Oracle SQL and PL/SQL Bad Practices

We learn wisdom from failure much more than from success.
—Samuel Smiles

This is a catalogue of bad practices in Oracle SQL and PL/SQL development. The aim of this document is to serve as a check-list for anti-patterns and to raise awareness about some common errors in Oracle database development.

Oracle PL/SQL is a powerful and feature-rich programming environment. As with any such complex programming environment, there are pitfalls that should be avoided, but recognising them and knowing how to solve the problem often requires years of experience and burning your fingers a few times. Due to data-driven nature and highly concurrent execution environment of Oracle databases, the side effects of such bad practices may not appear for a long time, but when they do the consequences are dire, often leading to data corruption and long nights spent in panic-mode support.

This document contains the patterns of bad PL/SQL code that I have repeatedly found in various applications and databases. Hopefully, it will help other people learn from my mistakes and the mistakes of developers whose code I have reviewed.

For each bad practice, I provided a list of symptoms in the code, an explanation why it causes problems and a list of preferred solutions. I have also listed exceptions to the rules, when they exist. One of the four severity levels is assigned to each bad practice, identifying the danger:

- Risk of data corruption — writing the code like this will probably lead to loss of data
- Makes system harder to use – writing the code like this will prevent reuse or make it harder for clients to develop on top of it; it may also make the system harder to maintain
- Reduced performance – writing the code like this will cause sub-optimal performance and waste resources (typically CPU or storage)
- Security risk – writing the code like this introduces a security hole in the system, risking sensitive data to be exposed to unauthorised users.

Since my experience with Oracle is mostly based around online transaction processing systems, often fronted by a Web server, this list is by no means complete. My goal with this catalogue is primarily to start a discussion about similar recurring issues that other people have noticed. That discussion should lead to a more complete catalogue of common recurring problems that can be used as a check-list for code reviews or given to junior database developers so that they can learn from our mistakes.

Please provide feedback and help make this list more complete. You can do that by sending information on similar bad practices you have noticed or commenting on issues I have outlined in this document. I would especially like to hear from you if you have a better solution for any of the issues or if you do not agree with any of the explanations or potential problems.

Currently there is no dedicated mailing list or web site for this effort — if this document succeeds in starting a discussion on bad practices, I expect that such a site or mailing list will follow shortly. For now, please provide your feedback either by sending an e-mail to gojko@gojko.com, or by visiting http://gojko.net/effective-oracle and submitting your comment there. If you wish to be notified about updates to this list, drop me an e-mail and I will keep you posted, or monitor the RSS feed on http://gojko.net.

Gojko Adzic

http://gojko.net

Using non-deterministic functions directly in conditions

Severity: Reduced performance

Symptom

Non-deterministic function that does not depend on current row values (or depends on a subset of row values) is used in Where clause. Examples might be functions that fetch data from a referential table depending on current database context, or perform some calculations on derived data; Functions are not enclosed in select from dual or a subquery.

Why is this bad?

Functions may be executed much more times than required. It might be sufficient to execute the function just once per query, but Oracle might execute the function for each row of the result (or even worse, for each row of the source).

Solutions

- Turn function into (select function() from dual) if it does not depend on any row values
- Move function into a similar subquery if it depends on referential data, and join the subquery in the main query
- Mark function as deterministic if it is by nature deterministic (i.e. always returns same result for same parameter values)
- Use Oracle 11 result caching if possible

See also

http://gojko.net/2007/11/02/speed-up-database-code-with-result-caching/

Catch-all error handling

Severity: Risk of data corruption

Symptom

A catch-all exception block (`WHEN OTHERS THEN`) used to process an error (or a group of errors) in a PL/SQL procedure.

This is done typically either to ignore an expected error which should not affect a transaction (for example, when duplicate key in index should be disregarded, or if a problem with inserting should be quietly logged).

Catch-all error handling might also be written when a single block of code is expected to throw several types of exceptions (no data found, duplicate value on index, validation…) and the developer wants to handle them all at once.

Another example is using `WHEN OTHERS` block to catch domain exceptions thrown by `RAISE ERROR` because they cannot be individually checked.

Why is this bad?

- Catch-all block will prevent other unexpected errors from propagating. For example, a mutating table error caused by a trigger may be hidden with a catch-all block, completing the transaction successfully when it did not really do everything it needed.
- There is often an assumption about the error that can occur, which may be incorrect when a different type of exception is thrown. This may lead to inconsistent data. For example, if a catch-all block is used to insert a record in case of a missing data error, then that record may be inserted on any other error as well.

