LuxOUG Virtual Tech Days 20

Handling Errors during Bulk DML operations

Erik van Roon

EvROCS
COMPLETING THE PUZZLE

Who Am I?

- Erik van Roon
- Originally analyst of microbiology and biochemistry
- Oracle developer since 1995, self-employed since 2009
- Most of my work takes place inside the database
- *Playing* with APEX since 2008
- Did several successful major datamigration projects

EvROCS
COMPLETING THE PUZZLE

ORACLE
ACE

: erik.van.roon@evrocs.nl
: www.evrocs.nl
: @evrocs_nl

Background of this talk

Did a number of data migrations:

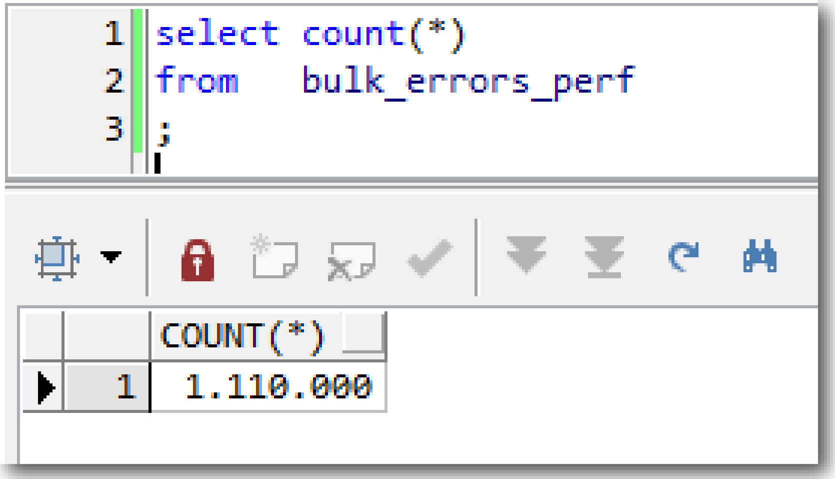
- Merge data of acquired competitor into own database
- Similar data but completely different data models
- Transformation may/will lead to problems
- Errors should be handled gracefully
- An error in a child record sometimes means the parent shouldn’t be present either
- Limited window for executing the migration

In this presentation

Scripts will be mentioned like this → Preparation\create_demo_objects.sql

The scripts need some objects created by ↑

Among which a table based upon sh.customers



```sql
select count(*)
from   bulk_errors_perf
;
```

```
     COUNT(*)
▶ 1  1.110.000
```

When you're done → Cleanup\cleanup.sql

So why
Bulk Opperations?

Row By Row

Not the fastest

But fast

And Full Control

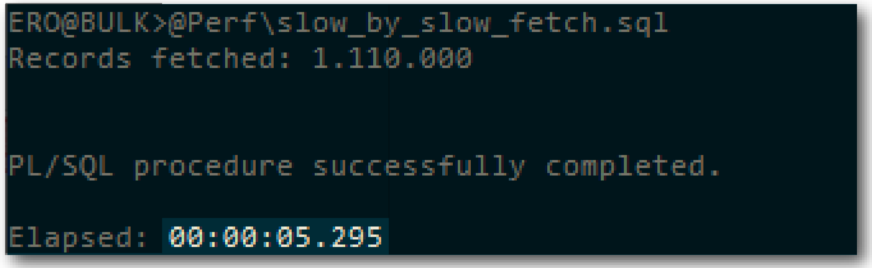
Single SQL Statement

How fast is Bulk Fetching?

Slow-by-Slow fetching

Perf\slow_by_slow_fetch.sql

```
open  c_err;
 loop
   fetch c_err
   into  r_err;

   exit when c_err%notfound;
```

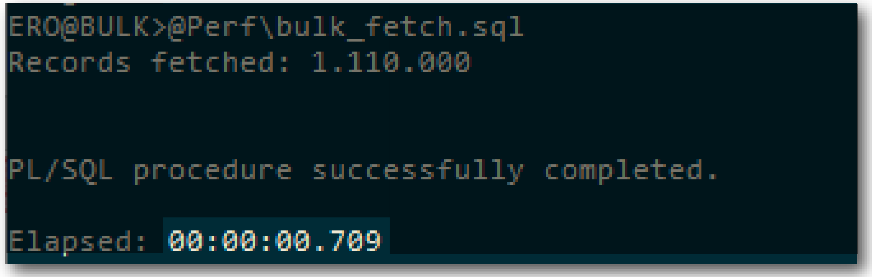


```
ERO@BULK>@Perf\slow_by_slow_fetch.sql
Records fetched: 1.110.000


PL/SQL procedure successfully completed.

Elapsed: 00:00:05.295
```

Bulk fetching

Perf\bulk_fetch.sql

```
open  c_err;
 loop
   fetch c_err
   bulk collect
   into  a_err
   limit cn_bulk_limit;

   exit when a_err.count = 0;
```



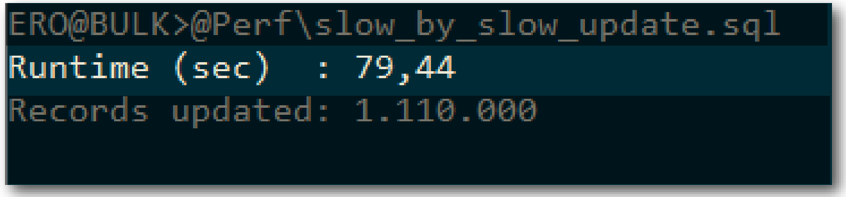
```
ERO@BULK>@Perf\bulk_fetch.sql
Records fetched: 1.110.000


PL/SQL procedure successfully completed.

Elapsed: 00:00:00.709
```

How fast is Bulk Data Manipulation?

