NO MORE FIREDRILLS GUIDE TO

BUSINESS INTELLIGENCE TESTING

BY THOMAS BOLT, CTO BIG EVAL DATA QUALITY SOLUTIONS

BiG EVAL

High Data Quality. Mastered.

BIGEVAL.COM

IMPRINT

Author

Thomas Bolt, CTO BiG EVAL

Publisher

BiG EVAL - Bolt Technology Consulting GmbH
Oberfeldstrasse 12a
8302 Kloten
Switzerland

All Rights Reserved

Reprinting, publishing or copying this document or parts of this document is only allowed with the written approval of the publisher and by referencing the source. All Copyrights are held by the publisher and the authors or the stated holder of rights.

INHALT

Why Testing is so Important ... 4
- Testing by Developers ... 4
- Developer Reviews ... 4
- Testing a feature one time only ... 5
- Acceptance tests by business experts and end users ... 5
- In-house-built testing framework ... 5
- Conclusion ... 5

Testing a Business Intelligence System ... 6

Source Data ... 7
- Conformity of Source System Interfaces ... 8
- Schema of Source Data ... 8
- Availability of Source Data ... 8
- Accessibility of Source Data ... 8

Staging Area ... 9
- Availability of Source Data ... 10
- Records count ... 10
- Business key completeness ... 10
- Format checks ... 10
- Historization complete ... 10
- Performance Checks ... 10

Data Integration ... 11
- Initial Load ... 12
- Regular Load ... 12
- Rollback ... 12
- Reload ... 12
- Rerunning data loads without rollback ... 12
- Business Logic ... 12
- Historization ... 13
- Data Cleansing ... 13
- Performance ... 13

Data Model and Meta Data ... 14
- Data Model fully implemented ... 14
- Correctly sized data types ... 14
- Check for data truncation ... 14
- Referential integrity checks ... 14

WHY TESTING IS SO IMPORTANT

Often I get asked why a data warehouse should be tested. This question sounds really weird when you think at the huge impact data has on your business today. It should be clear that a robust testing strategy is needed, and I'm sure the questioner is aware about that. So, every time when I hear this question I try to dig deeper to find out what the questioner really means. And here are some of the statements I usually hear:

"Developers already tested the system components they built for the data warehouse."
"We do developer reviews."
"We test a feature before it gets released."
"Our business experts and end users perform acceptance tests."
"We built our own testing framework."

Is that really enough testing?

Let us spot some light on each of these statements.

Testing by Developers

That's great and it should be done by every developer. But usually there is no clear strategy behind these white-box tests. Developers test specific situations they may have experienced in the past, or situations that they believe to be critical. When tests succeed, they proceed with their work. When tests fail, they fix the issue immediately.
These tests usually cannot be repeated to ensure that the problem doesn't arise again. And the test coverage is usually low or very limited.
And there's one more thing: It's in the nature of each individual to think, that things built by themselves work flawless. As a result, testing is usually done insufficiently.

Developer Reviews

The four-eyes-principle is clearly one of the most effective ways to prevent development errors. But it's just one little cogwheel in the whole gear of your testing strategy. And only things that are obvious and understandable by both people can be reviewed.

Testing a feature one time only

There are many reasons why a system component breaks or stops working correctly when another feature gets added or existing ones get changed. Often there is no direct connection between these features and no one thought this could happen. So testing a component should be repeatable with every release to ensure that nothing breaks what already worked before.

Acceptance tests by business experts and end users

Acceptance tests are really important in every project. But they are usually done once before releasing a new feature. As I already told you before, there are many reasons why a system component could break suddenly. So in fact, acceptance tests should be repeated with every release to test a system reliably. But that's usually unrealistic because of the huge effort needed by multiple involved in acceptance tests.

In-house-built testing framework

Great! You're on the right way. But what was the effort to build this framework? And what efforts are needed to maintain it? Are there other developers who are able to take over the code when the creator leaves your company? And is the feature-set flexible and rich enough to fulfill future needs? These are just some questions that you should think about before relying on a proprietary development. And keep in mind, that there are proven standard tools on the market that were built with an effort of many years.