Solutions

- Do not use `WHEN OTHERS` to check for a particular exception such as `NO_DATA_FOUND`. Check directly for a particular type of exception.
- To handle domain-specific errors, declare your domain exceptions so that you can check for a particular exception later in the code.
- Do not assume that you know all errors that may occur at a given place in the code. Storage problems, mutating tables and similar issues should surface to the client, and not be handled and discarded in an unrelated piece of the code.

Exception

Shielding parts of the system from errors in other parts

See also

Client access as object owner

Severity: Security risk

Symptom

External applications, application servers, web servers or client applications accessing database with user-names/passwords of object owners (schema containing objects).

Why is this bad?

- Schema owners have all privileges on their objects, so this kind of access compromises security and allows SQL injection from web or other application servers to cause significant damage
- Schema owners often need access to other (internal) schemas that would not necessarily be exposed to external applications.

Solution

1. For each DB schema with externally accessible views, tables and procedures, set up a special "client" role, and grant only required access rights to that role.
2. For each client application (or set of related applications), create special "client" users with connect privilege, but without privileges required to create or modify objects in the database.
3. Grant appropriate client roles to client users

This will enable you to control access and minimise security risks from client applications, and you will also easily manage and maintain access to database objects - when adding new views and procedures, you just have to grant required access privileges to 'client roles'

Exceptions

- Small, low-risk internal applications
- Internal batch data processing utilities

Client access to tables

Severity: Security risk

Symptom

External client code (web, applications) reads and writes data directly to tables

Why is this bad?

- You will not be able to modify underlying table structure easily if external code depends on it — especially if not all that code is under your control.
- You cannot fine-grain access control to different columns for different applications
- You are giving clients access to all private data, and they might start using columns which are not intended for external use (tracing, timestamps etc). Getting all external code changed later on, when you decide to drop or modify "internal" columns, might be very difficult, even impossible.

Solution

Build a set of client views for different applications, and expose only required columns — wait for client programmers to ask for a column before putting it into a view. This will allow you to easily modify underlying table, and even replace it with a set of different tables (and possibly instead of triggers) while not modifying any client code.

Embedding complex SQL code into PL/SQL

Severity: Makes system harder to use

Symptoms

- Large chunks of SQL code (typically a complex `Select` statement), used inside PL/SQL function or procedure.
- Using an optimiser hint inside PL/SQL function or procedure.

Why is this bad?

- SQL may not be easy to optimise without affecting clients — since recompiling package body will discard existing package state and connected clients will get an exception next time they call any package method. If package procedures are used in jobs, recompiling package body might require shutting down those jobs in order to obtain package lock. View bodies can always be recompiled without side-effects.
- Pl/SQL code becomes hard to read and understand

Solutions

- Extract any complex SQL code from Pl/Sql and put into views, then select from those views in procedures/functions
- Move optimiser hints to views instead of having them in procedures

Error handling with magic numbers

Severity: Makes system harder to use

Symptom

RAISE_APPLICATION_ERROR used in pl/sql procedures, with arbitrary error codes that conform to some implicit contract with other users/front end developers (no pl/sql exception declaration for that error code).

Why is this bad?

- if there is no string message, errors are not clear - making the code harder to use and maintain
- pl/sql code that uses procedures cannot easily filter/handle only those exceptions

Solution

Declare exception numbers and exceptions in pl/sql package headers, throw these exceptions instead of application errors.

To send a string message along with exception, use RAISE_APPLICATION_ERROR and the appropriate exception code.

When throwing exceptions from stand-alone procedures, create a public package with just exception declarations, use those exceptions from standalone procedures

Exception

In package-private procedures that are not used directly from outside world, when exceptions will be handled internally by other procedures in the same package (although it is a good practice to use declared exceptions even then).

See also

http://www.oracle.com/technology/oramag/oracle/03-may/o33plsql.html

Error handling with output parameters

Severity: Risk of data corruption

Symptom

Procedures or functions declared with return code (and/or error message) output parameters. In case of errors or problems, these parameters get special values that should signal the caller about problems.

Why is this bad?