Slow-by-Slow update

Perf\slow_by_slow_update.sql

```
for i_err in 1 .. a_err.count
   loop
     update bulk_errors_perf
     set    cust_last_name = trim(cust_last_name)
     where  rec_id         = a_err(i_err).rec_id
     ;
   end loop;
```

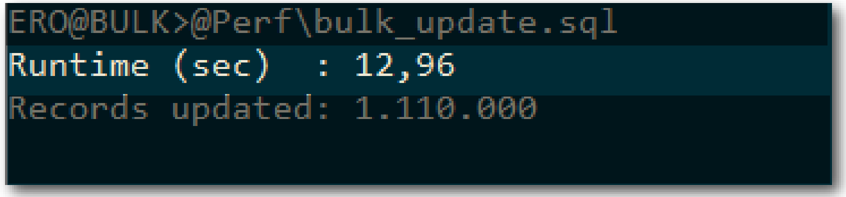


```
ERO@BULK>@Perf\slow_by_slow_update.sql
Runtime (sec)  : 79,44
Records updated: 1.110.000
```

Bulk update

Perf\bulk_update.sql

```
forall i_err in indices of a_err
     update bulk_errors_perf
     set    cust_last_name = trim(cust_last_name)
     where  rec_id         = a_err(i_err).rec_id
   ;
```



```
ERO@BULK>@Perf\bulk_update.sql
Runtime (sec)  : 12,96
Records updated: 1.110.000
```

But what if .....

Error Handling

During Bulk Fetching…

None

During Bulk Data Manipulation…

~~No exception handling per row~~

No savepoint to rollback to

Luckily there are Save Exceptions and Log Errors

Save Exceptions

Saves the exceptions until all iterations of the forall are processed

```
declare
  type l_tst_aat is table of some_table%rowtype index by pls_integer;
  l_tst_aa     l_tst_aat;
begin

  [....]

  forall i_tst in 1 .. l_tst_aa.count save exceptions
    insert
    into   some_table
    values l_tst_aa(i_tst)
  ;
end;
/
```

And then….

ORA-24381: error(s) in array DML

ORA-24381

This can be handled in an exception handler

```
declare
  e_bulk_errors    exception;
  pragma exception_init(e_bulk_errors, -24381);
begin
  [...]

  forall [...]  save exceptions
    [...]
  ;
exception
  when e_bulk_errors
  then
    [...]
end;
/
```

Pseudocollection

Pseudocollection sql%bulk_exceptions is available

"Composite attribute that is like an associative array of information about the DML statements that failed during the most recently run FORALL statement"

Two attributes:

Error_index

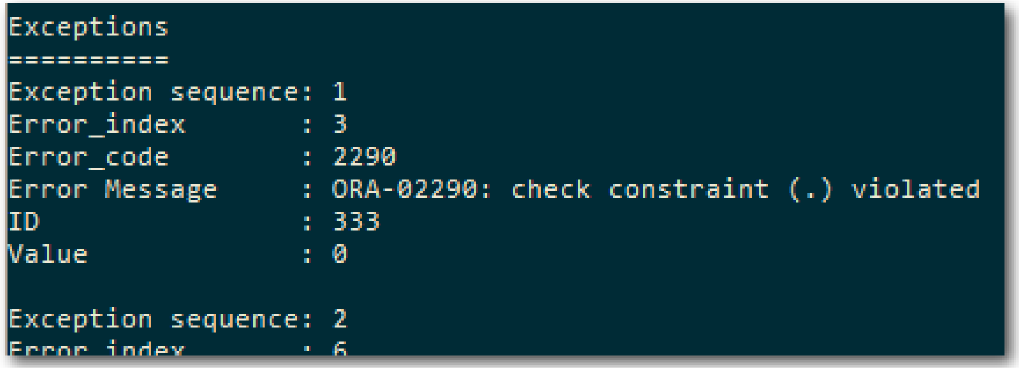
The number of the DML statement that failed

Error_code

The Oracle Database error code for the failure

Handling the exceptions

Pseudocollection identifies which statements in forall raised which exception.



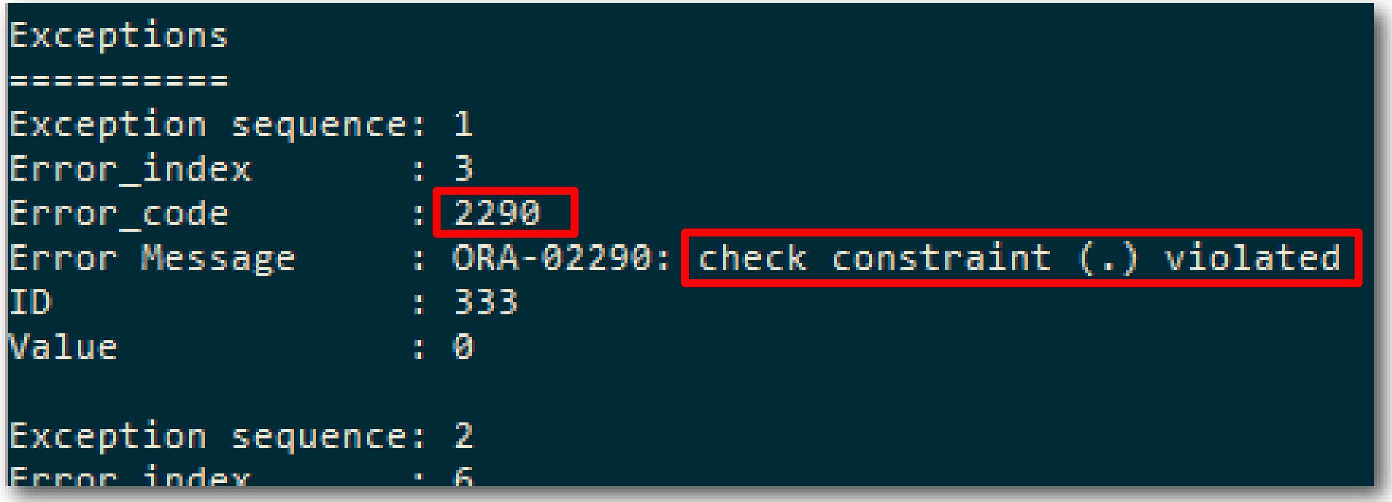
```
Exceptions
==========
Exception sequence: 1
Error_index        : 3
Error_code         : 2290
Error Message      : ORA-02290: check constraint (.) violated
ID                 : 333
Value              : 0

Exception sequence: 2
Error_index        : 6
```

```
exception
   when e_bulk_errors
   then
     dbms_output.put_line ('Exceptions');
     dbms_output.put_line ('==========');

     for i_err in 1 .. sql%bulk_exceptions.count
     loop
       l_error_index := sql%bulk_exceptions(i_err).error_index;
       l_error_code  := sql%bulk_exceptions(i_err).error_code ;

       dbms_output.put_line ('Exception sequence: '||i_err);
       dbms_output.put_line ('Error_index        : '||l_error_index);
       dbms_output.put_line ('Error_code         : '||l_error_code);
       dbms_output.put_line ('Error Message      : '||sqlerrm(-1 * l_error_code));
       dbms_output.put_line ('ID                 : '||l_val_aa(l_error_index).id);
       dbms_output.put_line ('Value              : '||l_val_aa(l_error_index).value);
     end loop;
```

