they want to use it for.

If you are going to create a subset database the first questions you will want to ask are: "*who will be using the database*" and "*what information do they need to see in it*". These are important issues, and the answers will largely define the content and focus of the created subset.

Knowledge of what the end users will be using the subset database for will help you define the primary table (or tables) on which the subset will be focused. The primary table is also known as the "pivot" or "driver" table.

Typically, a base set of data rows are retrieved for the driver table and the remaining tables in the schema are populated with rows which are related to the rows contained in the driver table. These relationships, as they cascade down through the dependent tables, are what will provide the subset database with the desired referential correctness.

An example:
Assuming the development team was implementing improvements to the invoicing system, the INVOICE table might be a logical choice for a driver table. Taking 10% of the INVOICE table could provide the base data and the CUSTOMER, INVENTORY and INVOICE_LINE tables would be populated with data relevant only to the rows currently selected for the INVOICE table. Carrying the concept onto the next stage, tables related to the CUSTOMER table (an ADDRESS table for example) would then be populated with rows relevant only to the customers in the subset. In this manner, the content of the subset ADDRESS table is populated with data which is only indirectly related to the original INVOICE table but is still referentially meaningful within the subset.

Another example:
Changing the driver table is often a better way of changing the data content of the subset schema.

The previous example assumed that the INVOICE table was to be the driver for the subset. What if, after some discussion, it was determined that the development team really needed a subset database consisting of certain representative customer types? The previous approach would probably not guarantee that the CUSTOMER table would be populated with a desirable diversity of data. Rather than go through the exercise of adjusting the base data for the INVOICE table to somehow pull a desired range of customers, it would be far better simply to change the driver table to CUSTOMER. The CUSTOMER table could then be populated, (by whatever criteria are desired)

and the INVOICE table would be populated with rows containing invoices relevant to the customers. All dependent tables for the INVOICE table would then be populated as before. The choice of the driver table for the subset database can dramatically change the data content of the subset database. A highly desirable feature of any subset creation mechanism is the ability to change the driver table without requiring a complete re-write of the creation software.

Choose Your Base Data

Once you have decided on the choice of driver table, the next step is to define the base data that will form the content of that driver table in the subset database. This issue requires careful consideration, as the end users will not be able to develop or test for situations which are unsupported by the data content of the subset schema.

An example:
The development team wish to develop a new invoicing system – but the subset database, although referentially correct, only contains examples of three industrial customers. Thus, their screens and reports are hard to test because they have only a minimal amount of data to work with.

It should be possible to populate the driver table using multiple criteria.

Ideally a subset database creation system should be able to use multiple criteria to populate a driver table. For example, it should be possible to populate the driver table with a sample of data (20% of all rows). It should also be possible to populate via a date range and then again via specific WHERE clauses in a SELECT statement. Of course, such multiple queries have an extremely high probability of returning some of the same rows. The subset database creation system should be able to filter out such duplicates without requiring a manual removal SQL run, or the formulation of the base table creation statement in a complicated multi-join or SQL UNION statement.

Practical Considerations

In order to create a populated subset database you will need a full size database to act as the data source. Also required will be a subset database with an empty but appropriately sized schema.

Find a Person that Knows the Schema

Populating a subset database – either manually, or by building it with an automated tool – is all about being aware of the meaning of the data in the tables.

If you do not know the relationships between the tables in the schema you will probably need to work with someone who does.

The next step will look at the importance of determining the relationships between tables and discuss ways in which these associations might be discovered. However, it must be emphasized that when identifying table-to-table relationships there is no substitute for working with someone who really knows the schema well.

Sometimes that person may not be the DBA. Database Administrators work with the schema from the viewpoint of maintenance and support and may not always know in great detail how the tables are related. Systems analysts, testers and developers are often the people who approach the database from the standpoint of table relationships.

Relationship Mechanics

Once the driver table is loaded, all new rows inserted into the subset database must be related somehow to the data in this "driving" table. If this is not the case, then the incoming data will not be referentially relevant. This requirement necessarily forms a parent-child relationship between the tables. The parent table with rows already in the subset needs supporting rows to be inserted in the child. Conversely the child table must obtain rows from the full size database but select only those related to the rows already in the parent.

All rows inserted into the subset database must be related to the rows already in place. This is what creates the referential correctness of the subset.

In other words, in order to populate the subset database table you will eventually have to construct a select statement (either manually or by proxy through an automated tool) which selects rows from the child table on the full size database and inserts them into the subset schema. The rows selected from the full size child table must be related to the rows already in the parent table in the subset database and the relationship is a standard join between one or more columns. The exact methodology of

the select can be performed in any one of a number of different ways.