- Such errors might be ignored unintentionally. Callers might not check the return code and just continue working.
- Client applications can use auto-commit option of database drivers, which means that if there was no exception the database driver will automatically commit after every call. If this option is used, partially complete transactions will be committed, breaking the rule of atomicity.
- Code for using the procedure (both pl/sql and object) must check error status after every call. This becomes quite cumbersome if several procedures should be called in sequence and makes the code very error-prone.

Solution

Use declared exceptions for error handling. Exceptions have to be handled by the caller, so they will not ignore errors unintentionally.

Exceptions allow you to specify both the error code and message at the same time, and do not require output parameters to be specified (leading to a cleaner api)

Exception

Working with legacy client environments that do not support exceptions/do not map database exceptions to host errors. In that case, an additional layer of procedures should be written, which just handles exceptions into error codes/messages, so clients can choose which version to call. Special care should be taken in those legacy client environment not to use auto-committing.

See also

section *"Error handling with magic numbers"* on page 7

Formatting data in views

Severity: Makes system harder to use

Symptoms

- to_char used on date or numeric values in views
- strings concatenated in view code to form pretty printed values

Why is this bad?

- data format cannot be easily modified on the front-end, since some information may be lost
- values cannot be easily modified (i.e. applying time zone shifting becomes much harder)
- filtering based on underlying date or string values becomes much more processor-heavy and requires full table scans and/or substring matching/comparisons.
- internationalisation becomes much harder — instead of translating elements and then combining them, translation engines must analyse and translate/reformat formatted data

Solutions

- Format data on the front-end, not in the database.
- Perform formatting in queries coming from the front end, specifying exactly what the front end needs — but database views should not suppose any specific data format.

Hardcoding local Varchar2 variable size

Severity: Makes system harder to use

Symptoms

- PL/SQL function or procedure declares local Varchar2 variables for temporary storage of table values, with hard-coded length
- Views declared with Varchar2 types with hard-coded length

Why is this bad?

- Code is error prone, because hard-coded values may not allow for enough space to store the entire value coming from a database table.
- Even if the size is correct, if the underlying type ever changes, errors such as `ORA-06502` 'Character string buffer too small' may start appearing in procedures.

Solution

use `%TYPE` to reference the underlying column type instead of hard-coding the type and size for local variables.

Exceptions

- variables and fields not related to underlying table data
- fields or variables that combine several table fields

Ignoring exceptions

Severity: Risk of data corruption

Symptom

This is a typical example:

```
begin
...
Exception When others then
 NULL;
end;
```

This kind of code is written when errors such as attempts to insert a duplicate record or modify a non-existing row should not affect the transaction. It is also common in triggers that must be allowed to fail without effecting the operation which caused them (best-effort synchronisation with an external resource)

Less frequently, this code is written by junior developers who do not knowing what to do in case of an error, so they just disregard exceptions.

Why is this bad?

Serious errors such as storage problems or table mutations might be hidden from the calling code

Solutions

- If you do not want any errors to affect current transaction, execute the code in an autonomous transaction and log errors to an error table/log table. For critical functions implement some sort of administrative notifications for those errors. For low priority functions, check the log table to periodically for errors.
- If you want to ignore certain exceptions, because they can be solved by re-processing, handle only those specific exceptions.

Exception

low-risk functions where any errors can safely be ignored

See also

- section *"Error handling with magic numbers"* on page 7
- section *"Catch-all error handling"* on page 3

Not using bound variables for changing parameters

Severity: Reduced performance

Symptom

Frequently executing the same query with different parameter values, but specifying parameter values literally, without using bound variables.

Why is this bad?

- Database engine will have to compile the query every time and will not be able to cache the statement.
- If care is not taken to prevent SQL injection, may open a security hole in the system.

Solution

For all parameters that are genuinely changing, use a bound variable instead of specifying the value literally.

Exceptions

- Ad-hoc queries that are run only once or infrequently
- Parameters where statement caching is pointless for different values.

See also

section *"Using bound variables for constants"* on page 24

Relying on conditional column predicates in triggers

Severity: Risk of data corruption

Symptom

A trigger uses conditional predicates to check whether a particular column is being updated.

```
if updating('ColumnName') then
  -- do something with column
end if
```

Why is this bad?

it can be misleading. this does not get the job done in most cases where gui screens are used (the original message mentions JSP forms). GUI persistence layers may save all fields, not just the ones that were modified. So there might be a dummy update to the current value, and trigger will need to compare values. Here's a quick test:

```
create table tbl(col1 number, col2 varchar2(10));

CREATE OR REPLACE TRIGGER trig1
BEFORE UPDATE ON tbl
FOR EACH ROW
BEGIN
  if not updating('col1') then
     raise_application_error (-20001,'Col1 was not updated');
  end if;
END;
/

insert into tbl values (1,'test');

update tbl set col1=1;
```

in this case, even though value in col1 stays the same, exception is not raised. updating('col1') only checks whether col1 was present in the set clause of the update, not whether the value actually changes.