SaveExceptions\dense_collection.sql

Notice:

- That the error code is a positive number
- How we may lose information (depending on the exception raised) because we only have the error code:
  ORA-01476: divisor is equal to zero ✓
  ORA-02290: check constraint (.) violated ✘



```
Exceptions
==========
Exception sequence: 1
Error_index       : 3
Error_code        : 2290
Error Message     : ORA-02290: check constraint (.) violated
ID                : 333
Value             : 0

Exception sequence: 2
Error_index       : 6
```

Sparse collections

Remember

error_index is “The number of the DML statement that failed“

For sparse collections

error_index <> index of erroneous record

error_index = x means

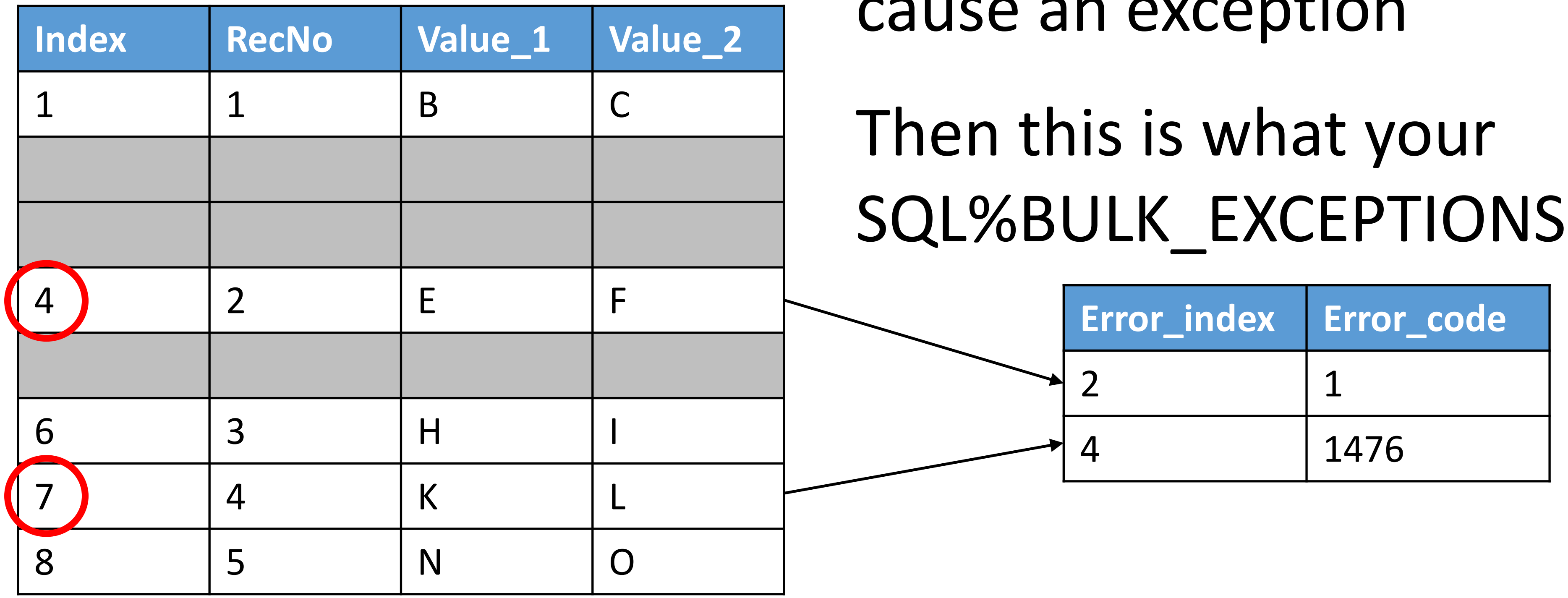
The x-th record in the original collection raised an exception

Not: the record at index x

SaveExceptions\sparse_collection.sql

If index 4 & 7 in this collection cause an exception

Then this is what your SQL%BULK_EXCEPTIONS will be:



Sparse Collection

Index	RecNo	Value_1	Value_2
1	1	B	C
4	2	E	F
6	3	H	I
7	4	K	L
8	5	N	O

Error_index	Error_code
2	1
4	1476

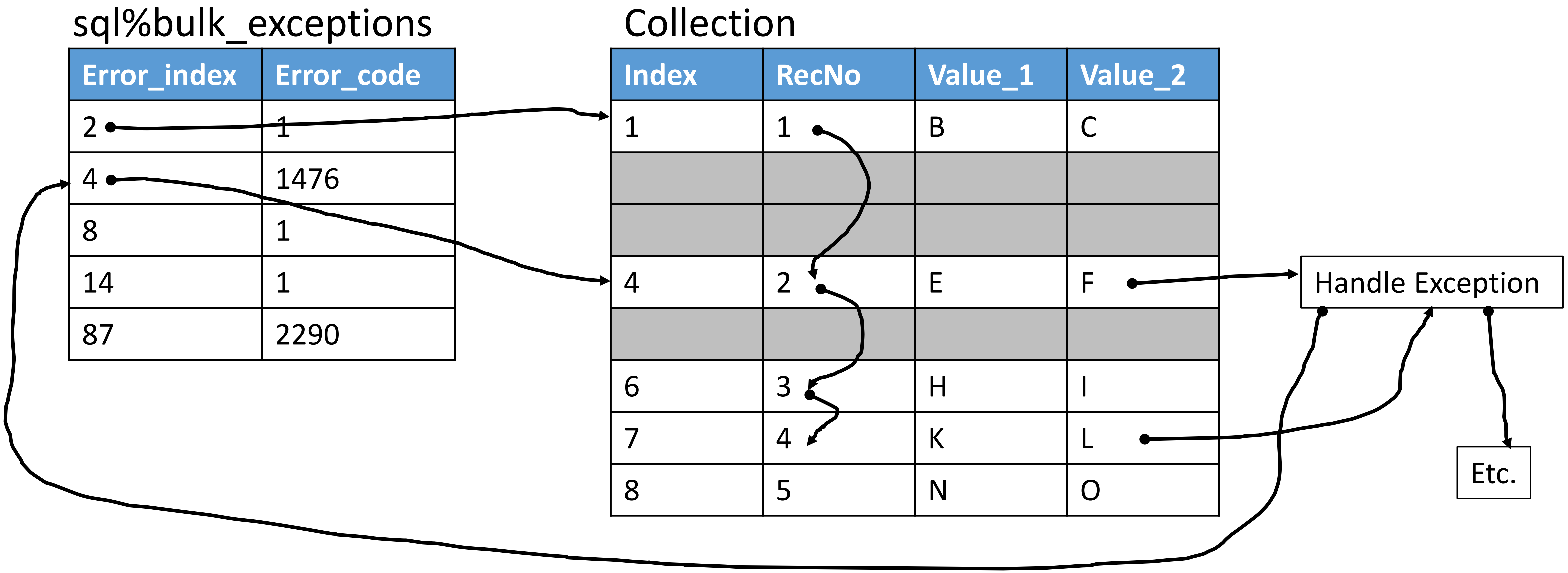
error_index, solutions

Loop through original collection
    counting to error_index-th record

Make the sparse collection dense again

Avoid sparse collections,
    mark rejected records using an extra status attribute

Solution “count”



sql%bulk_exceptions

Error_index	Error_code
2	1
4	1476
8	1
14	1
87	2290

Collection

Index	RecNo	Value_1	Value_2
1	1	B	C
4	2	E	F
6	3	H	I
7	4	K	L
8	5	N	O

Handle Exception

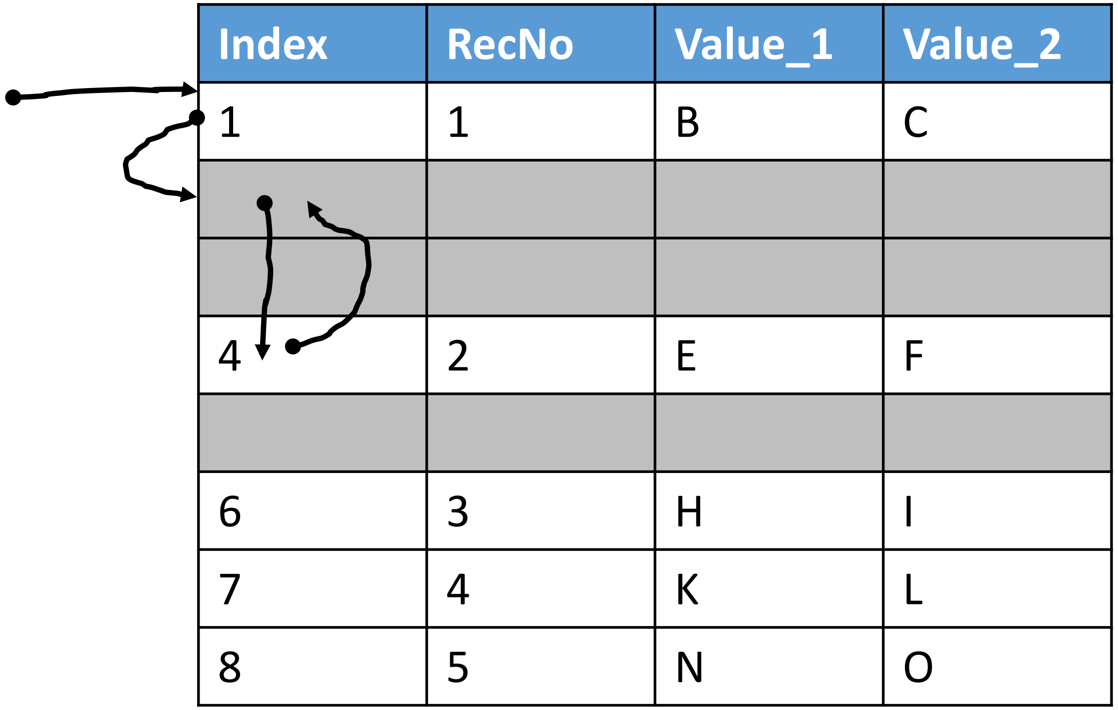
Etc.

disadvantage: extra loop through the collection (though only if there *are* exceptions)

SaveExceptions\sparse_collection_count.sql

Solution “make dense”

Collection



Index	RecNo	Value_1	Value_2
1	1	B	C
4	2	E	F
6	3	H	I
7	4	K	L
8	5	N	O

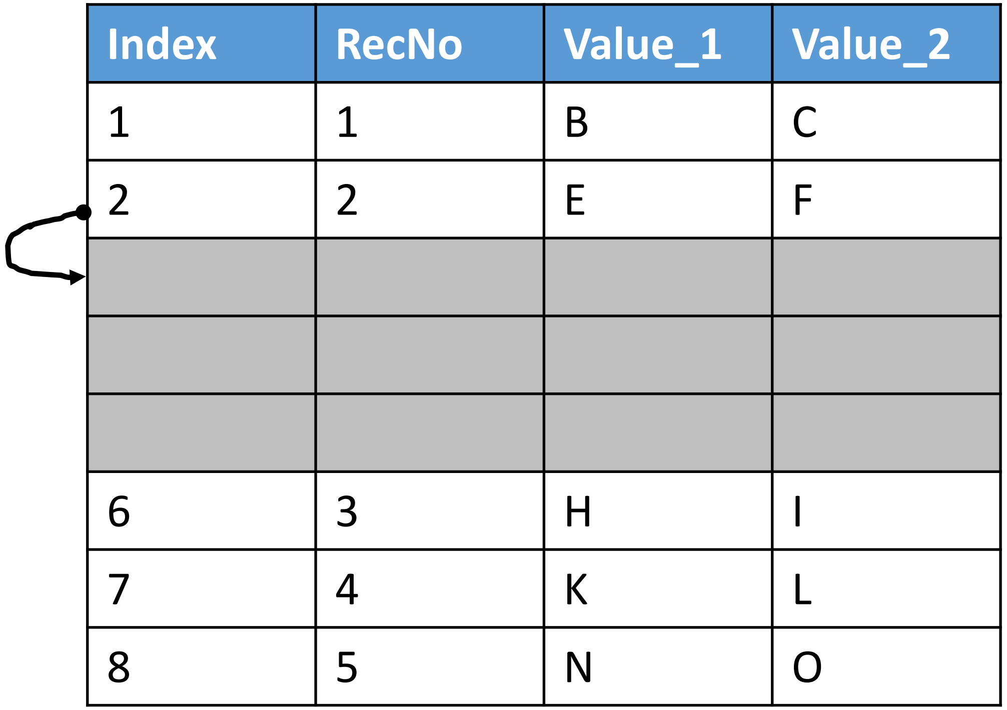
Solution “make dense”

