Guide to Using SQL: Sequence Number Generator

A feature of Oracle Rdb

By Ian Smith
Oracle Rdb Relational Technology Group
Oracle Corporation

The Rdb Technical Corner is a regular feature of the Oracle Rdb Web Journal. The examples in this article use SQL language from Oracle Rdb V7.1 and later versions.

Guide to Using SQL: Sequence Number Generator

There have been many requests for Oracle Rdb to generate unique numbers for use as PRIMARY KEY values. In Rdb 7.1 we chose to implement two models that capture the functionality of most SQL database systems on the market and reflect the current planning for the next SQL Database Language Standard, informally called SQL:200x.

This paper describes the sequence database object. A related feature, the IDENTITY attribute, can also be used to derive unique values. This is described in a companion paper entitled *Guide to Using SQL: Identity Columns*.

CREATE SEQUENCE

The sequence functionality is based on the SQL language and semantics of the Oracle RDBMS. The current draft SQL database language standard describes a sequence object very close to that implemented in Oracle Corporation's database products.

The problem solved by sequences is common to many database applications. These applications need to assign unique values to rows stored in tables. Often these values are used directly as PRIMARY KEY and hence FOREIGN KEY values, or possibly the application requires synthetic data to force uniqueness on index key data.

How did we live without sequences?

To understand the motivation for the implementation of sequences, and to better understand the benefits of this feature we will examine some problems areas in current applications and later examine how sequences address these problems.

Most application programmers used intuition to solve the common problem of deriving unique values in a concurrent application system.

There are many ways to get unique values; here are several ways that we will compare with the sequences solution.

(a) Use the MAX function to determine the currently used maximum value and then increment that value for the new row.

```
SQL> insert into ORDERS_TABLE (…, ORDER_NUMBER, …)
cont> values (…, (select max(ORDER_NUMBER)
cont>                    from ORDERS_TABLE) + 1,…);
```

The immediate problem is that now we embed new logic in the application to derive the unique sequence number. Certainly the new AUTOMATIC column support in Rdb 7.1 can help here, as the computation can be stored once in the table definition. As we will see shortly we also have a transaction scoped solution that can lead to duplicates values generated by this technique.

(b) Older applications probably resorted to using triggers to maintain this sequence number. Requirements differ between applications but something like this trigger definition has been observed in several application systems.

```
SQL> create trigger ORDER_TABLE_SEQ
cont>   after insert on ORDERS_TABLE
cont>   (update ORDERS_TABLE
cont>    set ORDER_NUMBER =
cont>            (select count(*) from ORDERS_TABLE)
cont>    where DBKEY = ORDERS_TABLE.DBKEY
cont>   ) for each row;
```

Now using this approach it doesn't matter what is inserted in the table, the trigger overwrites all values. A unique index on the column may cause problems because it will be checked on the inserted value before the trigger is executed. This has implications for PLACEMENT VIA INDEX storage maps, index contention, and I/O to the index.

(c) The *Oracle Rdb7 Guide to Database Design and Definition* states:

"For example, assume that the following trigger has been defined to calculate the next sequence number to be assigned (by adding 1 to the count of orders):"

```
SQL> create trigger SEQUENCE_NUM_TRIG
cont>    after insert on ORDERS_TABLE
cont>    (update SEQ_TABLE
cont>      set SEQ_TABLE.NUMBER =
cont>        (select count(*) from ORDERS_TABLE) + 1)
cont>    for each row;
```

This example assigns the next sequence number directly from a secondary table (SEQ_TABLE). After the insert is complete a trigger is used to maintain a high-water mark in this sequence table. The assumption here is that there is just one row in this table, which unfortunately now becomes a bottleneck for the application.

(d) The previous solutions can be enhanced so that each concurrent updater is allocated a range of values, or possibly use multiple rows in SEQ_TABLE to spread the contention.

In each case we have now moved the complexity into the application that must now manage these value ranges, special tables, and locking semantics.