Conclusion

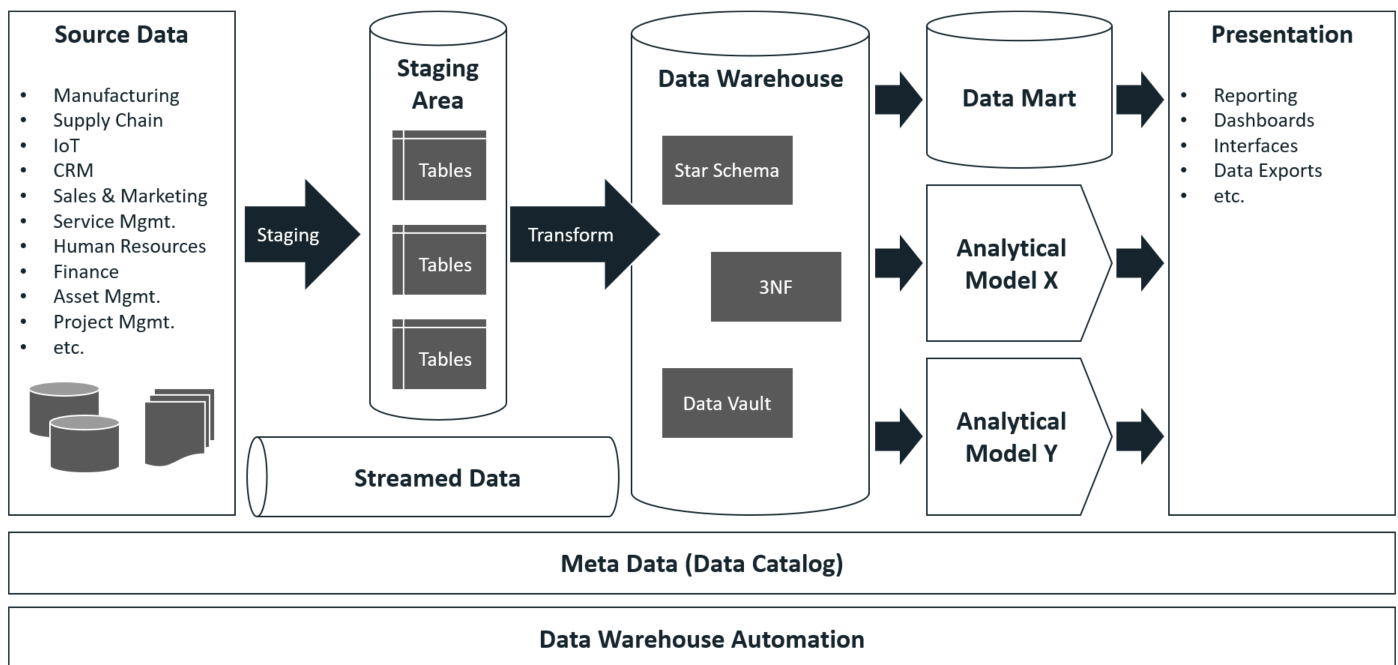
Most of these approaches are adequate and important. They should be part of your testing strategy for sure. But when we look at a data-driven project, there are many reasons why we cannot only rely on these. Most important is the automation aspect. As you may have recognized, all type of testing should be repeatable to assure that the tested component keeps working correctly during the whole lifetime. It shouldn't break suddenly with a following release.

With the repeatability in mind, there comes the need for automation. Running tests manually over and over again, needs a lot of personnel resources what costs a lot of money. And it needs time what you clearly don't have in an agile project.

So, let us look at what should be tested, how it should be tested, and how you can automatize things. Go on to the next chapter.

TESTING A BUSINESS INTELLIGENCE SYSTEM

A data warehouse is not just a database that stores integrated data. There are many different system components and processes that build together the data warehouse system. And if we go further and think about how data gets analyzed or used in processes and decisions, there are subsequent components as well, that may build a whole business intelligence system. So, when thinking about our testing strategy we should broaden our focus on all these components.