Solution

Compare :new.column and :old.column. Make sure to use decode if column is nullable (or make sure to check for NULL properly).

Exceptions

- this does not rule out using a conditional predicate for the whole statement (without specifying column name)
- when you really want to do something even if updated value stays the same.

Relying on context information in package variables

Severity: Risk of data corruption

Symptom

Variables in PL/SQL packages used to store context (session) values between DB API calls. They are typically initialised by calling a procedure explicitly, or using a log-on trigger. Stored procedures rely on these variables being properly initialised and do not check whether they are empty.

Why is this bad?

Packages may be recompiled/revalidated due to dependent structure changes, or modifications — and local variable values will be lost when that happens. At best, this will cause context loss (NULL references when a value is expected), but depending on usage pattern, may cause updates or deletes of arbitrary records (if commands depend on package variables having a proper value).

Solutions

- use sys_context instead of package variables to store simple cross-call context values (numbers/strings)
- use global temporary tables to store more complex inter-transaction context
- use tables to store complex inter-call context information.

Exception

variables used for lightweight caching (procedures can behave correctly if variables have no value. for example,. try to load configuration if it's not yet loaded)

Relying on external context initialisation

Severity: Risk of data corruption

Symptom

Views and stored procedures use sys_context() to retrieve a global context, depending on that context not being empty. Context is not initialised automatically, typically requiring the caller to initialise it by calling a stored procedure.

Why is this bad?

Due to networking problems connection between web sites and the database can break. Most modern web frameworks will automatically attempt to re-connect, in which case the procedure to re-initialise context will not be called, and the old context will be discarded. That will cause the code that depends on the context to break or produce incorrect results. If the context is used for auditing purposes, the audit log will be corrupt.

DBAs and support staff will need to initialise contexts properly before running queries in order to troubleshoot problems or optimise performance. This is not obvious and most of the time they will get a different, misleading result because the context is not initialised.

Solutions

- Force context initialisation by using a logon trigger.
- Change the code to behave properly even if the context is empty.

Exceptions

- Contexts are used to decide if some action should/should not be performed in a very controlled environment, and when there is no option to control it on the level of username connecting to database (i.e. turning off auditing or synchronisation).
- Tthere is no risk of automatic re-connection (i.e. the client is not using a connection pool).

Secondary processing with unshielded triggers

Severity: Risk of data corruption

Symptom

Trigger used to propagate updates of a table to a secondary subsystem (vertically adding functionality, like external publishing, auditing, etc — not part of the main workflow). Subsystem transfer can happen either synchronously or asynchronously (with a queue/job combination). In any case, problematic symptom is that the trigger does not protect the primary system from secondary subsystem exceptions.

Why is this bad?

Even though the secondary system is not part of the primary system workflow, problems in the secondary subsystem can propagate to the primary system, which defeats the whole point of adding the functionality into a separate subsystem. This can get even more complicated with data replication, as exceptions in publishing triggers can effectively prevent data from being created or updated in the first place.

Solution

Create a log table for the secondary subsystem, and log errors any in the trigger using a `When others` clause. Use `dbms_utility.format_error_stack` or `sqlerrm` to record error details.

Exception

When a problem in the secondary system should genuinely prevent updates in the primary system

See also

section *"Catch-all error handling"* on page 3

Storing ROWID for later reference

Severity: Risk of data corruption

Symptom

ROWID values for references are stored in a table or kept in client code in order to access referent rows easier

Why is this bad?

ROWID values for a row can change. Importing and exporting data, moving tablespaces and similar operations actually modify the ROWID, so storing that value for later reference and then using it to update records may lead to data corruption.

Solution

Use primary keys to reference rows and store references to them.

See also

http://www.orafaq.com/node/993#comment-1981

Storing empty LOBs

Severity: Reduced performance

Symptom

Empty `CLOB` values used instead of `NULL` for `CLOB` fields that do not hold a value

Why is this bad?