Transaction Scoped Solution

Any solution that is tied to the current transaction will be problematic. These queries will not see uncommitted data and so will likely derive the same values as other concurrent users. This can lead to a “duplicate value” error when the index is updated, or perhaps a similar constraint failure.

The secondary table solutions can avoid this problem but using PROTECTED WRITE access to the SEQ_TABLE requires a SET TRANSACTION statement with a reserving clause. As we have already noted this single row table is a bottleneck and all transactions would be serialized by the protected access to the SEQ_TABLE.

A mechanism outside the current transaction is required for optimal performance. For instance, an order number server could be created that would issue sequence numbers using a separate transaction. The applications could communicate with the server via mailbox or shared memory and an external function could be written to interact with the server. In this way the sequence can be incremented outside transaction scope and be immediately visible to all concurrent users. Having the server commit immediately eliminates the locking bottleneck.

I know of at least one customer who took this approach initially, but didn't want to maintain their server in the long term since it complicated their own application development and maintenance.

The SEQUENCE Object

Certainly a better mechanism is needed. When examining the requirements for such functionality it became clear that Rdb needed to meet these goals:

1. It should be easy to manage and if possible completely managed within the database system.

2. It should be efficient; with low contention between competing users and low I/O.

3. It should be resilient. If an application, process or system fails we shouldn't return duplicate values after resuming operations.

4. It should provide uniqueness, and a large range of values.

5. It should tolerate database creation, meaning that SQL EXPORT and IMPORT, RMU/LOAD and /UNLOAD and user reload applications should be possible and new values generated after the reload should remain unique.

In Rdb the database administrator creates a named object called a SEQUENCE. The definition language is very simple with a limited number of optional clauses. Consider this example:

```
SQL> create sequence DEPT_ID;
```

The defaults inherited by this simple definition are probably adequate for most applications. By default this sequence will start at 1, and increment the values by 1 up to the largest BIGINT (64 bit integer) value supported.

Each sequence can be granted access control information, so that they can be referenced, or altered and dropped. The supported privileges are: SELECT – that allows the sequence to be referenced within a query, ALTER – to change the sequence attributes, DROP – to delete a sequence, and DBCTRL – to grant or revoke privileges for the sequence.

The sequence name is used to fetch the value of the sequence, using two special suffixes, called pseudo columns, which either force the return of a new value (NEXTVAL), or return the current value (CURRVAL) for the current session.

CREATE SEQUENCE in detail

This section reviews each of the clauses that can be specified by the CREATE and ALTER SEQUENCE statements.

All the clauses for create sequence are optional. Rdb will default values for the options to create a workable sequence.

Each sequence must be given a name that is unique within the sequences name space. The syntax used by SQL to reference a sequence in a query uses the sequence name followed by a pseudo column, for example: SEQ.NEXTVAL. This reference could be mistaken for a table and column reference. Therefore, sequences, tables and views share the same name space[1].

INCREMENT BY

The NEXTVAL function returns a sequence by adding the increment to the previous value. Therefore, a positive INCREMENT BY defines an ascending sequence, and a negative value

[1] There is only one case where a table and sequence have the same name and that is when an **identity** column is defined for a table. Please reference the companion paper entitled *Guide to Using SQL: Identity Columns* for further discussion.

makes it descending. This clause also defines the step size between values. For example, to generate only even numbers[2] use INCREMENT BY 2 with a starting value of 2.

MINVALUE and MAXVALUE

These clauses specify the minimum and maximum values that can be returned from the sequence. If these clauses are omitted then Rdb applies default values based on the INCREMENT BY clause; an ascending sequence starts at one and increments up to the greatest allowed **bigint** value[3], a negative sequence starts at –1 and decrements to the least allowed **bigint** value.

