Experiences of Global Temporary Tables in Oracle 8.1

Global Temporary Tables are a new feature in Oracle 8.1. They can bring significant performance improvements when it is too late to change the design.

by David Kurtz, Go-Faster Consultancy Ltd.

What was the problem?

My first encounter with Global Temporary (GT) tables came while trying to increase the performance of a Swiss Payroll System. It calculates payroll for 37000 employees. It’s a fabulously complex calculation, which generates an average of 890 values per employee per month that are written to intermediate result tables in the database. Of these, 120 values are retained, by being copied to the permanent balance table. The retrospective nature of the payroll process means that you can have to go back and recalculate payslips that are up to 2 years old, and then every payslip since, in a single process run. This calculation shuffles huge amounts of data. Each month the volume of data and processing would grow as another month was added to the calculation. At its peak, the weekend 'retro' calculation process had extended to 54 hours, and generated over 500Gb of redo log at rates of up to 24Gb per hour.

We had a major I/O bottleneck. The archive log writer could not keep up with the redo log switching while the payroll process ran. There were 20 redo logs each of 500Mb, and at times all of them would require to be archived. This had implications for the backup strategy. We could have faced taking 3 days to reapply up to 500Gb of redo logs to recover a 150Gb database.

The long-term solution has been to change the process to reduce unnecessary repetitive processing, thus reducing I/O. However, this was not a short-term option, we had run out of weekend batch window, and running the payroll calculation during the day would serious degrade the performance of the online HR system. Redesigning the disk sub-system to handle the I/O load helped. However, converting the intermediate result tables to GT tables significantly reduced the volume of redo and archive logging, greatly alleviated the I/O problem, and improved the performance by a factor of 3. This bought enough time to develop changes to the process. Critically, GT required only minimal code change.

What is a GT Table?

First of all, it is a table, it looks like an ordinary (heap) table when you describe it, and you can access it like an ordinary table. Its definition is persistent, but its content is temporary.

There are two flavours of GT table:

- Delete on Commit: Data is lost at the end of each transaction, as well as the end of the session. This is the default action.
- Preserve on Commit: The data is preserved at the end of a transaction and only lost when the session terminates.

This flavour is specified in the 'create table' statement.

CREATE TABLE heap (a number) TABLESPACE USERS;

CREATE GLOBAL TEMPORARY TABLE gt (a NUMBER) ON COMMIT PRESERVE ROWS;

Physically, a GT table exists in the temporary segment. Hence it is not possible to specify any physical storage parameters, or tablespace for the GT table. The values for all these parameters will be reported as NULL on DBA_TABLES for GT tables. Like an ordinary table, you can create indexes, constraints (other than referential constraints), comments and triggers. Indexes on GT tables also reside in the temporary segment. There is no special syntax to be added, but the same restrictions exist on specifying physical storage parameters, which will also be reported as null on DBA_INDEXES.

There is nothing to stop you from analysing a GT table, but no cost-based optimiser statistics will be generated or stored. The columns in DBA_TABLES and DBA_INDEXES will also be null. The only way to specify optimiser statistics is with the DBMS_STATS package (see *Oracle8i Supplied PL/SQL Packages Reference*).

