Allianz Global Assistance

1. Introduction

The question of exception handling is something that most data warehouse projects will run in to at some point. The more transformation logic is built into the data logistical layer (the ETL) - to handle data quality issues in the organization and to cater for the data behaviour of the source system(s) feeding into the data warehouse - the higher the chance that *exceptions* to these transformation rules will occur.

What constitutes an error or exception, and how the ETL behaves when it encounters one, are very much business decisions and a number of the most common scenarios will be described in this document. The Staging Layer is slightly different, since only pure technical errors (such as invalid data types) can occur here. It is important however to keep two things in mind when considering which exception handling strategy should be implemented as a standard within the data warehouse:

Do not implement an exception handling strategy without considering the necessary business process.

The more advanced your exception handling strategy, the higher the overhead when developing ETL.

Or, in other words: exception handling takes time to implement and, once implemented, will take effort by the data warehouse operational support team and business end users to manage effectively. These are considerations that must be taken into account early on in the design process when choosing an exception handling strategy.

1. Exception handling in the ETL Framework

Exception handling is part of the metadata model of the ETL Framework; it is shown as separate to indicate the modular nature of the model. It is possible to use the OMD (Operational Meta Data) solution without implementing an exception handling strategy.

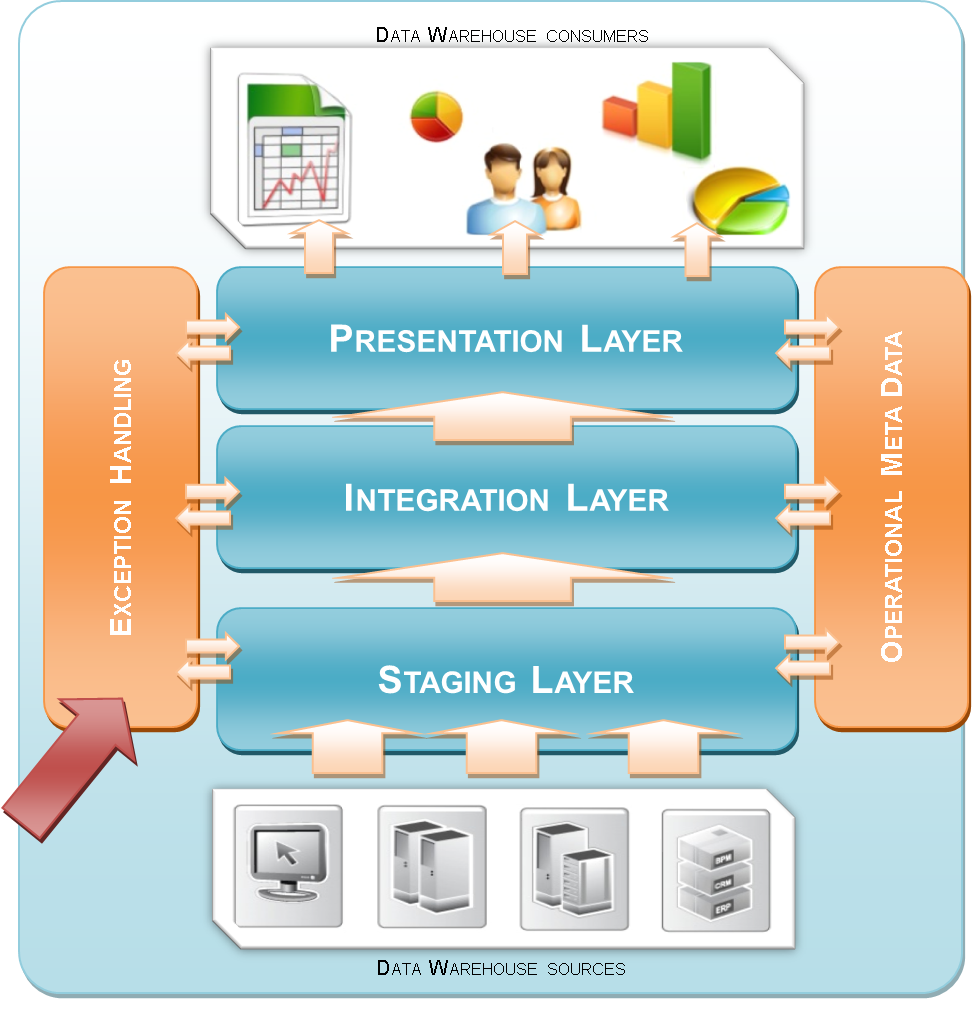


Figure 1: Positioning

1. Exception handling strategies

In essence the possible strategies with regards to the handling of exceptions in the ETL process are limited. In order of severity:

The information requirement is very dependant of the situation, weighing quality versus completeness. For instance some financial systems will require completeness in records so that the total sums show a number that matches the reality. On the other hand there may be other BI solutions that require data that is 100 % correct, which means that no (automatically detected) errors may pass to the target table.

This is why the type of error handling (as a business requirement) should be determined in the general design. Flexibility is key; an exception handling strategy will realistically use the second strategy for most exceptions, three for a few target tables with specific business requirements, and one for a small number of critical processes.

* 1. Strategy 1: report and continue

The most common approach to the ETL detecting an exception is to mark the exception as it occurs, setting the value in question to the default as defined in the ETL design documents, tagging the record in question with an exception bitmap logging a message detailing the nature of the exception to the operational meta data layer, but to write the record in question to the target table as it normally would.