For example if a company defined company cost centers as always having three digits then the sequence might look like this example:

```
SQL> create sequence COST_CENTER
cont>           minvalue 100
cont>           maxvalue 999;
```

We expect that most Rdb users would generate value ranges that could be legally assigned to one of the integer types. Therefore, a shorthand was added to allow easy definition of these ranges. The MINVALUE and MAXVALUE clause specifies one of the target data types **tinyint**, **smallint**, **integer**, or **bigint**. The following example uses the shorthand to specify the full range of integer values for the **customer_id** column that will be defined as **integer**.

```
SQL> create sequence CUSTOMER_ID_SEQ
cont>           minvalue integer
cont>           maxvalue integer;
SQL> show sequence customer_id_seq
    CUSTOMER_ID_SEQ
 Sequence Id: 2
```

[2] If you choose a **cycle** sequence and the incorrect **maxvalue** and **minvalue** odd values might be returned when the sequence cycles.
[3] The current implementation reserves a small amount at the upper and lower end of the range to allow for overflow testing.

Initial Value: -2147483648
Minimum Value: -2147483648
Maximum Value: 2147483647
Next Sequence Value: -2147483648
Increment by: 1
Cache Size: 20
No Order
No Cycle
No Randomize
Wait

The shorthand eliminates the need for the database designer to know the range limits for the various integer data types.

Rdb provides defaults if these clauses are omitted or the NO form of the clause is used. e.g. NOMAXVALUE. The NO form doesn't mean unlimited, just that the user specified no value.

START WITH

At first this clause may see redundant, as surely the MINVALUE clause provides the same purpose? However, this clause can be useful when adding sequences to existing databases. When an older scheme was in use by the application (choose one from the examples in the previous section) and has already consumed some values from the legal range this clause can be used to skip those consumed values. MINVALUE and MAXVALUE are used to specify the legal range but START WITH would initiate the sequence usage within that range so that previously generated values would not reappear. This arrangement is most important when a CYCLE sequence is generated.

Consider our example, if we know that the old scheme generated cost center values less that 245, we would start the new sequence using that upper limit.

```
SQL> create sequence COST_CENTER
cont>           start with 245
cont>           minvalue 100
cont>           maxvalue 999;
```

IMPORT uses this functionality to resets the sequence to start at the next unused value from the old database thus avoiding overlapped sequence values. RMU Extract outputs the last used value as a new START WITH clause to assist in creating a database copy.

If this clause is omitted it will default to the MINVALUE.

CACHE or NOCACHE

CACHE is used to control the efficiency of the sequence. The default for CREATE SEQUENCE is CACHE 20. This means that Rdb will update the root file only when 20 calls to NEXTVAL have been performed. The larger the cache the fewer times the database needs to be updated.

Using the NOCACHE clause disables caching. In this case each NEXTVAL reference will cause an update to the database root file.

ORDER or NOORDER

Selecting the ORDER option ensures that each NEXTVAL request across the cluster will be synchronized so that usage is chronologically ordered. This is achieved by having all sessions share a single cache for this sequence. Each reference to NEXTVAL will involve a lock request to claim a new value. The session that finds the shared cache empty will be assigned the task of refreshing the cache from the Rdb root file.

NOORDER (which is the default) directs Rdb to create a per session cache. In this environment each NEXTVAL allocates a value from the local cache, which involves no lock synchronization. When all values are consumed a disk access is performed to refresh the cache.

The ORDER and NOORDER clause have no meaning if the NOCACHE option is selected.

CYCLE or NOCYCLE

What happens when the sequences has been processed up to MAXVALUE, what happens next?

The default is NOCYCLE that instructs Rdb to return an error as show in this example:

```
%RDB-E-SEQNONEXT, the next value for the sequence "COST_CENTER" is not available
```

If you specify CYCLE then Rdb will return new values starting at the MINVALUE. Obviously, if you use CYCLE then there is no guaranteed uniqueness.

RANDOMIZE or NORANDOMIZE

This option directs Rdb to generate a random a sequence of **bigint** values. The most significant 32-bits of the returned **bigint** value are filled with a random value, the least significant 32-bits are the normally generated **integer** sequence.