Oracle allocates space for EMPTY CLOBs. In tables with large number of empty CLOB columns, this can take up significant storage space.

Solution

Use NULL instead of EMPTY CLOB

See also

section *"Using magic numbers for non-existing values"* on page 27

Too many levels of views

Severity: Reduced performance

Symptom

A large hierarchy of views containing sub-views or subqueries is in place. Such hierarchy is usually established as several layers of abstraction over abstraction, typically when adding new core features to underlying models, but keeping client API for backward compatibility using a set of higher-level views.

Why is this bad?

Optimiser will give up and run full table scans even if indexes could be used after typically 8 or 9 levels of nesting.

Solutions

- Flatten the structure so that it has less than 8 levels. Use joins instead of subqueries where possible
- Use materialised views to cut off a part of the hierarchy
- If materialised views cannot be used for performance reasons, use an aggregated table maintained by triggers to do the same.

Transactional control in non-autonomous procedures

Severity: Risk of data corruption

Symptom

`Commit` or `rollback` statement in a stored procedure or function without `PRAGMA AUTONOMOUS TRANSACTION`

Why is this bad?

Effectively prevents stored procedures from being used in a wider context — rolling back or committing inside a stored procedure will delete/permanently write data that was used in a wider transaction, in the middle of that transaction.

May cause issues that are very hard to trace/debug - you will not be able to check if data was processed correctly when the procedure rolled back. audit logs will contain references to records which, from the logical point of view, never existed.

May cause inconsistent data — since rolling back/committing will split a wider logical transaction into two — one which rolled back and another one which is running, relational constraints might fail in the second transaction. even worse, if the relational constraint checks were not enforced, inconsistent data might be written permanently.

Solutions

- Throw exceptions in case of errors; let the caller decide what to do in case of error. Do noting in case the operation succeeded — let the caller decide if the entire wider transaction is correct or not.
- Add `PRAGMA AUTONOMOUS_TRANSACTION;` to the procedure header to make it run as an autonomous transaction

Exceptions

- Long-running worker procedures such as batch updates (may include suboperations and save-points to store partial results. should be marked as autonomous transaction).
- Auditing (should be done with autonomous transactions)

Trigger depending on order of execution

Severity: Risk of data corruption

Symptom

Triggers relying on other triggers for the same action — examples might be sharing contextual information, assuming that a log/audit record added by another trigger exists.

Why is this bad?

- In general, order of trigger execution is not guaranteed for triggers in the same category and for the same operation. Even if a combination of interdependent triggers works correctly on the development machine, there is no guarantee that it will work on production (or tomorrow). Order can change with a database upgrade or schema import/export, and like other context-dependent errors it is very hard to troubleshoot/diagnose.
- If one trigger is using context information to complete the work of another trigger, data corruption may occur when the order changes.

Solutions

- Encapsulate workflow-dependent business logic in stored procedures, and call those procedures from triggers
- Combine inter-dependent triggers them into a single trigger

Exception

statement triggers are guaranteed to execute after row triggers

Using Sequence nextval without curval

Severity: Risk of data corruption

Symptoms

- Sequence `currval` method used in a procedure or trigger without calling `nextval` first — typically in a trigger that updates a log record, or a procedure that partially processes data
- sequence `currval` used to calculate the next value which will be read by `nextval`

Why is this bad?

- Calling the method will cause an exception if the sequence is not initialised — so the method/trigger depends on the caller to initialise the sequence first
- Procedures relying on someone else to initialise the sequence must be called in a specific context, which limits their reuse
- Triggers may or may not work depending on the order of execution
- If procedure/trigger uses the current sequence value to update the relevant record, calling it in a different context/order of execution may update the wrong record
- Sequences are not transactional — they can be cached or changed in another session between calls to curval and nextval. `currval+1` is not guaranteed to be the next value; using that to predict IDs is very dangerous, as it can lead to wrong records being deleted or updated.

Solutions

- Do not use `currval` to read contextual information. Pass IDs explicitly to procedures or use `sys_context` to store contextual information
- Use `nextval` instead of `currval` where appropriate (if you just need a unique number)

See also

Using a sequence as a counter

Severity: Risk of data corruption

Symptom

A sequence object used in a manner which relies on consecutive numbering without gaps. Often with arithmetic operation on sequences used to guess the next or previous value without actually using

```
sequence.nextval
```

.