SOURCE DATA

Usually an enterprise data warehouse has many different sources of data. Each of them with its own technology and data model. The quality of the information that these systems provide, needs to be of adequate quality before you should load it into your data warehouse. But the quality also varies tremendously because of the very different ways, data gets entered and managed either by people or by technical processes.
You don't need 100% accuracy, because this would be nearly impossible to reach without an enormous effort. You should find the level of quality that allows you to fulfill the goals your project has with adequate costs. There are data quality management solutions like BiG EVAL that allow you to ensure a high level of data quality in an automated way to lower costs and efforts.

Conformity of Source System Interfaces

There are interfaces that allow to access source data. Often these are industry standards that don't need to be tested explicitly. But when it comes to customized interfaces like an export of flat files or web service API's for example, there should be a clear definition of the interface.

Testing is needed to ensure that there are no differences between the interface definition and the actual implementation. Otherwise, a data integration process could fail.

Running these test cases automatically in a productive data warehouse environment makes sure that the interface conforms the definition even when changes and bug fixes get deployed.

Schema of Source Data

When data integration processes fail, often changes in the schema of source data are the reason. There are many different changes that may harm your data integration processes like:

» Names of files changed

» Names of tables or columns changed

» Data Types changed

» Columns get removed or others get added

» etc.

These changes usually happen because the vendor or the development team apply system updates. Often the teams that are responsible for subsequent systems like a data warehouse, do not get informed about that.

Adding a column or additional tables is usually not a problem. But when columns or tables are missing, or even worse when data types get changed, this leads to huge problems.

Automatically monitoring the schema of the source data for changes, helps you to detect these problems early. That's important during development to ensure that nothing breaks shortly before your release deadline. But it's much more important when your data warehouse is productive.

Availability of Source Data

You should frequently check whether your source data is available in two different manners.

Data should be available technically. That means that the technical infrastructure like networks, firewalls etc. need to work correctly. You can ensure this using a monitoring infrastructure to check whether your infrastructure works, and whether your data integration servers are able to access the data.

But data should be available on time as well. So, if you have access to data from technical point of view, it doesn't mean you have access to current data. Apply test cases that check your data for timeliness.

Accessibility of Source Data

Enterprise data is usually secured by a strong data security concept. To access data, a security context with adequate permissions is needed by the data integration process. Usually these are user credentials stored in the data management system or in directory services like Active Directory.

Accidentally it may happen, that permissions get lost, user accounts get disabled or passwords expire. A test case that runs in the same security context as the data integration process does, is able to check whether access to data is allowed at any time it is needed.

STAGING AREA

The staging area is the first point where your data collection starts. There are various concepts for building a staging area. The easiest kind of staging area is just a central place where your source data gets stored during the first step of the data integration process (ETL process). This step is also called “extraction” and represents the “E” within ETL. When the whole data integration process finishes, the staging area may be emptied completely. There is usually no special functionality, why testing is quite simple.
But when there are special requirements like querying older versions of data records from the source systems to rebuild history, or when there is no possibility to get delta-datasets from the source systems that only contain the records that were changed since the last data load, a persistent staging area can be a viable solution. This kind of staging area architecture needs some more testing to ensure it is working correctly whenever it is needed.

Availability of Source Data

Ensure that all source data is available in the staging area when the extraction step finished. This test is about validating that on a high level. At this point, it is enough to check whether data could be fetched from all the source systems.

Records count

The next test should be about checking whether there is the right amount of records for every entity that was extracted. Is the amount of records plausible? Does it meet the expectations? Does it exactly meet the amount of records in the source system. The latter can be done by comparing the records count in the source system with the one in the staging area.

Business key completeness

Counting the amount of records is not always enough. When some records are missing and some were duplicated anyhow, you get a false-positive test result. So checking whether each and every business key or record ID is present in the staging area, is much more safe. But it takes more time and may harm the systems performance.

Format checks

There are many reasons, why extracted data could be falsified during extraction. The main reason is, that the goal of a staging area is to collect data form many different technologies with different configurations.

First at all, there could be a loss of special characters because of different character sets or collations used in your source systems and your staging area.