Collection

Index	RecNo	Value_1	Value_2
1	1	B	C
2	2	E	F
4	3	E	F
6	4	H	I
7	5	K	L
8	6	N	O

Solution “make dense”

Collection



Index	RecNo	Value_1	Value_2
1	1	B	C
2	2	E	F
6	3	H	I
7	4	K	L
8	5	N	O

Solution “make dense”

Collection

Index	RecNo	Value_1	Value_2
1	1	B	C
2	2	E	F
3	3	H	I
7	4	K	L
8	5	N	O

Solution “make dense”

Collection

Index	RecNo	Value_1	Value_2
1	1	B	C
2	2	E	F
3	3	H	I
4	4	K	L
8	5	N	O

Solution “make dense”

Collection

Index	RecNo	Value_1	Value_2
1	1	B	C
2	2	E	F
3	3	H	I
4	4	K	L
5	5	N	O

Process this dense collection

disadvantage: extra loop through the collection

Either always, just before the forall

Or only do this, just before exception handling

SaveExceptions\sparse_collection_to_dense.sql

Solution “use status”

Don’t throw away records you don’t need/want but rather set the status indicator

Index	RecNo	Value_1	Value_2	Status / Action
1	1	B	C	
2	2	BBB	CCC	
3	3	BCD	DCB	
4	4	E	F	
5	5	EEE	FFF	
6	6	H	I	
7	7	K	L	
8	8	N	O	

Solution “use status”

Don’t throw away records you don’t need/want but rather set the status indicator

Index	RecNo	Value_1	Value_2	Status / Action
1	1	B	C	OK / Process
2	2	BBB	CCC	NOK / Reject
3	3	BCD	DCB	NOK / Reject
4	4	E	F	OK / Process
5	5	EEE	FFF	NOK / Reject
6	6	H	I	OK / Process
7	7	K	L	OK / Process
8	8	N	O	OK / Process

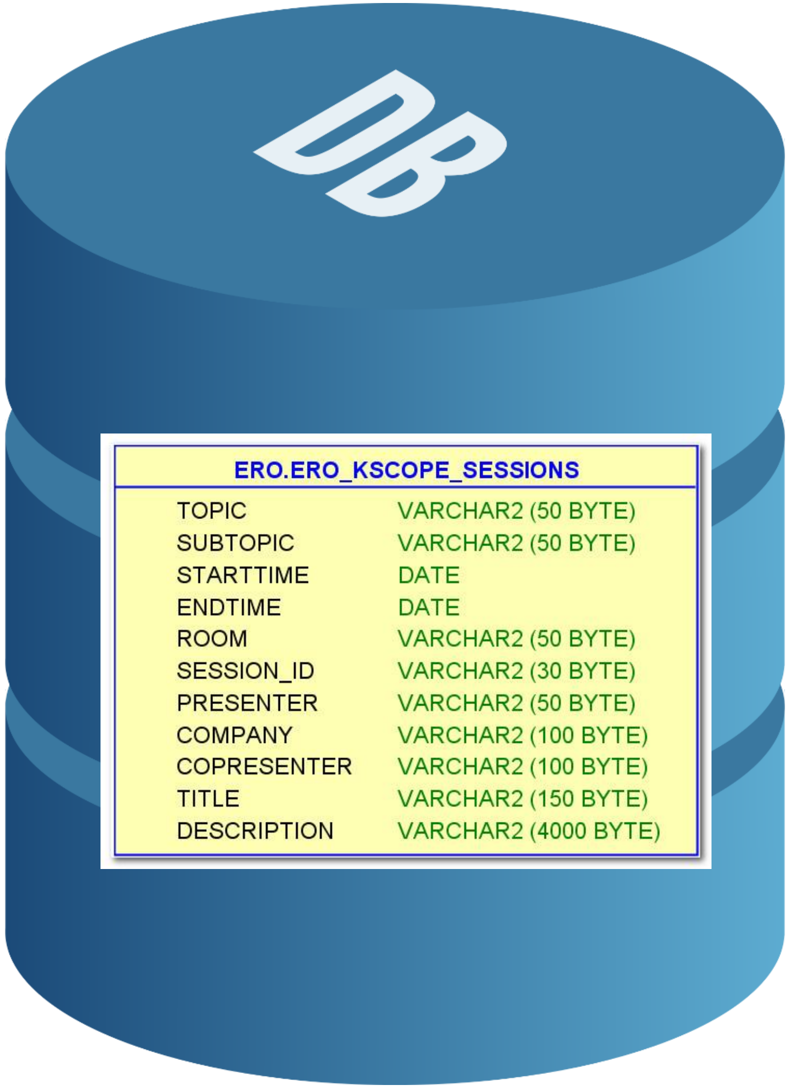
Process this dense collection,
DML only for ‘OK’ records

disadvantage:
DML statements for rejected records that won't do anything
(fast per record, but ‘unnecessary’)

SaveExceptions\sparse_collection_status.sql

But is your database....

as simple as this one?



