A Better Method for Dealing With Identity Gaps

By Rob Verschoor

A new way to defeat the challenging and time-consuming "identity gap" phenomenon while using ASE.

Rob Verschoor is a freelance consultant in The Netherlands, specializing in Sybase performance and tuning issues. He can be reached at rob@sypron.nl.

A well-known feature of Sybase Adaptive Server Enterprise is the "identity column." Identity columns are useful because they automatically generate consecutive numeric values upon insertion of new rows, while having minimal impact on performance. A disadvantage of using identity columns is that so-called "identity gaps" may occasionally occur. Identity gaps are large, sudden jumps in the value of an identity column, which often cause problems for applications.

Unfortunately, there is currently no quick or easy way of repairing an identity gap. The standard remedy recommended by Sybase can be rather time-consuming, resulting in hour-long application downtimes. For this reason, DBAs or developers sometimes prefer to avoid the use of identity columns for applications with high availability requirements.

This article presents a database design technique that allows identity gaps to be fixed easily and quickly, taking no more than a few seconds. With this approach, full advantage can be taken of the functionality of identity columns, while minimizing the impact on application availability in case identity gaps need to be repaired.

What Identity Gaps Look Like

To illustrate the problems related to identity gaps, let's assume the following simplified database design for storing invoice data:

```
create table invoices
(invoice_nr numeric(10,0) identity,
customer_nr int,
amount money)

/* insert new invoice */
insert invoices (customer_nr, amount)
   values (@new_customer, @new_amount)
```

This involves a database table named "invoices," having an identity column called "invoice_nr". When a new invoice is created, the customer number and the amount payable are inserted into the invoices table. No value is specified for the invoice_nr column, as this value will automatically be generated as a result of the identity property. The value assigned to the new invoice number will be 1 higher than the previous invoice number that was generated. This way, identity columns automatically generate unique, consecutive numbers which make ideal primary keys in database systems.

An identity gap has occurred when there is a large, unexpected jump in the value of an identity column, as in this example:

```
1>    select invoice_nr from invoices order by 1
2>    go

invoice_nr
----------
    (..)
   10028
   10029
   10030
   10031
  5000002
  5000003

  (10033 rows affected)
```

For some reason, the invoice inserted after 10031 was not assigned number 10032 as expected. Instead, the invoice number jumps to 5000002 and continues to count up from there. This phenomenon is referred to as an "identity gap."

Such discontinuities in identity values are often a serious application problem. For example, some applications might not be able to handle invoice numbers of more than, say, six digits. Indeed, identity gaps are usually discovered because of application errors resulting from the unexpected high values in an identity column. From the point of view of a DBA or end-user, identity gaps are almost always inconvenient, and often should be fixed immediately.

It should be noted here that it is always possible that some individual identity column values are missing. This can happen when the transaction containing the insert operation is rolled back. The identity value that was already issued for this rolled-back row will not be re-used and therefore will never show up as an invoice number. For the purposes of this article, an "identity gap" refers to a gap of a large number of units, and not to individual missing identity column values.

Why Identity Gaps Occur

Identity gaps can occur following a rough server shutdown (shutdown with nowait) or a crash of the server process. This is related to the algorithm the server uses to generate identity column values. Essentially, a counter is kept in server memory, holding the identity value most recently issued. When a new row is inserted, this counter is incremented and the resulting value is assigned to the identity column in that row. While the new data row itself is written to disk, the new value of the in-memory counter is not. Only when the server is shut down in a normal way is this value saved on disk.

This algorithm makes the identity feature very fast, because no I/O is required to generate a new value. On the other hand, should the value of the in-memory counter be lost, it is not possible to continue at the next identity value, because the previous issued value was not saved. In this case, the server will continue generating identity values starting at some much higher value—which is what creates the identity gap.

Exactly at which higher value the server picks up is determined by the configuration parameter identity burning set factor, which, to a certain extent, can be used to limit the maximum possible size of an identity gap. However, because this a server-wide setting, it is not possible to apply this to individual tables. See the ASE System Administration Guide and Technical Document #20113 at techinfo.sybase.com/css/techinfo.nsf/DocID/ID=20113 for a detailed description of how to use this configuration parameter.

In view of these underlying technicalities, the risk of running into identity gaps could be seen as the price one has to pay for the high performance offered by the identity column feature. Also, it is clear that this risk cannot be completely excluded. DBAs should be aware of this, and be prepared to perform recovery procedures.