Date and Time values are hot subjects for errors as well. There are many different ways to store date and time information. Also timezone-awareness is important and may need complex handling.

One more thing is to check, whether data gets truncated. A feasible test could be to compare meta-data (data type and length) between source system and staging area.
Another good indicator for such errors is to check whether strings fill the columns in the staging area completely. If there are many strings that fill the columns completely, there is a high risk that data was truncated.

Test all these things by one of the following ways or by a combination:

» Compare data between source systems and your staging area to find differences.

» Compare staged data against a reference data set.

» Test extreme values during the system development. Check whether your system behaves correctly.

Historization complete

The complexity of a staging area rises when you decide to build a history for your data records. A persistent staging area is needed that tracks the state of records from your source systems.
One of the most common problems with persistent staging areas is, to keep the history complete. There shouldn't be any gaps or overlaps between versions. And the most current records should be available.

Performance Checks

Staging data can be time consuming depending on the volume of data and the communication technology between the source system and the staging area. Staging can be a bottleneck in the whole data integration process. That's why it is wisely to check the performance regularly. Not only during development, but much more when the system is in productive use.

DATA INTEGRATION

This is where all data comes together and where the most complex development work has to be done.
The data integration process usually consists of multiple steps that cleanse data, make data from different sources comparable, combines it and finally loads it into the data warehouse model - usually in a historized form.
Depending on the data and the complexity of the data, there could be many things that could go wrong. Therefore, a good testing strategy is needed for this part of the data warehouse.
There are also different data loading scenarios that maybe all or at least some of them need to be realized. There's the initial load that get run one time only to load all available data from the past until now into the data warehouse. The regular data load runs continuously on a daily basis or whatever loading interval you use. Then there may be loading scenarios for fixing errors that could arise also in a very well tested data warehouse. After rolling back a data load, a robust process should be able to rerun it again to incorporate data corrections made in the source systems or anywhere between.

Initial Load

An initial load scenarios gets implemented to run only once. Because of that, it happens quite often that implementation quality lacks. Make sure that the initial load runs smooth, correct and within the time available to prevent fire drills at the time when your data warehouse release should go live.

Regular Load

In contrast to the initial load, the continuous load will run regularly and is the main pillar of the data warehouse system. Take care that this process is tested very well. These tests need to be repeatable by the push of a button to reuse them every time you intend to build a new release or a hot fix. This is the most effective way to keep a high quality during the whole life cycle of the data warehouse system.

Rollback

Every good data management technology is transaction aware to give you the possibility to roll back changes easily. But these features are usually made for short and atomic transactions. Whereas in a data warehouse load, we usually perform a huge amount of data operations that take much more time to run.
Depending on whether you can rely on the transaction features of your underlying data management technology or not, the rollback mechanism can get very simple or very complex. Depending on the complexity, you should take attention within the testing strategy.

Reload

Running a data load again usually means, that the last data load gets rolled back and rerun. But there may be much more complex scenarios, where a specific range of dates should be reloaded or similar. This is quite difficult to do, because history may be lost and should be rebuilt in a similar way, the initial load does it. This could be a huge effort and must be tested very well. Otherwise you could loose a part of your data history.

Rerunning data loads without rollback

Are you aware about what happens, when the same data load gets run multiple times without rolling it back in between? This is a scenario that gets overlooked many times. Take care of that.

Business Logic

When talking about business logic in the context of a data warehouse system, we like the naming of hard and soft rules as Dan Linsted used to do it in his Data Vault concepts. Hard rules do not change the meaning of information and are usually more technical (e.g. concatenation, splitting, trimming, changing data types etc.). Whereas soft rules are the respective business rules, that change and interpret data.

The implementation of hard rules can be tested quite easy, because they are usually not very complex, and the behavior and output is well defined.

Soft rules can vary from a simple one to very hard understandable rules. Try to break down the rules into atomic tasks that are easier to understand and also easier to test.

Often I get asked whether there is a testing software, that automatically understands business rules implementation based on the code. Sadly, theres no such thing on the market. Furthermore, such a software wouldn't make sense. It would test the artifact wrong if there were errors in the code because it would assume that the error is the correct implementation.