Examples of situations in which this strategy is appropriate:

* Missing foreign key relationships.
* Invalid dates (e.g. Feb 31);
* A business rule defines that certain (range of) values is expected for a field, but the incoming value does not match it.

While this approach means that the data that is made available to the end users is not 100% as it ought to be, an exception occurring in one element of a record generally does not invalidate the record completely. This approach exposes the nature of the data quality issue and allows the record to be available for use while a solution can be determined. Typically 100 % data completeness is generally more important than 100 % data correctness.

* 1. Strategy 2: report and reject

In those cases where data correctness *is* considered the highest priority, the ETL can be designed to 'reject' a record in which one or more transformation exceptions are detected. Rather than writing the record to the intended target table, it will instead be tagged with an exception bitmap (explained in detail in this document), logging a message detailing the nature of the exception to the OMD layer, and written to a reject table. The reject table can then be used as the temporary repository for the record while the recycling process put in place improves its quality.

Examples of warehouse solutions in which it may be appropriate to reject records:

* Data warehouses with an operational CRM application (e.g. customers can check their own information through an online application), where it is better to display a 'temporarily unavailable' message than showing 'bad' data;
* General Ledger reporting solutions, where data inconsistencies can have wider implications (e.g. the sum of the total transaction elements in a GL item must be balanced and the dates must be correct, otherwise reject the record for inspection). In this case all transactions for a GL item should be rejected not just the single record (transaction control)

A reject table is identical in structure to the target table (e.g. D\_CUSTOMER and D\_CUSTOMER\_REJECT). Both tables will include an exception bitmap value; it may be that once a certain level of data quality is reached, the record will be allowed to pass from the reject table to the true target table, even though it is not yet 100 % correct.

It is important to keep in mind that this strategy will result in a greater impact on the development timeline; on top of the additional complexity needed in the main ETL process to enable rejection, separate processes will also be required to enable a certain level of exception recycling, and to process the data from the reject table back into the main target table once an acceptable level of quality has been achieved.

* 1. Strategy 3: report and abort

When an ETL process detects a data value that does not conform to the transformation rules it was designed to handle, the most basic approach is to abort the load process and wait for manual intervention to correct the issue. Obviously, this approach will have a major impact on the daily ETL process cycle and should be avoided in any but the most critical of exception situations.