An example:
Consider a simple two table system. The parent table on the subset database is the INVOICE table which has 100 rows. These invoices were pulled from the much larger table on the full size system and are all identified by a unique invoice number. The INVOICE table is the parent and it requires supporting rows in the child INVOICE_LINE table. These rows must be relevant to the 100 invoices already present. Pseudo code for the SQL insert statement would look as follows:

```
INSERT into SUBSET.INVOICE_LINE
(select * from FULLSIZE.INVOICE_LINE FS
where FS.INVOICE_NUMBER IN
(select INVOICE_NUMBER from
SUBSET.INVOICE));
```

Finding Table-to-Table the Relationships

Finding the relationships between tables can be quite difficult. Here are some suggestions.

Follow the Foreign Keys

Foreign Keys (if they exist) are one of the easiest methods of discovering table-to-table relationships.

The foreign keys (if present) intrinsically define relationships between tables. Each one of the foreign keys defines a relationship which absolutely must be supported. If the rows in the foreign key parent table are not present in the subset database the foreign keys will not enable. The logic of foreign keys can be reversed (for the purposes of creating a subset database) so if there is already data in the INVOICE table the foreign key relationship can be used to construct a population statement for the INVOICE_LINE table – even though the constraint really works in the opposite direction.

If you are manually creating a subset database, then it might be possible to automate the creation of extraction SQL statements based on the foreign keys. An automated subset creation tool should be able to read the foreign key constraints from the database schema and build population SQL statements (or whatever mechanism is used) in an automatic manner.

Column Similarity

Similarity between column names can also be a useful technique to discover table-to-table relationships.

Many schemas have the referential integrity implemented at the application level rather than in the foreign keys. One useful way to find table relationships is to find tables with the same column names. For example, it is probably a reasonable assumption that every table with an EMP_ID column can reasonably be expected to be related to each other via that column. Of course, this is not always the case – but it is an area worth investigating. Manually such an exploration requires a large number of selects on the schema structure views. An automated tool should provide a point and click solution.

ER Diagrams

ER Diagrams are a good source of relationships.

Many table-to-table relationships should be determinable from an ER diagram – if it exists. Experience suggests that for many sites this is not a very useful option because the ER diagrams, if they exist at all, are usually not kept up-to-date.

Knowledgeable Person

There is really no substitute for a knowledgeable person when designing a subset database.

One of the most useful ways of discovering table-to-table relationships is to ask a person knowledgeable in the schema. Unless this person has been specifically tasked with the job of helping you build the subset schema, you will find yourself using up goodwill at rapid rate. A useful way of maintaining harmonious relations is to create a list of tables in the subset that are already populated (using rules you have figured out yourself) and a list of tables that have no rows. Then ask the schema guru "*Are any of these empty tables joinable to any of these non-empty ones?*" If you are creating the subset manually you will have to create the table and row count lists yourself. An automated tool should be able to generate such lists for you.

Population Mechanics: Actually Doing It

There are two fundamental approaches to the population of subset databases. The first method is to determine all of the desired relationships and to run the SQL extraction code in the required order. The second method is to develop the subset population mechanism incrementally through a populate, see what you get, add a rule and repopulate procedure.

You can populate all at once or use an iterative method.

The first method (all at once) is probably more useful when manually creating subsets. When one is doing the subset population manually the task is so complex it isn't really practical to pursue an iterative approach.

Iterative development of the extraction routines should be the preferred option and is possible with automated subset creation tools. It is also the most desirable approach since they hide most of the complexity from you. With automated tools, such as the DataBee software, you would add only one or two rules per trial and perform a sample population to see the effect. The newly populated tables can activate previously inactive table-to-table relationships (perhaps put in place to support foreign keys) and cause the population of numerous other tables.

An automated subset creation tool should let you browse the relationships currently in place between tables. In addition, it should let you see which tables are involved in the most relationships. By far the most effective way to build a subset database is to figure out how to populate those frequently referenced tables using the rows from tables in the subset database that are already populated. Once rows are placed in such tables, the relationships they are involved in should cause the automatic population.

Subset Population Issues

There are a number of issues that need to be considered during the population of a subset database. These issues should be taken care of in an automated subset creation tool. However, they are some of the primary sources of trouble when creating subsets manually.

The Duplicate Row Problem

Duplicate rows are a major issue which must be coped with when populating subset schemas.

It is quite common for a child table to have two or more parents. This means that the child will need to contain rows that support all of the parent rows in each parent table. The problem arises because the child table is generally loaded separately for each parent table. Of course, many of the rows required to support one parent table will be duplicates of the rows required to support another.

The duplicate rows cannot remain in the child table – at the very least they will prevent you from enabling any primary keys. There are really only two available options to resolve this matter. The duplicate rows can be removed manually or the population SQL query has to be written to join against all parent tables. Either way is slow and introduces yet another step into what is already a complex process. An automated subset creation tool should be aware of the rows it already has for a table and automatically sift out duplicates (either by discarding them as they arrive or by not requesting them in the first place).