So there's no better way as building tests manually. In fact, it makes sense to build tests by another person than the developer itself. Because the other person will build the tests based on the requirements definition instead of using the implementation know how that may be wrong.

There are the following options for building business logic test cases:

» Black box testing. Run the smallest possible task of your business logic by using reference

data as the input. Then check the result whether it meets the expected result. Depending on the data load technology you may run a stored procedure or check the output of a view against the expected outcome.

» **White box testing** can be used to check specific things that were implemented in the business logic. It needs the implementation knowhow of developers. But it works quite similar like black box testing. Input some reference data and check the outcome. See what happens when you use extreme values. Does the task behaves as intended? Does it succeed? Does it fail? Does it write error logs?

Historization

Most data warehouse systems build up a kind of data history. The store different versions of data records to allow end users to analyze development of values or data changes over time.

Every data modeling concept uses architectural patterns for data history. Therefore you can use a defined set of test cases for every historized entity.

An automation solution like BiG EVAL makes you capable of building these test cases only once and automatically applying them onto all historized entities in your data warehouse system.

These are the most important things your test cases should consider:

» Is the history complete from start to end?

» Are there any gaps between versions of data records?

» Are there any overlaps of versions? E.g. V1 ends in March, but V2 starts in February.

» Get older records correctly terminated when adding new versions? E.g. Setting end-date or current-flags etc.

Data Cleansing

We do not like data cleansing mechanisms in data integration processes because we recommend to correct data in the source systems. But we are aware about the difficulties of doing that.

Data cleansing is a wide spread term from an architectural point of view. It could be anything regarding mangling data to bring it into the needed form. Sometimes there is complex business logic and complex data mappings or similar that gets applied to correct errors made in the source systems.

Ensure that there are automatable test cases for all cleansing fixes. Especially when these cleansing fixes rely on manually maintained mapping tables or similar, you should implement test cases for runtime. The reason is simple: Where manual work is needed, there could be human input errors.

Performance

Usually there is only a short time window within data loads should be completed. The loading time evolves over time. In the first iterations, the data loads need a lot of time because they are not completely optimized. After a refactoring iteration, the loading time drops and is acceptable. But when data volumes rise - and that usually the case when building history - the loading time rises as well. Sometimes exponentially. That's why you should build performance tests for the productive runtime of the system. These performance tests can be based on loggings. They can check whether the loading time exceeds a specified amount of time.

DATA MODEL AND META DATA

Data Model fully implemented

Check whether the actual data model is fully implemented compared to the one defined in the systems specifications.

Correctly sized data types

An architectural rule says, that one should choose the data types with the smallest need of storage space. That's true regarding system performance. But there could be situations, where you are better to choose larger data types. There are the following reasons for that:

» Information stored could grow over time.

» The aggregate of a huge amount of small numbers could be really big and may not fit into the same data type.

» Vendors of source systems may change data types. Therefore, you need enough flexibility in the data warehouse.

» Source systems could be replaced (migrated).

We recommend to validate data types during development. This could be a manual task or even an automated test. Using an automated test, you may aggregate data and check whether it fits into the data types used. Or you may compare data types of the source systems with the data types of the target columns in your data warehouse.

Check for data truncation

When data truncation occurs, you are going to loose information. So there is a need to check for data truncation on a regular basis. The logic behind this kind of test is easy. Check for every character- or string-field in the data warehouse, whether there is a huge amount of records that fill the field completely. E.g. you have a Status-Field with a width of 10 characters. If there are many records that use all these 10 characters, there is a high risk that data was truncated.

Referential integrity checks

There are forms of data warehouse architectures, where the referential integrity checks of the data base management system (DBMS) are disabled. Data Vault 2.0 for example requires to disable referential integrity checks when you want to load data in parallel or in other words - asynchronously.

Do not run a critical data warehouse system without any referential integrity checks. When the DBMS's functionality must be disabled for any reason, you should check referential integrity by an automated test case that runs continuously.
We described such an implementation using BiG EVAL in the following blog post:
https://bigeval.com/dta/data-vault-consistency-without-referential-integrity/