Examples of situations in which it is appropriate to abort the ETL:

* Environment errors (network down, table space full, database connection failed);
* Risk of corrupting a data set in a manner that cannot be rolled back;
* Failure to reconcile a business-critical metric;
* Absolute data quality as well as data completeness are essential to the end solution. Given the nature of data warehousing however - DQ being one of the major challenges in this field - this requirement can be considered unrealistic in most situations

It is important that even though the ETL process in question is aborted before completing, it should still be able to write a message detailing the reason of its failure to the operational metadata layer so that its occurrence can be tracked and reported on.

1. Introducing Error Bitmaps

During the development of an ETL component any number of data checks can be implemented to validate the data quality of that record at runtime. While it is of course possible that only one exception occurs, it is very much possible that more than one issue will be detected by the ETL while processing a record.

Rather than recording multiple exception messages in the metadata layer (which is cumbersome both from a performance as well as from a maintenance perspective) the exception detection process will 'tag' the record in question with one value that sums up the total data quality level of that record.

The idea behind the exception bitmap is that one number shows which errors have occurred for that particular row. By doing this only one run is required to see all the errors which have been found for that particular mapping and record. The result of the error bit expression is stored in an extra audit attribute (ERROR\_BITMAP).

The basic principle works as follows:

| **Possible Exceptions** | **Bit Position** | **Binary Bit Position** |
| --- | --- | --- |
| 1 | 1 | 1 |
| 2 | 2 | 10 |
| 3 | 4 | 100 |
| 4 | 8 | 1000 |
| 5 | 16 | 10000 |
| 6 | 32 | 100000 |
| 7 | 64 | 1000000 |
| 8 | 128 | 10000000 |

For each possible exception a (reusable) expression checks if it occurred for the current records. If this is true the corresponding bit position (a multiplier of 2) will be selected. The eventual error code is the sum of these bit positions.

For example:

|  |
| --- |
| If the exception number **2** and **6** are detected, the resulting exception code will be **34** (2 + 32). This concept/example can also be shown in binary bitmap form as: **00100010** (read from right to left). |

Bitmaps, in theory, put no upper limit to the number of exceptions that can be attributed to a record (since every additional error simply adds an additional position to the bitmap) but experience shows that generally speaking anything more than ten individual exception checks per target table quickly becomes cumbersome to develop and to maintain.

The result can be implemented as follows:

IF (IS NULL(Error\_Bit\_1,0)\*1) +

IF (IS NULL(Error\_Bit\_2,0)\*2) +

IF (IS NULL(Error\_Bit\_3,0)\*4) +

IF (IS NULL(Error\_Bit\_4,0)\*8) +

IF (IS NULL(Error\_Bit\_5,0)\*16) +

IF (IS NULL(Error\_Bit\_6,0)\*32) +

IF (IS NULL(Error\_Bit\_7,0)\*64) +

IF (IS NULL(Error\_Bit\_8,0)\*128)

**Each target table will have its own unique exception bitmap, or combination of exceptions.**

However, it is important to understand that exceptions themselves *are* reusable across bitmaps. A customer dimension, for instance, could be linked to by any number of tables. The exception "Could not create foreign key to the customer dimension" would only exist once however, and be linked to multiple bitmaps. This way, it is possible to analyse the data quality level of individual data elements across the data warehouse.

Having an exception bitmap in the data warehouse Integration Layer makes it possible to process data to data marts based on different data quality aspects (including the severity of the exception). It is possible to provide a raw dataset, a dataset that has successfully passed a certain set of business rules or, alternatively, to publish the data with a quality indication based on the weight of the errors.

* 1. Translating Bitmaps to exception descriptions

Exception bitmaps provide full view of the data quality level through one single bitmap value, but the process is primarily aimed at high-level data quality reporting on a table ("Fact table 'x' has an invalid foreign key relationship to the customer dimension in 13% of its records, while fact table 'y' this occurs in only 0.75% of the records") and automated exception recycling ("process records into the publication layer that have no critical exceptions, and no more than two exceptions marked 'high'.").