Fixing Identity Gaps the Slow, Classical Way

Let's assume that, once an identity gap has occurred, it should be repaired as soon as possible. Basically, this has always involved the following two actions:

- Updating those rows which have received very high values in their identity columns to the correct values that should have been generated instead. In the above example, 5000002 and 5000003 should be changed to 10032 and 10033, respectively.
- Resetting the value of the identity column downwards, so that a correct value generated when the next row is inserted (10034 in the example).

Unfortunately, none of these actions can be performed directly. The DBA is not allowed to update the value of an identity column in an existing row, nor can the value of future identity values be modified downwards. Therefore, the only available procedure to fix a situation where identity gaps have occurred has been the following:

1. Switch on the “identity_insert” option for the invoices table with the following statement:

```
set identity_insert invoices on
```

   This option will allow an explicit value to be specified for the identity column in an insert statement. Note that this option can be enabled for only one table at a time.

2. For all rows with problematic “high” identity values, delete the row and re-insert it with the proper value one would have liked to see there in the first place, effectively performing an update. In the example, this would update invoice numbers 5000002 and 5000003 to 10032 and 10033, respectively.

3. BCP the invoices table to a file, naming it, for example, “invoices.bcp”.

4. Drop and re-create the invoices table. Note that dropping the table will implicitly switch off the “identity_insert” option.

5. BCP the invoices.bcp file back into the invoices table using the BCP “-E” option (for “identity insert”).

6. Rebuild any indexes on the table, if applicable.

Applying this procedure to a multi-million row, real-life application table could well take a few hours, during which the invoicing application is unavailable. In many cases, end users and management would probably find this situation unacceptable and ask their DBA unsettling questions as to why this problem could not have been avoided.

Even though this will hopefully remain a rare incident, there is always a possibility that such a repair procedure may have to be performed, because the risk of identity gaps just cannot be fully excluded. For this reason, DBAs or developers sometimes choose not to use identity columns at all for applications with high availability requirements, because this could lead to unacceptable application downtime.

The above is not a far-fetched or hypothetical scenario. Cries for help are regularly posted in the Usenet newsgroup comp.databases.sybase by DBAs suddenly facing an identity gap who are desperate for a quick solution. Unfortunately, Sybase has not felt it necessary to implement additional functionality for making the process of fixing identity gaps easier, leaving DBAs with nothing but the rather clumsy procedure described above.

A Better Way of Repairing Identity Gaps

We will now look at a database design approach that allows DBAs to fix identity gaps quickly, in a matter of seconds or, in the very worst case, minutes. The first step to achieve this is to use two database tables instead of one: the application table “invoices” plus a separate keytable named “invoices_keytable”.

```
create table invoices_keytable
(dummy_key numeric(10,0) identity)

create table invoices
(invoice_nr numeric(10,0),
 customer_nr int,
 amount money)
```

Note that the “invoice_nr” column is no longer an identity column, but an ordinary column of datatype numeric. The identity column has moved to table “invoices_keytable”, which contains just this one column and nothing else. The purpose of this identity column, named “dummy_key”, still is to generate key values for new invoices, but in a slightly different way than before.

When creating a new invoice, first a new invoice number is generated by inserting an “empty” row into invoices_keytable. The identity value assigned to the dummy_key column in this row is then automatically available through the global, session-specific variable @@identity. This invoice number is then used to insert the actual new invoice data into the invoices table:

```
/* insert "empty" row to generate new invoice number */
insert invoices_keytable values ()

/* use identity value as key value for new row */
insert invoices (invoice_nr, customer_nr, amount)
      values (@@identity, @new_customer, @new_amount)
```

This two-step method of inserting a new invoice is functionally identical to the classic situation where the invoice_nr column in the invoices table would have the identity property. Also, identity gaps can still occur in this design, with the same consequences for the application as before. However, once this happens, this new approach offers a much better way to repair the identity gap. A DBA should then take the following steps:

1. Update the invoices table using a normal update statement:

```
update invoices
set invoice_nr = 10032
where invoice_nr = 5000002
```

Contrary to the classic approach, this update will work because the invoice_nr column is a normal, not an identity, column. (NB: A similar statement is required for correcting invoice 5000003.)

2. Drop and re-create invoices_keytable. No data is lost here, because the data rows in this table do not contain any useful information.

3. Reset the identity column value in invoices_keytable with the following statements:

```
set identity_insert invoices_keytable on
insert invoices_keytable (dummy_key) values (10033)
set identity_insert invoices_keytable off
```

The effect of these statements is that for the next row that will be inserted, the identity value generated will be 10034, which is exactly what the next invoice number should be. This is because the statement set identity_insert ... on allows an explicit identity value to be inserted in the dummy_key column of invoices_keytable. If this value is higher than the highest identity value issued, the identity value is adjusted upwards. Because this mechanism doesn't work in a downward direction, the table must be re-created first to make this trick work.