This option may allow primary/foreign keys values to be widely dispersed. A wide spread in the key values may reduce node contention for insertions within large SORTED indices.

This option is not compatible with ORDER, MAXVALUE and MAXVALUE.

COMMENT IS

As with all Rdb objects you can add a multiline comment to describe the sequence. The comment can be altered using: the ALTER SEQUENCE ... COMMENT IS statement or, the COMMENT ON SEQUENCE statement.

WAIT, NOWAIT, and DEFAULT WAIT

These clauses control the wait mode on the lock Rdb uses a lock to synchronize access to the sequence values.

The DEFAULT WAIT option requests that Rdb inherit the wait mode from the current transaction. That is, if your transaction statement had looked something like: **set transaction read write wait** then the sequence lock would also use WAIT mode, and similarly if the transaction had requested NOWAIT.

However, if you specify WAIT or NOWAIT then these settings will be used instead of those specified by the transaction. The default behavior is WAIT. Oracle recommends using the WAIT option in most cases. During refresh of the cache the synchronization lock is held for only a short time so waiting is a reasonable option.

Database Design and Tuning

This section describes the impact of using sequences on the database and application environment.

Limits and Allocations

When you create a database in release V7.1 or later Rdb automatically allocates a single structure that initially holds thirty-two sequences. **RMU Convert** also adds this structure to every converted database.

Sequences are managed as part of the Rdb root (.rdb file). Thus the ALTER DATABASE statement is used to expand space in the root. The clause RESERVE … SEQUENCES is used to expand the number of sequences available in the database. When a sequence is dropped that space is reused by the subsequently created sequences. However, once this structure is expanded it cannot be reduced without a rebuild of the database.

The sequences portion of the Rdb root file is called the CLTSEQ or Client Sequence block and is sized in multiples of 32 sequences. Therefore, all RESERVE SEQUENCES clauses will have the final total rounded up to a multiple of 32.

Several Rdb features use sequences and if there are no free sequences then some DDL operations may fail. These features currently include CREATE PROFILE and CREATE ROLE … NOT IDENTIFIED statements.

Locking Considerations

Each referenced sequence requires two locks to manage; one for the metadata, and one for the runtime distribution of new values. As previously described the WAIT/NOWAIT attributes will affect the access to the distribution lock.

The metadata lock is initially requested in SHARED READ so that many users of the sequence can share the metadata. Most ALTER SEQUENCE operations cause this lock to be requested in PROTECTED WRITE mode to prevent incompatible changes to the on-disk structure to those definitions held in memory by other sessions.

The runtime lock is used to communicate with other users who are also using the sequence. It is primarily used as a gatekeeper to the sequence itself, and depending on the options selected may also hold information on what next values can be delivered to the application.

There are really three classes of sequences:

1. NOCACHE
   This option disables caching which will require each NEXTVAL for the sequence to update the root file (CLTSEQ structure) as each value is reserved and returned to the application.

2. CACHE and ORDER
   Used together these options cause a single cache to be shared across the OpenVMS cluster. This is less costly, in terms of I/O to the root file, than NOCACHE since the cache size will determine how many NEXTVAL references occur before a disk I/O to the root will be required. The default CACHE size is just 20 values that may not be adequate for a system with many concurrent applications requesting NEXTVAL for the sequence.

3. CACHE and NOORDER
   These are the default options. In this class each session has a private cache, i.e. it is independent from all other sessions. This setting has lower lock contention, but probably allows more lost values but is the least costly in I/O. In this class it is possible for one session to allocate a cache and use it slowly while another session requests and loads many values. Therefore, the values returned from NEXTVAL in one session could be very distant from the values in another session.

Lost Sequence Values

A CACHE and NOORDER sequence will allocate a full cache to each session. When a DISCONNECT occurs then any remaining cached values (i.e. those that have not been returned with NEXTVAL) will be lost. Rdb does not attempt to reuse these values because it is quite possible that the sequence has been progressed by a different session.