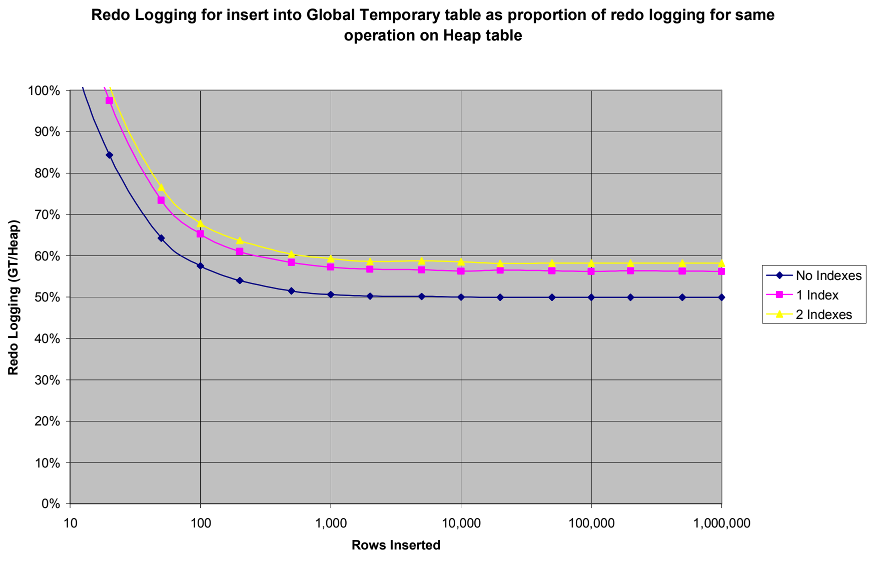
The content of the GT is only visible to the session or transaction that created it. Two different sessions can insert identical rows into what is apparently the same GT table with a unique constraint. Physically, each session has its own version of the GT table in the temporary segment, which was created when the session or transaction first referenced it. If one session inserts two identical rows into the same table then a unique constraint error will still be produced as usual. When a session terminates the temporary segment is released. For ON COMMIT DELETE tables, the segment is released at the end of the transaction. The space overhead in the temporary segment is comparable to that required for a similar heap table. The instantaneous space overhead for GT tables, and any other temporary object, can be observed in *v$sort_usage*. It is not reported in DBA_SEGMENTS. When a transaction is committed, entries for ON COMMIT DELETE tables disappear.

GT tables are not recoverable, and so redo logging is greatly reduced. Hence also, it is not possible to specify the LOGGING, NOLOGGING, MONITORING or NOMONITORING options when creating a GT table. There is still some log information written when accessing a GT table, which is mainly related to rollback segment activity.

The SQL Reference Manual also lists restrictions on partitioning, clustering, index-organisation, parallel DML, distributed transactions, and nested table or varray type columns.

How much redo logging is saved?

In order to quantify the saving in redo logging I constructed a test. I created a number of heap and GT tables with identical structures, but with different numbers of indexes and columns in the indexes. I populated them with a simple PL/SQL loop, and measured the difference in the 'redo size' parameter on *v$mystat* for different numbers of rows inserted into the tables. I used *v$mystat* rather than *v$sysstat* to remove the background noise of other database activity.



The proportionate saving in redo is fairly constant for larger numbers of rows, but there is a saving for a few as 20 rows. I have tried this test on different platforms and with different sizes of tables, and number of indexes, and indexed columns. The results are always similar, but the savings when only working with a small number of rows depend upon the volume of data.

How did we use GT tables?

Detailed testing and measurement had shown that introducing Global Temporary tables would significantly improve payroll performance. Their introduction became the major driver for the upgrade from Oracle 8.0 to 8.1.6.

We changed the intermediate result tables to be global temporary tables. The payroll calculation commits between the sixty or so payroll 'rules'. The temporary calculation results are written to 'temporary calculation result' tables that are truncated between rules. These became DELETE ON COMMIT tables. The retained results are written to 'permanent result tables' that are copied to the final balance table at the end of the calculation for each month. The permanent result tables are truncated between calculations for each month. These became PRESERVE ON COMMIT tables.

No High Watermark Problems

Batch processes sometimes temporarily store data in tables. Such tables are sometimes keyed by a process instance number, so that many instances of the process can run in parallel. Such tables often drive the processing, and so the queries on them tend to access them by full table scans, which scan the table up to the High Water Mark (HWM). An unusually large volume of data to be processed can raise the high water mark, and so increase the process time. Even an insert into a table that is rolled back, rather than committed, can raise the HWM. This sort of table is a good candidate GT table. Physically, a copy of the table is created afresh for each instance of the process. Thus the HWM is always set to zero at the start of the process.

In the payroll system we observed that the performance of a payroll process, run in isolation, improved when the intermediate result tables were truncated at the start of the process for each month.

Truncate Peformance Problems

All DDL that performs space management (CREATE TABLE, DROP TABLE etc) takes out a lock on the Space Transaction enqueue. Thus, only one session can perform space management at a time, and so DDL that performs space management will serialise. This includes the TRUNCATE command. In Oracle 8.0.5 the truncate command repeated calls an internal function, kcbcxx(), and generates significant quantities of redo log (bug 650614, fixed in Oracle 8.1.4). This degrades its performance, and aggravates the serialisation.