Of these three steps, the last two will always be very fast; they should take no more than a few seconds. The first step (updating the invoices table) should normally not take much time either; in case there are many invoice numbers that need to be corrected, this should still not take longer than a few minutes in the worst case.

The obvious way of implementing this reparation procedure is to put these actions in a stored procedure. In case an identity gap is found, the DBA just needs to execute this procedure and the problem will be fixed automatically. An example of such a stored procedure can be downloaded from www.euronet.nl/~syp_rob/idgaps.html.

Technical Considerations

The first feature that makes this design work is the use of the global variable @@identity, which always holds the identity value assigned most recently in the current session. Because this variable is session-specific, different user sessions can be inserting into the invoices_keytable concurrently, without influencing each other's @@identity contents.

The table invoices_keytable is used here in an unconventional manner: Its only purpose is to quickly obtain a new invoice number in @@identity by inserting an "empty" row. The inserted row itself is not of any interest: The table could be truncated regularly to stop it from growing too large, for example, by putting the table on a separate segment and using a threshold procedure.

In this scenario, inserts are performed into two database tables instead of in one table as in the classic situation. The extra insert into invoices_keytable is the price to be paid for the increased recoverability of the application. Fortunately, this overhead is very small. First, there is no need for an index on invoices_keytable, because no data will ever be retrieved from this table. Second, the table can be partitioned so that concurrent users will be inserting on different data pages, thus avoiding lock contention. In practice, the extra overhead turns out to be hardly noticeable.

Another point worth mentioning is that the two insert operations do not need to be encapsulated in a transaction. Suppose that the two inserts are indeed part of one transaction, and for some reason it is decided to roll back the insert into invoices. While this will cause no data row to be inserted into invoices_keytable, this will not have any effect on the next identity value to be assigned. Once an identity value is issued, it cannot be "given back" or re-used anymore, due to the underlying memory-based algorithm. Therefore, transactional consistency between these two tables is not relevant.

This design technique works in all Sybase versions from 10.0 onwards. Note that table partitioning is only available in version 11.0 and later.

Proactive Reparation of Identity Gaps

Designing a database to allow quick reparation of identity gaps is a major improvement compared to the traditional situation. Still, it might be preferable to ensure identity gaps will never strike an operational application at any time. This can be achieved by always running a program directly after server startup, which performs the following actions:

1. It inserts an "empty" row in invoices_keytable to obtain the next identity value through @@identity.

2. It retrieves the highest existing key value from the invoices table; hopefully, there would be an index to support this query.

3. It then compares these two values. If everything is normal, then the difference between these two values is not more than a few units (small gaps can always exist because individual insert operations can have been rolled back). If the difference is greater than, say 100 units, this means an identity gap exists. A reparation procedure can then be run automatically, which drops and recreates the invoices_keytable.