ERO.ERO_KSCOPE_SESSIONS

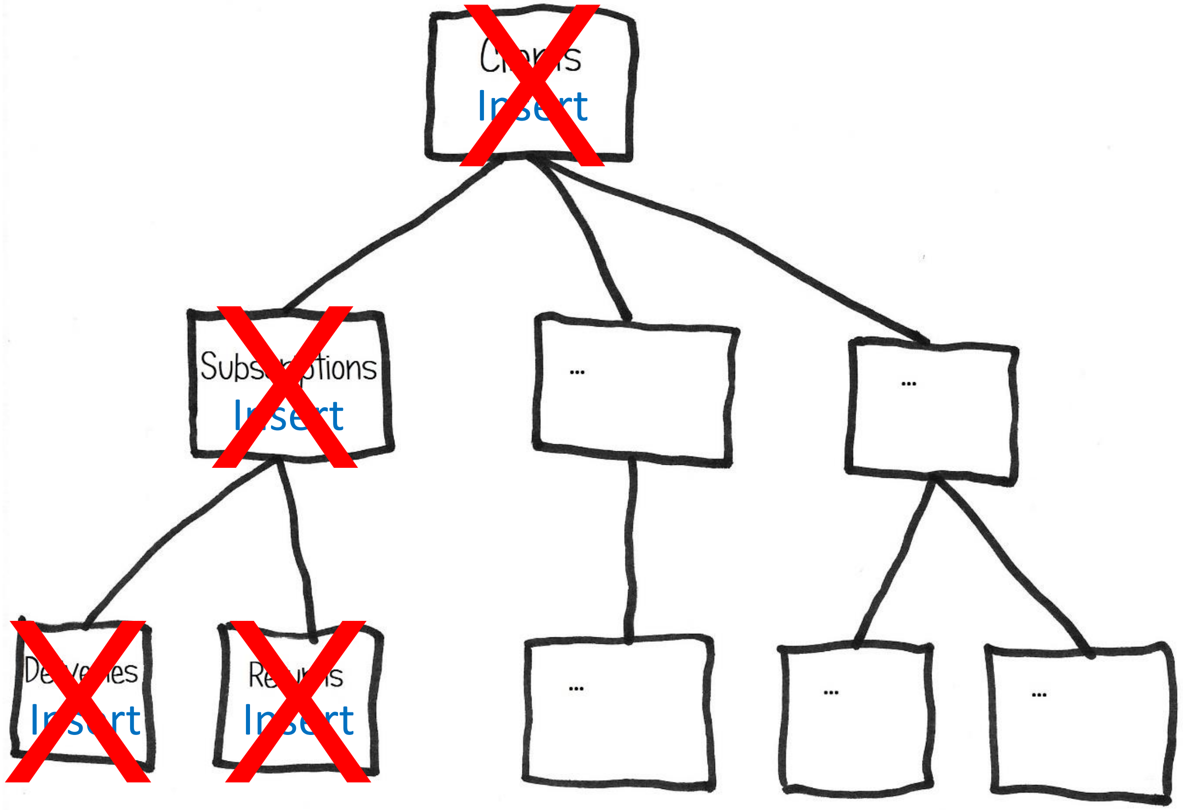
TOPIC	VARCHAR2 (50 BYTE)
SUBTOPIC	VARCHAR2 (50 BYTE)
STARTTIME	DATE
ENDTIME	DATE
ROOM	VARCHAR2 (50 BYTE)
SESSION_ID	VARCHAR2 (30 BYTE)
PRESENTER	VARCHAR2 (50 BYTE)
COMPANY	VARCHAR2 (100 BYTE)
COPRESENTER	VARCHAR2 (100 BYTE)
TITLE	VARCHAR2 (150 BYTE)
DESCRIPTION	VARCHAR2 (4000 BYTE)

Multiple Foralls

So, we want to prepare data

for several tables and

do a forall-insert for each



What if an error anywhere in the tree means the entire tree for that master record must be rejected?

Save Exceptions – Multiple Foralls

Option 1: Foreign Keys

Temporarily change the FK’s to “Cascading Delete”

Usually not an option for live databases

Only works if the tables actually have FK’s between them

Logging of these ‘extra’ rejections hard/impossible

Requires good administration of original states of FK’s and
correctly reestablishing these states, whatever happens

Save Exceptions – Multiple Foralls

Option 2: Add the PK/UK of its parent to each collection

For each bulk exception:

- Determine PK/UK of the parent
- Delete all rows with that parent-PK in table for this collection
- If parent has other children set by step delete entire tree below
- Delete the parent record
- Determine PK/UK of the parent of this parent
- Etc, etc, etc

Gets really ugly, really fast

Save Exceptions – Multiple Foralls

Option 3: Create a Metadata-Collection

Create an extra collection (preferably nested table)

- Map each index of each collection in the tree to the PK/UK of top-level parent

For a record that causes an exception

- Get PK/UK of top-level parent from the Metadata

For each table in the tree

- From the metadata determine the rows belonging to the top-level-parent
- Delete every record belonging to the top-level-parent

Metadata Example

Nested table for mapping could be like:

```
create or replace type mapping_ot force as object
 (table_name          varchar2(30)
 ,collection_index    integer
 ,top_level_pk        varchar2(50)
 ,table_pk            varchar2(100)
 );
/

create or replace type mapping_ntt force as table of mapping_ot
/
```

Metadata Example

Mapping Nested Table

Tablename	Collection Index	Top Level PK	Collection PK
Clients	1	ABC	ABC
Clients	2	KLM	KLM
Subscriptions	1	ABC	S001
Subscriptions	2	ABC	S002
Subscriptions	3	ABC	S003
Subscriptions	4	KLM	S004
Deliveries	1	ABC	D101
Deliveries	2	ABC	D653
Deliveries	3	KLM	D871

If delivery at collection index 2 fails,
we can query the Nested Table to instantly find out the PK of its top-parent (ABC)

Then we can easily identify for any table the PKs that belong to the same top-parent (ABC)
by querying the Nested Table again

Metadata Example

Deleting the records for the same top-level PK/UK:

SaveExceptions\MultiForall.sql

```
Suppose PK ‘R-1534’ in table RETURNS raised an exception

delete
from    subscriptions
where   subscriptions_pk in
        (select to_remove.table_pk
         from   table (mapping_nt)   erroneous
         join   table (mapping_nt)   to_remove
           on   to_remove.top_level_pk  = erroneous.top_level_pk
         where  erroneous.table_name    = 'RETURNS'
           and  erroneous.table_pk      = 'R-1534'
           and  to_remove.table_name    = 'SUBSCRIPTIONS'
        );
```

Well, not *completely* different: LOG ERRORS

What is LOG ERORS?