The Circular Path Problem

When populating subset schema tables it is usual to focus in on only two tables at any one time – a parent table and a child table. This simplistic viewpoint can have some unusual side effects.

The circular path problem causes new rows to appear in tables for which you have already processed the dependents.

The table-to-table relationships in most schemas are very complex. It is quite common to have circular paths of parent-child relationships. For example: table A needs supporting rows in table B, table B needs supporting rows in table C and table C needs rows from table A. Circular path relationships can have quite large number of tables in the chain. Since tables can require supporting rows from more than one child table, it is possible to see very complicated arrays of multiple interlinked table-to-table relationships.

The effect of this complexity is that new rows may well appear in tables you previously thought you had completely processed. For example, you may well have populated table A and

retrieved supporting rows for all of table A's children. Later on, some other table-to-table relationship will treat table A as a child and add a few more rows to it.

The newly added rows in table A will not have supporting rows in that tables children. The arrival of new rows must be noted and the child tables must again be populated to contain supporting rows for the incoming table A rows. Of course, it is undesirable to completely repopulate the child table in its entirety – the optimum solution is to only perform a fetch for the new rows. In any case, duplicate rows are sure to arrive and will need to be removed.

It is the Circular Path Problem that makes manually creating subset databases so difficult. The whole procedure quickly becomes an iterative process in which tables and their dependent chains (and the dependents of the dependents) must be processed until every table has the rows required to support everything else. An iterative process as described above is practically impossible to manage manually.

The Self-Referential Relationship Problem

The self-referential relationship problem can cause many thousands of iterations if not handled properly.

Tables can have a table-to-table relationship with themselves. This is most often seen in self-referential foreign key constraints. Theoretically there is no real problem with self-referential relationships – they are just another table join. However, in practice, self-referential relationships cause a massive number of iterations to be performed. Since the parent and child tables are the same on each run, populating the child also populates the parent. Then the Circular Path Problem begins and the new parent record must then be processed for its own children. Depending on the database design these iterative chains can be tens of thousands of links long.

Self-referential relationships should be identified (either manually or by the automated tool you are using) and the normal SQL population mechanism should be replaced with a Connect By Prior, Recursive Common Table Expression or similar functionality. This will prevent massive amounts of iteration as the self-referential relationship chains are followed.

A Production Process

The creation of subset databases should form part of a production process.

Once you have created a subset database for the development and test teams you will probably find that they like it so much they want another. Of course, if things haven't changed all that much in the full size version you can probably just clone the existing subset or restore it from a backup somewhere. But what do you do if there have been major changes in data content on the original full size database? The point of having a subset database is that it should be a faithful, referentially correct small copy of the original. The only option is to create a new copy of the subset database. This is where automated tools really become useful. A manual procedure will take roughly the same amount of time each time it is used. An automated tool might require configuration in order to create the first subset database, but subsequent runs should be a push button operation.

Schema changes are also an issue – if the schema structure changes in the full size database, then the subset databases derived from it will also need to change. Ideally the creation process should be robust enough to cope with schema changes without major rewrites of the software. This is true for both manual and automatic subset database creation solutions.

Subset Creation Software

The previous discussion outlined the major issues involved in the creation of a subset database. A number of references were made to automated subset creation software. These tools are designed to simplify the process of creating subset databases and are intended to eliminate most problems for you. One such software tool is DataBee.

DataBee

The DataBee software creates subset databases.

The DataBee software is an automated tool that creates referentially correct subsets of large Oracle and SQL Server databases. It was specifically designed to address the issues discussed in this paper.

DataBee contains a suite of useful tools for discovering table-to-table relationships.

As with any subset creation software, DataBee requires configuration before use. However, DataBee contains the tools to discover table-to-table relationships and readily lends itself to a step-by-step development approach.

Base Data and Driver Tables

DataBee supports multiple driver tables and a variety of base table population measures.

The DataBee software tool supports multiple driver tables and these tables can be populated with an unlimited number of SQL queries. Duplicates are automatically removed.

Duplicate, Circular Path, and Self-Referential Problems

The many troublesome issues surrounding duplicate rows and relationship complexity are handled automatically by DataBee.

DataBee handles the Duplicate Row, Circular Path and Self-Referential problems automatically. You, as the user, never have to remove duplicates – it is all done automatically. In addition, if new rows are added to tables already processed DataBee will automatically and efficiently handle the fetch of the rows for dependent tables. DataBee ensures each and every foreign key will enable in the subset database and that all logical data relationships are valid.

A True Production Process

DataBee forms a true production process and creating subset databases is a simple push button operation.

DataBee provides a true production process. Once the initial configuration has been performed, the creation of subset databases is a push button operation. With DataBee, it is simple and easy to create as many high quality subset databases as is required. More information and free evaluation downloads of the DataBee software can be found on the DataBee Web site at: http://www.DataBee.com