Any values consumed by the application are not recovered by ROLLBACK. This includes values used by statements that fail such as when a constraint fails, a SIGNAL SQLSTATE statement is used, a trigger ERROR condition is raised, a BEGIN ATOMIC block is aborted, or an explicit ROLLBACK is executed. Apart from being very difficult to manage, Rdb does not know how the sequence values are used. For instance, the application may use these values to identify rows in an RMS file and so ROLLBACK of the database transaction may not be relevant in this context.

Transaction reusable servers such as SQL/Services allocate a single cache per sequence for each executor. Each user of the service will share that cache but will still maintain their individual CURRVAL. NEXTVAL will return the next available value and the cache will be shared by all transactions that use that executor process.

CACHE and ORDER will use a single cache for the cluster. As long as some session has the metadata loaded for the sequence the cache will be available. The maximum loss will be those cached values remaining when the last user disconnects from the database.

Therefore, in an environment that runs applications that often attach and disconnect it would be prudent to start a single detached process that loaded the sequence (a seq.CURRVAL would be sufficient[4]) and then hibernated. This would preserve the cache for future users and reduce the lost sequence values because of partial cache usage.

The *Oracle Rdb V7.1 New and Changed Features Manual* claims the maximum loss will be less than the CACHE value. However, if the SET FLAGS 'SEQ_CACHE(n)' option is used then there could be more or less lost. The documentation will be corrected in a future version of the *Oracle Rdb SQL Reference Manual*.

[4] The select will raise an exception because **nextval** has not been referenced. However, the side effect will be that the sequence metadata will be loaded.

Tuning Suggestions

NOCACHE requires root file I/O on each NEXTVAL reference. Therefore - place the root (.RDB) file on a fast, low contention disk for high activity environments.

ORDER shares a cache between several processes – increase the cache size on active systems so that the contention is on the runtime value lock instead of the root file.

NOORDER uses a cache per session - adjust cache size or tailor cache size per session. For example, when loading large volumes of data the defined CACHE size can be overridden to reduce root file I/O using the RDMS$SET_FLAGS logical name, or by executing the SET FLAGS statement in SQL. The option SEQ_CACHE allows the cache size to be specified.

```
SQL> SET FLAGS 'SEQ_CACHE(10000)';
```

This setting must be made prior to the first access to the sequence in the session. If this cache has the ORDER option enabled then each time this session refreshes the cache it will use the changed cache size. The cache size cannot be set to a value less than 2, and this flag has no affect on NOCACHE sequences.

The following example uses **RMU Load** to load millions of rows. The application developer can reduce the root file I/O by increasing the cache size for this RMU Load run. We assume that some **automatic** column, **identity** column, or a columns DEFAULT will execute the NEXTVAL for the sequence.

```
$ define/user_mode rdms$set_flags “SEQ_CACHE(10000)”
$ rmu/load/commit_every=600 mydatabase mytable bulkload.dat
```

Monitoring

When a sequence has been created you can use the SHOW SEQUENCE statement to display the attributes, some of which will have been defaulted by Rdb. In addition the **RMU Dump Header** command will display the CLTSEQ block with the values of the next unused sequence value. In

Rdb V7.1.2 you can select the sequence related data using the /HEADER=SEQUENCE option. The system table RDB$SEQUENCES includes a COMPUTED BY column that fetches the next unused value from the CLTSEQ structure.

Within RMU Show Statistics you can monitor the activity on the CLTSEQ (Client Sequence) Object. This will indicate the activity of all sequence users. To watch the locking activity look at the "Client Lock" and “Stall” messages screens. Client locks are named using the first four characters of the name[5], and include a sequence-type if in the second longword (hex 19).

```
Data: '……EMPL' 4C504D450000000200000001000000190000000 55
                                (incr by) (seq #)(objtyp)(client)
Meta: '……EMPL' 4C504D450000000100000001900000055
                                 (seq #)(objtyp)(client)
```

These locks will be granted in NL to indicate that the sequence is in use, or in EX to indicate that the lock value block is being updated with a new cache range.