The payroll process in broken in 14 streams. Each stream processes a different set of employees and the 14 calculation processes are run in parallel. In order to reset high water marks, and improve the performance of certain scans, the intermediate result tables were truncated at points within the calculation. This created contention between the streams. In Oracle 8.0, we had found it was faster to delete from the tables than to truncate them, even though this further increased redo logging, and we had to reduce the number of concurrent streams to 7.

After introducing GT tables we could have removed some of the truncate or delete statements altogether. However, truncating a GT table is much faster than truncating a heap table because the space management occurs inside the temporary segment, and the space is not truly free for other segments. These truncates were no longer a problem, so we reverted to truncate. Thus, we retained the ability to run the same code on heap tables for debugging purposes in the development and test environments. Overall performance increased further when we went back to 14 streams.

Quirks

As I have already said, GT tables are *new* in Oracle 8.1, I have encountered one genuine bug, and discovered some slightly suprising behaviour.

If you truncate a PRESERVE ON COMMIT table with the REUSE STORAGE option, then the nothing happens. The table is not truncated, any data remains in the table, and no error is raised (Oracle Bug 1396741). This bug does not apply to DELETE ON COMMIT because the truncate implies a commit. I encounted this because our payroll process truncated the intermediate result tables with the REUSE STORAGE option, and I had simply replaced heap tables with GT tables. The process then crashed with a unique constraint error.

With a PRESERVE ON COMMIT table, you cannot drop it if DML has been performed on it in any current session, even if all transactions are committed. You get ' *ORA-14452: attempt to create, alter or drop an index on temporary table already in use*', all the sessions must be terminated before the object can be droppd. The message itself is misleading because I was trying to drop the table, and there was no index involved. This behaviour does not occur with DELETE ON COMMIT tables, because the DDL implies a commit. When converting a process to use GT tables, beware of performing DDL (other than truncate) on GT tables. Some processes create and drop working storage tables on-the-fly so that the table names are distinct. This approach is no longer necessary with GT tables, and all processes can share the same table name.

If your default tablespace is a temporary tablespace (not that it ever should be) you will be unable to create a permanent object unless you specify a permanent tablespace. You will also be unable to create an index on a global temporary table, because you will still get '*ORA-02195: Attempt to create PERMANENT object in a TEMPORARY tablespace*', and you cannot specify a tablespace because you get '*ORA-14451: unsupported feature with temporary table*'. While this is not a problem, it does indicate how this new feature has been bolted onto the existing CREATE TABLE command.

Benefits

Minimal Code Change: No code change was made to the payroll calculation in order to implement GT tables. Although another process which issues the DDL to build the sets of intermediate result tables was changed.

Reduction in Redo Logging: Overall, redo logging generated by the payroll process reduced by 40% due to GT tables alone, and 60% in combination with the truncate bug fix on upgrading to 8.1.6. Execution time reduced by 67%.

High Water Marks: GT tables naturally start with a reset HWM. This improves full scans on them and their indexes.

Truncate Table: Where it is still necessary to truncate a GT table, it will be even faster than on a heap table.

Optimiser: Use of GT tables does not force use of the cost based optimiser.

Drawbacks

You cannot pass data between processes via a GT table. There they are not suitable for use in multi-threaded processes, such as an application server.

During development or debugging it is frequently useful to see data written by a process. If this is written to GT table it will be lost when the process terminates. With the payroll process it was possible to switch between GT and heap tables in specific streams.

It is not possible to capture cost based optimiser statistics for a global temporary table with the ANALZYE command. It be may be necessary to hint queries that reference GT tables, or explicitly set statistics with the DBMS_STATS package.

Conclusion

GT tables helped me, in my situation, and with my particular set of circumstances. They treated the symptom rather than addressing the cause. I would avoid using them in an initial design. It is better to design the data model so that you don't need to store temporary data in the database, and/or to use working storage variables. However, if it's too late, or too expensive to change the design, then GT tables can be a powerful performance tuning technique.

David Kurtz is an independent consultant with more that 11 years experience of Oracle. He specialises in the design and performance tuning of applications that run on Oracle databases, and is currently the chairman of the UKOUG Unix SIG. He can be contacted on 07771-760660, or e-mailed at david.kurtz@go-faster.co.uk. A presentation of the same title was given to the RDBMS SIG in February 2000 and can be downloaded from the UKOUG website (www.ukoug.org), and the Go-Faster website (www.go-faster.co.uk).