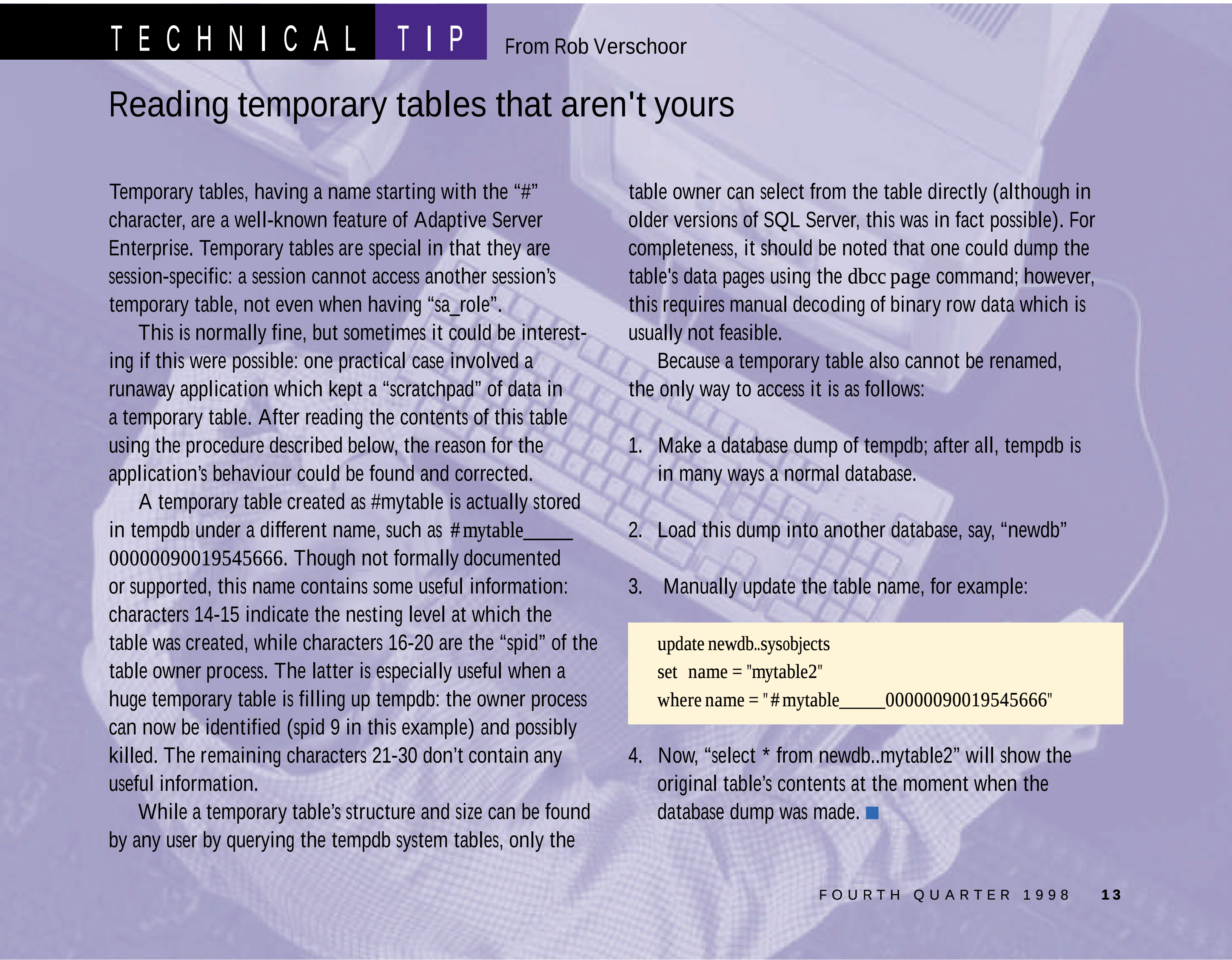
Note that this check will cause one invoice number value to be missing from the invoices table, in case no identity gap exists. If this is not desirable, a variation on this procedure is to always rebuild the invoices_keytable, using the highest invoice number from the invoices table.

Assuming these actions are performed directly after server startup, and before any applications are using the database, an identity gap (if present) will not yet have affected the values in the invoices table because no new invoices have been inserted yet.The situation will now be corrected immediately before wrong invoice numbers are generated. This ensures that identity gaps do not get a chance to go unnoticed until the first serious application problems start to appear.

The downloadable example stored procedure mentioned earlier also works for this situation.

Conclusion

It is possible to avoid the problems caused by identity gaps, the risk of which is implied when identity columns are used. Using the two-table design technique described in this article, identity gaps can be repaired quickly and even automatically, in contrast with the much more inefficient classic approach. This results in a significant improvement in application availability at a negligible performance cost. ■



TECHNICAL TIP From Rob Verschoor

Reading temporary tables that aren't yours

Temporary tables, having a name starting with the "#" character, are a well-known feature of Adaptive Server Enterprise. Temporary tables are special in that they are session-specific: a session cannot access another session's temporary table, not even when having "sa_role".

This is normally fine, but sometimes it could be interesting if this were possible: one practical case involved a runaway application which kept a "scratchpad" of data in a temporary table. After reading the contents of this table using the procedure described below, the reason for the application's behaviour could be found and corrected.

A temporary table created as #mytable is actually stored in tempdb under a different name, such as #mytable_____00000090019545666. Though not formally documented or supported, this name contains some useful information: characters 14-15 indicate the nesting level at which the table was created, while characters 16-20 are the "spid" of the table owner process. The latter is especially useful when a huge temporary table is filling up tempdb: the owner process can now be identified (spid 9 in this example) and possibly killed. The remaining characters 21-30 don't contain any useful information.

While a temporary table's structure and size can be found by any user by querying the tempdb system tables, only the table owner can select from the table directly (although in older versions of SQL Server, this was in fact possible). For completeness, it should be noted that one could dump the table's data pages using the dbcc page command; however, this requires manual decoding of binary row data which is usually not feasible.

Because a temporary table also cannot be renamed, the only way to access it is as follows:

1. Make a database dump of tempdb; after all, tempdb is in many ways a normal database.

2. Load this dump into another database, say, "newdb"

3. Manually update the table name, for example:

```
update newdb..sysobjects
set   name = "mytable2"
where name = "#mytable_____00000090019545666"
```

4. Now, "select * from newdb..mytable2" will show the original table's contents at the moment when the database dump was made. ■