RMU Verify will check that the CLTSEQ structure in the root file is consistent with the RDB$SEQUENCES table. See the Rdb V7.1.1 Release Notes for details.

Frequently Asked Questions

These questions and answers may repeat information already presented in the preceding sections.

The values assigned to my table have gaps in the sequence, why is that? The default action for a sequence is to cache 20 values when the first NEXTVAL is used in the session. If the application uses only a few values each session when the unused cache values will be discarded. If the default cache is too large you can use **alter sequence** to change the cache size, or if a few sessions need smaller cache sizes then use the **set flags** statement to establish a new cache size for the session using the SEQ_CACHE option.

Can I completely eliminate lost values for my sequence? This is not possible with the current implementation in Rdb. If you rollback a transaction the used sequences are not returned to the cache, nor are unused values in the cache returned on **disconnect**. If you want finer control over sequences then you will have to include that in your application using a sequence server.

[5] This assumes Rdb V7.1.0.4 or later.

Is it possible to see repeated values from nextval? Rdb is designed to avoid this in the **nocycle** sequences by always saving the next unused sequence value in the Rdb root file for use by other processes. However, it is possible to see repeated values using the **cycle** option.

Note: a synchronization problem did exist before Rdb V7.1.1 that occasionally caused the next unused sequence to revert to a lower value when multiple processes were updating the client sequence (CLTSEQ) block in the root file at the same time. This problem has been resolved for Rdb V7.1.1. and Oracle recommends using this version or later when using sequences.

Why do I get an error when I use currval prior to using nextval? **Currval** only returns the value most recently fetched from the sequence. You must execute **nextval** at least once in the session to provide a value for **currval**.

Can I see what the highest value is across the cluster? Each session using sequences will fetch its own cache of values, once fetched that range of values becomes private to the process. Rdb doesn't track the highest value fetched by other processes in the OpenVMS cluster.

However, you can query the system table RDB$SEQUENCES for the column RDB$NEXT_SEQUENCE_VALUE which holds the next available value. Repeated queries will observe different values as other processes advance the sequence. This is true even if the **isolation level** is **repeatable read** or **serializable**, because this column is a COMPUTED BY column which executes an internal function to get the sequence number directly from the CLTSEQ structure.

When I change the cache size using SEQ_CACHE are other sessions affected? If the cache is a **noorder** sequence then there is no interference. When this session requires a new range of sequences it will allocate the range size as specified by the **set flags** statement (or **rdms$set_flags** logical name). However, if the sequence is an **order** sequence then the cache is shared across the OpenVMS cluster. Therefore, when this process has the job of refreshing the cache it will do so using a different size.

When I use SHOW SEQUENCE it displays '(none)' for MAXVALUE, what does that mean? It means that you specified **nomaxvalue** or omitted the **maxvalue** keyword from the create sequence statement. Rdb will apply the maximum allowed value for a sequence – in Rdb this will be a 64-bit integer value. The same is true for **minvalue**.

I need to use sequences for integer columns. Will I get an error trying to assign a bigint value? Rdb will automatically convert between the **bigint** and **integer** data types. An error would result if the value in the sequence was larger than that supported by the **integer** data type. You can avoid

this by assigning **minvalue** and **maxvalue** ranges that are assignable to integer. SQL allows you to use the **integer** keyword when creating or altering the sequence for just such a purpose.

We must sometimes perform an EXPORT and IMPORT of our database, how does this affect sequences? The SQL EXPORT DATABASE statement saves the sequence definitions as they exist in the database, this includes saving the value from RDB$NEXT_SEQUENCE_VALUE. This value represents a value for the sequence that has not been used by any application. IMPORT DATABASE uses this value as the new START WITH value in the new database. Therefore, all new references to the sequence will result in values not currently stored in the tables of the imported database.

Oracle Rdb
Guide to Using SQL: Sequence Number Generator
May 2003

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
www.oracle.com

Oracle Corporation provides the software that powers the internet.