Translating the bitmap of a single record into its corresponding list of exception descriptions can be done in databases that support scalar bit manipulation functions; this includes most commonly used platforms. The metadata reference table contains the bitmap values with the corresponding exception. By joining the exception record from the mapping with the exception reference table (bitwise join) you can show what errors have been detected for the record.

An example query for this is as follows:

**SELECT** target.\*,

bitmap.EXCEPTION\_ID, except.EXCEPTION\_FIELD, except.EXCEPTION\_DESCRIPTION

**FROM**

<tablename> target

, OMD\_EXCEPTION\_BITMAP bitmap

, OMD\_EXCEPTION except

**WHERE**

bitmap.TABLE\_NAME = '<tablename>' **AND**

bitmap.EXCEPTION\_ID = except.EXCEPTION\_ID **AND**

**BITAND**(bitmap.BITMAP\_NUMERIC, target.EXCEPTION\_BITMAP) = target.EXCEPTION\_BITMAP

Bitmaps will only show the *types* of exceptions that occurred, not the value that caused it. These values are stored in the operational metadata layer for exception recycling and could potentially be exposed as well.

1. Exception Logging in OMD

The objective of the OMD framework is to provide a structured approach to describing and recording operational information about the ETL process. It provides a logical layer of abstraction so that a consistent view of the data logistical process can be visualised, maintained and reported independent of toolset in place.

Exception handling methodology uses the OMD framework to store the messages as they are produced during the batch process. The purpose of this table is a dual one; both to enable operational reporting on exceptions as they occur, and to support the exception recycling process (see chapter **Error! Reference source not found.** for more details on recycling). While OMD supports the Error Bitmap concept it is important to note that **Error Bitmaps can also be added as attributes to Interpretation Area, Helper Area and Reporting Structure Area tables**. This is because business logic is handled by ETL in these sections of the architecture.



Figure 2: OMD with exception handling

* 1. OMD Exception Logging Tables

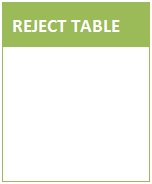
|  |  |
| --- | --- |
| **Table Name** | **OMD\_ERROR\_BITMAP** |
| **Table Description** | Contains the static error reference and their bitmap position. The table also includes a severity rating. |
| **Column Name** | **Description** |
| ERROR\_ID | Unique identifier for an exception within one bitmap. |
| ERROR\_DESCRIPTION | A description of the error. |
| SEVERITY\_CODE | Severity code (L / M / H / C) |
| ERROR\_TYPE\_CODE | Error Type code. |

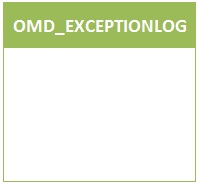
|  |  |
| --- | --- |
| **Table Name** | **OMD\_SEVERITY** |
| **Table Description** | Static reference table with the severity code and description. Standard values: Low (L) / Medium (M) / High (H) / Critical (C). |
| **Column Name** | **Description** |
| SEVERITY\_CODE | Severity code |
| SEVERITY\_DESCRIPTION | Severity description |

|  |  |
| --- | --- |
| **Table Name** | **OMD\_ERROR\_TYPE** |
| **Table Description** | Static reference table with the severity code and description. Standard values: Low (L) / Medium (M) / High (H) / Critical (C). |
| **Column Name** | **Description** |
| ERROR\_TYPE\_CODE | Severity code |
| ERROR\_TYPE\_DESCRIPTION | Severity description |

1. Exception recycling

Recycling is the procedure of detecting errors and logging these to a recycling table. After that the data can be reprocessed at any time, depending on the chosen strategy either from the reject table into the actual target table, or within the target table itself.