Why is this bad?

- Sequences are allocated in batches, and can be used by other sessions in parallel, so there may be gaps between the currently observed sequence value and the one that will actually be generated using nextval.
- Sequences are non-transactional (i.e. rollback will not turn a sequence back to it's old value), so guessing next or previous sequence value may lead to duplicate or inconsistent information.

Solutions

- Always use nextval to get the next sequence value as an atomic operation. If this value is required later on, store it into a local variable.
- If you need a list without gaps, use a programmatic counter (based on the number of rows in a table, not on a sequence).

See also

section *"Using Sequence nextval without curval"* on page 22

Using bound variables for constants

Severity: Reduced performance

Symptoms

- Bound variables used in queries for values that are never changing (often when client developers bind all variables in a query).
- Bound variables used for parameters where actual value can significantly effect the optimisation plan

Why is this bad?

- Optimiser will not use the most efficient plan to execute the query
- If variable peeking is turned on, it might actually chose a completely wrong execution plan for subsequent runs

Solution

For values that are not changing in a query, or where statement caching does not make sense, use a literal and not a bound variable (this does not imply that other genuine parameters in the same query should not be bound).

See also

- section *"Not using bound variables for changing parameters"* on page 12
- http://download-uk.oracle.com/docs/cd/B10501_01/server.920/a96533/optimops.htm#46980
- http://oracletoday.blogspot.com/2005/11/be-careful-about-bind-peek_113283667040369808.html

Using derived column values for existence checks

Severity: Reduced performance

Symptom

count, sum or some other aggregate function (i.e. custom made csv) executed in a subquery, and then compared to 0 or 1 or NULL in the WHERE clause of the outer query in order to check whether at least one (or exactly one) related element in the substructure exists; results of the aggregate function are then discarded (not passed on through the outer view).

The subquery was typically developed first, possibly as a full-fledged view, and then the outer query/view, which filters results of the first view; was written. Less frequently they are in the same view, first one used as a subquery.

```
CREATE OR REPLACE instrument_v AS
SELECT instr.idinstr, instr.name, count(opt.type) AS activeoptiontypes
FROM instrument instr, instrumentoption opt WHERE instr.ididinstr=opt.idinstr (+)
GROUP BY instr.idinstr, instr.name;
...

SELECT idinstr, name FROM instrument_v WHERE activeoptiontypes >0
```

Why is this bad?

Instead of just checking indexes, potentially a large number of records will have to be read and analysed.

Solution

Instead of comparing the results of aggregate function, create a different view that checks directly for existence of the related type — optimiser will be able to execute the plan using indexes and will not have to calculate aggregate functions for irrelevant rows

Using exceptions for flow control
Severity: Risk of data corruption

Symptoms

- Half-successful procedures, throwing exceptions when the work is not totally complete, but a part of it is
- exception when XXX then blocks used to control workflow other than error-handling (exceptions used instead of state-codes)

Why is this bad?

- client code relies on procedure internals (if this exception is thrown, customer data was written to the table correctly, so it can be committed). as clients are expected to commit in case of some exceptions, later changes in procedure internals may cause inconsistent data being written, or wrong data committed.
- Oracle will cancel effects of unsuccessful commands if an exception gets thrown out of the database, even if there is no explicit rollback. If the client code expects partially available results to be stored when they commit, data corruption will occur. See the following example:

```
CREATE TABLE dummy_g(name varchar(255));

CREATE OR REPLACE PROCEDURE TestInsert_G(val IN VARCHAR2, raiseerr BOOLEAN DEFAULT true) IS
BEGIN
     INSERT INTO dummy_g (name) VALUES (val);
  IF raiseerr then
     RAISE_APPLICATION_ERROR(-20400, 'Forced error!');
 end IF;
END TestInsert_G;
/

SELECT count(*) FROM dummy_g;

-- this one will write 1 record

exec TestInsert_G('p', false);

SELECT count(*) FROM dummy_g;

-- this one will not write anything - table will still have only one record, even in the same session
-- although the insert operation was successful and there was no rollback

exec TestInsert_G('p');

SELECT count(*) FROM dummy_g;

commit;
```

Solution

Use exceptions only for errors. Do not count on procedures being half-successful. Procedures should follow the Samurai Principle — return successful or not return at all. In cases when exceptions are used to report half-successful execution, status codes should be used instead (and no exception thrown), or the procedures should be divided into several parts.