DML-statement clause

Logs errors in table with structure:

```
Name                                 Null?    Type
------------------------------------ -------- ---------------
ORA_ERR_NUMBER$                               NUMBER
ORA_ERR_MESG$                                 VARCHAR2(2000)
ORA_ERR_ROWID$                                ROWID
ORA_ERR_OPTYP$                                VARCHAR2(2)
ORA_ERR_TAG$                                  VARCHAR2(2000)
[COLUMNS THAT NEED TO BE LOGGED]              VARCHAR2(4000)
```

Available since 10.2

Columns to be logged are all Maximum-length character datatype (or RAW)

LOG ERORS, create log table

Creation of logtable:

Manually (obey required structure!)

Or have Oracle do it for you

```
dbms_errlog.create_error_log
  (dml_table_name        in varchar2,          -- table to create log-table for
  ,err_log_table_name    in varchar2 := null   -- name for log-table
  ,err_log_table_owner   in varchar2 := null   -- owner for log-table
  ,err_log_table_space   in varchar2 := null   -- tablespace for log-table
  ,skip_unsupported      in boolean  := false  -- include unsupported datatypes?
  );
```

If you're using **extended datatypes**, you may want to do it **manually**.
Having each column in the table represented in the error table by a varchar2(32767) might not be what you want.

Parameters for create_error_log

err_log_table_name

If omitted: `'ERR$_'||[substr(tablename,1,x-5)]`

Where x = maximum tablename-length

(<= 12.1: 30

>= 12.2: 128

)

Resulting tablename may obviously not already exist, or:
ORA-00955: name is already used by an existing object

Parameters for create_error_log

err_log_table_owner

If omitted: the currently connected user

err_log_table_space

If omitted: The default tablespace of err_log_table_owner

skip_unsupported

If table contains columns with unsupported datatypes
True: those columns won’t be in log table
False: ORA-20069: Unsupported column type(s) found

unsupported datatypes are:
Long, *LOB, Bfile, Abstract Data Type(ADT)

So, how do we log the errors?

Add the LOG ERORS clause to the end of the DML statement

```
log errors [into logtable-name]
           [(‘Tag')]
           [reject limit integer|unlimited]
```

LOG ERORS, components

`[into logtable-name]`

Names the table in which logging is to be inserted

If no into clause is used, again table name

```
'ERR$_'||[substr(tablename,1,x-5)]
```

is implied

LOG ERORS, components

`[(‘Tag')]`

Is a value for column ORA_ERR_TAG$ in the log table

If not supplied ORA_ERR_TAG$ column will be null

Can be used to identify the log records for this statement

LOG ERORS, components

```
[reject limit integer|unlimited]
```

If **more than** errors than this occur, the entire statement fails

- All errors are still logged into the log table
- The exception raised = that last error that occurred

Default is 0, so just 1 error will crash the statement

Example

A statement could look like this:

```
insert
into   some_table
       (id
       ,first_column
       ,second_column
       )
select some_sequence.nextval
,      some_column
,      some_other_column
from   some_other_table
log errors
  into err$_some_table
  ('my insert’)
  reject limit unlimited
```

LogErrors\log_errors_example.sql

Some advantages over save exceptions are

Works for each and every DML statement, not just FORALL

We have the entire error message, not just error number

We (can) have every value of every column as it arrives at the table,
not just the ones we have in our statement
(for example, also the values supplied by triggers and defaults)

Some things to be aware of

No automatic clean up
Table grows over time
‘Old’ errors may contaminate your query if not sensibly tagged

Multiple error tables on single table possible
Can be just what you want/need.
Can also lead to developers all creating their own error table

The errorlog table is in no way ‘connected’ to the data table
So dropping or altering one has no effect on the other

Some things to be aware of

Identification of errors of ‘last DML performed’ only by TAG
So sensible tagging is essential to retrieve the correct errors

Log table is not session specific
Unless it’s a Global Temporary Table (see later)

Structure of log table needs to be kept in sync with original
No errors if not, but you may lose information (see later)

The logging is committed inside an autonomous transaction

Statement will always complete successfully (if within reject limit).
To know if records where rejected you need to query the Log table.

Log table can also be a Global Temporary Table (GTT)

Must be created manually

Must be defined as “on commit preserve”
or the commit in the autonomous transaction will remove your logging

Possible advantages to a GTT log table

All visible logging is done by ‘your’ session
So TAG only needs to identify statements in this session

Cleanup automatically at end of session

Table can be truncated during a session without hurting other sessions

Disadvantage to a GTT log table

If error logging needs to be preserved, manually insert it into another table is needed

Identification based on TAG is tricky

Procedure name is going to be the same in the next call

Session id’s are being reused

Date-time strings may not be unique in multiuser environment

One possible solution

Use SYS_GUID to generate a Globally Unique Identifier at the start of a transaction and use it to identify its logged errors

What if structure of original table differs from log table?

- **Column added to or dropped from original table**

  No problem, only columns present in both tables are logged

  After also adding a new column to the log table it’s also logged

- **Order of columns differ**

  Again, no problem. Columns are logged as expected

- **Column in log table smaller than data in DML statement**

  Statement crashes with
  ORA-38906: insert into DML Error Logging table “[.]" failed
  followed by the error message that was attempted to be logged

Multiple statements

What if we have the same situation we discussed for Forall:

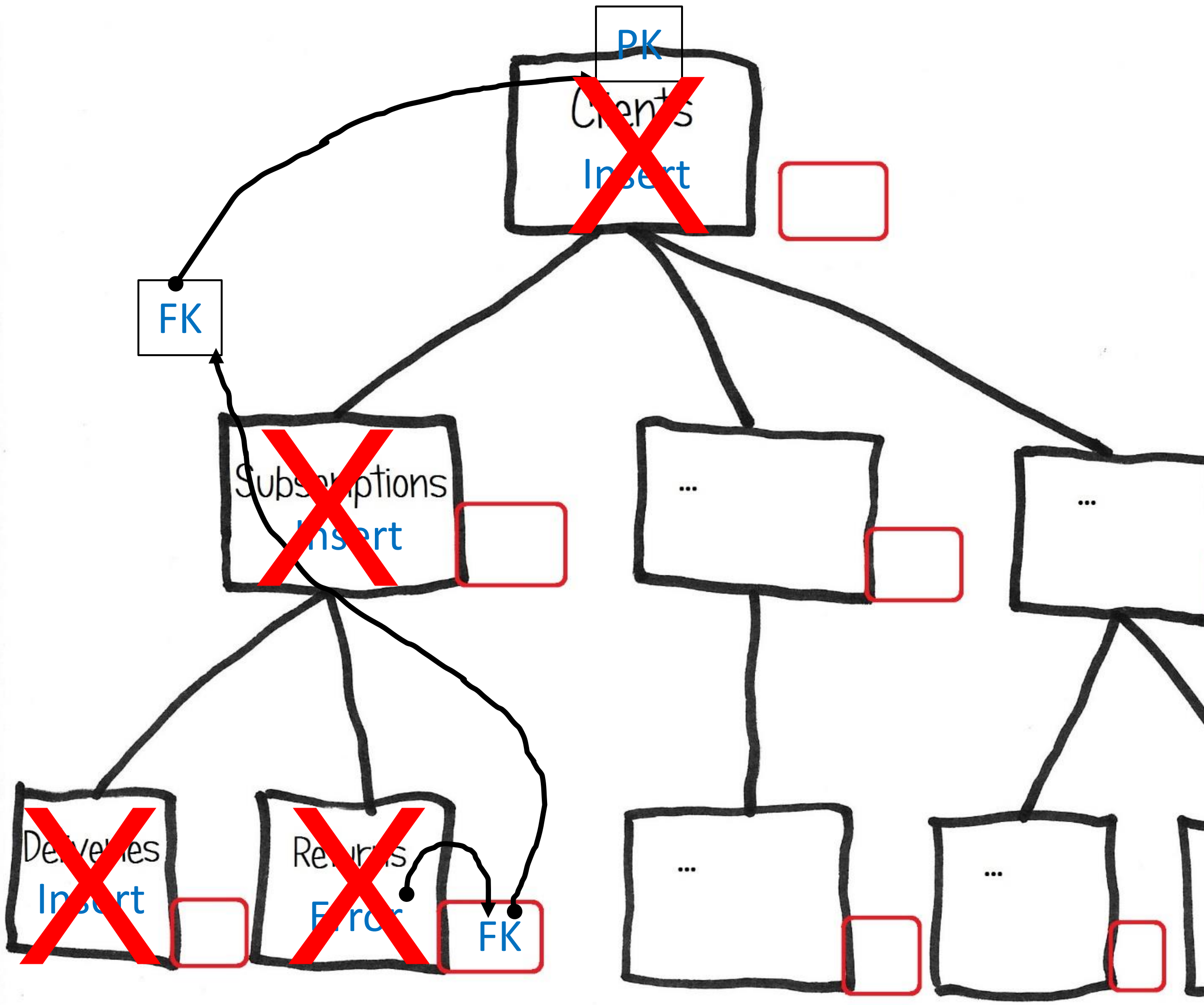
Several tables, and if anywhere in the tree a record fails, the whole tree for the top-level record it belongs to must be rejected?

Multiple statements

We could:

- Get the FK of erroneous records from errorlog table
- Get the PK of the top-level record
- Based on the top-level PK delete the entire tree

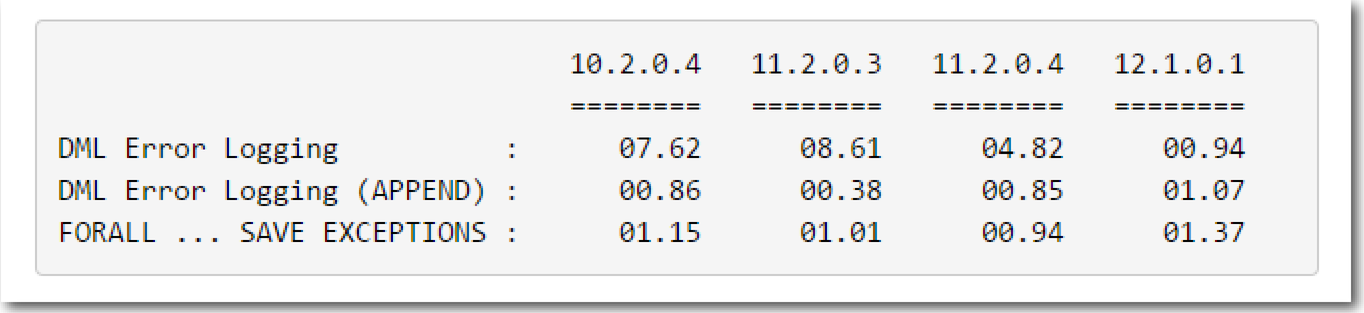
LogErrors\MultiInsert.sql



Performance

(Stolen from Oracle-Base, Tim Hall)

Test of insert of 100.000 rows with 2 errors on different database version (on different servers):



```
                           10.2.0.4  11.2.0.3  11.2.0.4  12.1.0.1
                           ========  ========  ========  ========
DML Error Logging          :  07.62     08.61     04.82     00.94
DML Error Logging (APPEND) :  00.86     00.38     00.85     01.07
FORALL ... SAVE EXCEPTIONS :  01.15     01.01     00.94     01.37
```

times are in seconds

https://oracle-base.com/articles/10g/dml-error-logging-10gr2

Summary

- Log Errors gives the actual error message
  Save Exceptions only gives the error code
- Log Errors also stores the actual data
  Save Exceptions only has pointers (which are tricky for sparse collections) to data in another collection
- Errors Logged by Log Errors is persistent
  Save Exceptions errors are volatile
- Log Errors can be used for the ultimate bulk operation: a single DML statement
  Save exceptions can only be used for Forall statements

Summary

- With Save Exceptions, correcting executed DML in a tree of tables requires extra collections and extra plsql coding.
  With Log Errors Just some extra DML statements are needed
- With Log Errors the statement always succeeds
  (when errors <= reject limit)
  Save Exceptions raises an exception that can be handled
- Log Errors makes you work to identifying the errors caused by the last statement
  Save exceptions only has the errors for the last statement
- Before 12c performance of Log Errors may be bad

“Stupid questions do exist.
But it takes a lot more time and energy to correct a stupid mistake than it takes to answer a stupid question, so please ask your stupid questions.”

a wise teacher who didn't just teach me physics

Thanks !!