**eeeXXe!**



**eeeXXe**

While in most cases the number of exceptions should not be big enough to warrant additional ETL processing to automatically recycle records, the exception logging framework has been set up in such a way that it is at least technically possible. Regardless of the degree of automation, the process will be the same however depending on the specific situation the process of recycling may differ slightly, as described below.

* 1. Step 1: Trigger for Recycling

The trigger of the recycling process is the appearance of a record in the exception logging table OMD\_EXCEPTIONLOG. Either on a periodic basis or triggered by an automated message (see chapter **Error! Reference source not found.** for more information on exception reporting and notification), the data warehouse support team will analyse the detected exceptions and, based on the type, location and severity of the exception, decide on an exception resolution.

There will likely be several levels of triggering; exceptions of a high or critical severity may for instance trigger an automated warning message, while medium and low exceptions are looked at by the responsible subject matter expert once every month.

* 1. Step 2: Resolving the Exception

The recycle process will differ depending on the nature of the exception and the nature of ETL process that caused it. However, the set-up of the recycling process relies on a number of design decisions.

### Automating the Exception Resolution vs. a Manual Approach

One approach to exception resolution can be to pre-design a number of automated exception resolutions that are either scheduled to ‘sweep up’ any exceptions that occurred over a certain time period, or to be called ad-hoc as and when required. Generally speaking however, exceptions by their very nature should not occur often enough or with sufficient structure to make an additional ETL component for their resolution feasible. Exceptions generally will require data analysis to determine the root cause, and likely will need some form of data manipulation or transformation logic very specific to the situation. Automate when:

A good example of an exception that would warrant an automated solution would be an issue with foreign key relationships; a missing record in a reference table will cause any table linking to it to default to ‘Unknown’. Once the referential record has been added, an automated exception resolution ETL can update the defaulted foreign keys to the proper value. Ideally, the original ETL would be made ‘dual purpose’, enabling it to reprocess records once the exception has been resolved. In cases where this is not possible, an automated exception should still stay as close as possible to the transformation logic of (or share reusable components with) the ETL component that raised the exception however to reduce the risk of differences between the ‘true’ transformation logic in the main ETL component and the automated data fixes that are related to it.

### Using the original source record vs. EXCEPTION\_DETAIL

To resolve an exception situation it is necessary to examine and (possibly) reprocess the value that originally caused the exception to be recorded. The OMD\_EXCEPTIONLOG table offers two possible approaches.

* **The target record originated from only one source record**

The easiest method of recycling an exception is to simply reprocess the original source record once the issue has been resolved.

The exception log table OMD\_EXCEPTIONLOG is used in this case to keep track of the source table and the unique primary key value of the source record in question (columns SOURCE\_TABLE and SOURCE\_PK respectively).

This approach could potentially even mean that the exception can be recycled, semi-automatically, by the original ETL process that caused the exception - with only minor modifications. A prerequisite, naturally, is that the source table in question is still available (this may not be the case if the source is a staging table for instance) at the time of exception resolution, and that the source table includes a primary key column.

* **The target record did NOT originate from only one source record**

In situations that the ETL is more complex, it may no longer be feasible to simply reprocess the original source record. For those cases the OMD\_EVENT\_LOG table contains a free-form text column that is used to store the values that caused the exception(s): EXCEPTION\_DETAIL.

In this case, the OMD\_EVENT\_LOG table is directly used as the source for the resolution of the exception. While the field is free-form, the format is fixed (source values delimited by a pipe character (”|”), the order of appearance in line with the exception bitmap. This ensures that it is still possible to automate exception resolution, if so required.

* 1. Step 3: Reprocessing into the Target

To reprocess a resolved exception into the target table, two approaches can be taken depending on business requirements.

* **Historically track data quality changes**

It is possible to track exception resolution as just another type of record change. The original record that includes the exception is closed off (end-dated) and a new record is added with the exception situation resolved. Choose this approach when it is important to expose which, when and how exceptions were resolved.