See also
http://c2.com/cgi/wiki?SamuraiPrinciple

Using magic numbers for non-existing values

Severity: Makes system harder to use

Symptom

Special numerical or string value used as "not applicable" or "non-existing" in stored procedure parameters, return values or table values. Usually this value is 0 or -1 if parameters are numerical IDs. This is typically done by junior programmers scared of using NULL (may be coming from a front-end environment where NULL operations cause exceptions) or in order to simplify queries (since using NULL in most expressions produces NULL, and special IS NULL syntax has to be used to check for NULL).

Why is this bad?

- Code is error-prone, because it is easy to mix valid and invalid values
- This usage is non-standard, and will make it harder for other people to use your code correctly
- This prevents proper usage of referential integrity on foreign keys

Solution

Use NULL to signal "non existing" and "not applicable", as it was intended.

Exception

Possibly when index on column is required to speed up processing, because NULL is not indexed with B-Tree indexes. (Bitmap indexes cover NULL as well, but they are not applicable to frequently changing tables.

See also

http://www.oracle.com/technology/pub/articles/sharma_indexes.html

Using non-dedicated packages for continuous jobs

Severity: Makes system harder to use

Symptoms

- A continuous or frequent database job executes a long-running procedure from a pl/sql package. That same package is also used for client access or by other schemas.
- The body for a continuous or frequent database job is a procedure from such a package.

Why is this bad?

The package will be locked while the procedure is running. With a continuous or frequently running job, this may require interrupting the job or causing down-time even for the smallest changes like granting execution to other users/schemas to the package.

Solution

Extract procedures for job bodies into dedicated pl/sql packages or into standalone procedures.

Wrapping everything into stored procedures

Severity: Makes system harder to use

Symptoms

- Client access layer consists exclusively or mostly of stored procedures - both for writing and for reading. `Ref Cursor` output parameters of stored procedures used to read data.

  This is typical for SQLServer programmers moving to Oracle, due to earlier SQL Server limitations in terms of access privileges and functionality.

  It is also often done under a false pretext that using stored procedures is more secure or brings better performance. In fact, the same security restrictions can be applied to views as to stored procedures in Oracle. If bound variables and statement caching are used, query access to views also brings pre-compilation benefits, so there is no performance gain.

- Stored procedures are used to fill in additional details when inserting or updating data (automatically calculated columns, timestamps and similar).

Why is this bad?

- Procedural access limits operations to a choice of parameters - i.e. deleting records with specific name. You end up by providing a stored procedure for every possible combination of parameters, or by providing "generic" stored procedures that are no better than allowing direct table access.

- Procedural access is typically done on row-level (procedures work with a single row), which has significant performance penalties over tabular access for operations that work on multiple rows.

- Output `Ref Cursor` parameters do not allow clients to apply further filters or conditions

- Output `Ref Cursor` cannot be easily joined with additional data

- Output `Ref Cursors` data cannot be easily paged on demand

- Packaging data-manipulation steps in a stored procedure does not prevent someone from using the table differently and inserting or modifying data directly. Your rules will only be applied if the clients are forced to use your stored procedures. As the code base grows, and different programmers join and leave the team, this will be harder and harder to enforce.

Solutions

- Use views for reading data, do not use stored procedures; instead of passing filter parameters to stored procedures, expose those values as view columns and allow clients to filter using `Where`. Apply appropriate security restrictions to views.

- Use views for inserting and updating rows if the operation requires dynamic parameters.

- If all required columns in views are not updatable, consider using instead-of triggers to provide the functionality

- Use triggers to perform action such as populating missing columns on data modification on tables. This ensures that the data will be populated regardless of how the table is used.

- Use stored procedures to encapsulate business rules and procedural processing logic; If front-end requires data that's calculated from database column values, rather than stored in them, encapsulate that logic in functions and include functions in views. Procedures/functions should be used to extract common procedural steps from triggers and to simplify triggers and jobs.

Exceptions

- Output `REF CURSOR` objects should be used when filtering view columns for read-only access severely effects performance — for example when subqueries have aggregate functions and filtering outer view

would cause query data to be executed on all data, and then filtering it. In this case, inserting specific values into the subquery could significantly reduce execution time.

- This technique can also be used to provide an uniform API for related reports, where query depends on parameter values or supplied parameters