* **Overwrite exceptions with resolutions**

Tracking exception resolution historically does mean that data quality related record changes are mixed in with ‘true’ data warehouse source driven change. Because separating the two can be technically challenging, or in the case of large data volumes undesirable from a performance perspective, there are situations where it is advisable to simply overwrite the value that caused the exception once a resolution has been determined. Choose this approach when there is no business need to track data quality related change in the data warehouse.

If reject tables are part of the exception handling process, records marked with exceptions will generally remain in the reject tables until an acceptable level of data quality has been reached. The above would occur in the reject table rather than in the actual target table. Once the record is deemed to be of sufficient quality, it will be inserted into the true target table, and deleted from the reject table.

* 1. Tracking resolution in OMD\_EVENT\_LOG

Because each record in OMD\_EVENT\_LOG is stored as an exception *bitmap* that can contain any number of exceptions, an updated record (minus the resolved exception) is then re-added to OMD\_EVENT\_LOG if there are still outstanding exceptions. This is conforming to the design of OMD, which creates a new module instance for every run.

1. Exception handling per Data Warehouse layer

The exception handling approach will differ depending on the layer of the solution architecture.

* 1. Staging Layer

The Staging Area is the most vulnerable to environment errors, which will (have to) be logged by the tooling at hand. The truncate/insert mechanism which is able to select the same set in case of errors is sufficient for this purpose.

The only conversions executed here, potentially, are data type conversions. In this case, the process will have to fail because there is no efficient way of handling these exceptions. For this reason transformations of any kind are advised against while processing data into the staging layer of the data warehouse; while records are still in their source system or stored in flat files records are not controlled or hard to retrieve, which will make exception resolution challenging if not impossible. Once a record is in the staging table it is visible, assigned a unique primary key, and can be tracked.

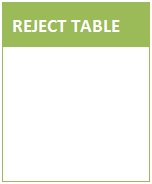
If an exception is detected in the staging layer, an exception will be logged in OMD\_EVENT\_LOG, but the process will be aborted.

Because the possible two-step process in the staging layer (first load the flat file, then convert the data types) two exception tables may exist for one particular source file: a reject table for the data type conversion and a reject table for the staging to integration process.

* 1. Integration Layer

Because the purpose of the Integration Layer is to integrate the various data warehouse sources, the most likely type of exception will be a failure to link two data elements: a failure to determine an expected surrogate key.

Combining data from various previously unlinked data sets is inherently prone to exceptions; the level of data quality between the source systems involved may differ, timing issues can occur, etcetera. In those cases where the issues are such that it becomes important to ‘protect’ the Integration Layer from the resulting data pollution it may be necessary to stop the data at the gates. An alternative rejection strategy can be implemented in this case. This is displayed in the next diagram.



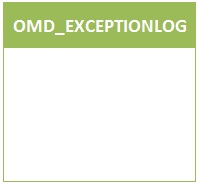
**eeeXXe!**

**eeeXXe**

**eeeXXe**

**eeeeee**

Integration Layer



Staging Layer

Rather than creating a reject table based on the target (integration layer) table, the reject table has the format of the source (Staging Layer) table. If the record is not allowed into the integration layer it is ‘parked’ in the reject table and reprocessed by the ETL process in question during its standard process run. While the exception remains unresolved the record remains in the reject table for reprocessing during the following run, until such time that the source alignment issue has been resolved and the record can pass into the integration layer. All other types of exception will be processed as described in the rest of this document.

* 1. Presentation Layer

Exception handling in the Presentation Layer is directly related to the functional requirements and, as such, extremely diverse. While discussing the business requirements, the question what to do in case the data does not adhere to the transformation rules defined (the “else” or the rule) should always be asked, and clearly documented. Rather than simply developing the ETL to cater for every possible exception, it should be the business users who define which exceptions are important enough to be logged, what their severity should be, and which response is required.