**WHITE PAPER** | APRIL 2016

# Fully Automated ETL Testing: A Step-by-Step Guide

ca technologies

**Section 1**

## The Critical Role of ETL for the Modern Organization

Since its eruption into the world of data warehousing and business intelligence, Extract, Transform, Load (ETL) has become a ubiquitous process in the software world. As its name suggests, an ETL routine consists of three distinct steps, which often take place in parallel: data is extracted from one or more data sources; it is converted into the required state; it is loaded into the desired target, usually a data warehouse, mart, or database. A developed ETL routine will also often include error-handling, logging infrastructure and the routine environment.[1]

Until now, ETL has been used primarily to prepare often large and disparate data for analytics and business intelligence. However, its use is broadening beyond simply moving data, with migrating data for new systems an increasingly common application, as well as handling data integrations, sorts, and joins.[2]

ETL is therefore a critical feature of the modern, fast-moving development lifecycle, where multiple releases and versions are developed in parallel at any given time. Organizations must be able to constantly enhance, integrate and innovate their software, with data available to testers and developers in the right state for each iteration and release. The data will be drawn from the same sources, but must be transformed to match each individual team's specific requirements. This is especially true if an organization is striving to be "Agile", or to successfully implement Continuous Delivery.

A good example of the critical role of ETL in Continuous Delivery was found at a large multi-national bank, which CA Technologies worked with. The bank was in the process of making an acquisition, and had to migrate the acquired banks customers, products and financials into their existing infrastructure. This meant that 80 insertion files had to be retrieved, converted, and validated, before being uploaded to the banks' back-end systems. Such data further had to be available across 47 projects in parallel, with the referential integrity of data maintained.[3] In this instance, ETL was critical to enabling the parallelism required for successful Continuous Delivery.

However, in spite of the increased use and importance of ETL, ETL testing reflects the state of testing in general, in that it is too slow and too manual, and yet allows an unacceptably high amount of defects through to production. This paper will set out the challenges encountered in a typical approach to ETL testing is typically approached, exploring the challenges encountered. An alternative, broadly model-based approach will then be set out, considering how it might make ETL testing far more efficient, effective, and systematic.

**Section 2**

## The Typical Approach to ETL Testing and the Common Challenges Encountered

When validating ETL transformation rules, testers typically create a shadow code set, use it to transform data, and then compare the actual results to the expected results. Usually, ETL scripts or SQL is manually copied to the source data and run, with the results recorded. The same script is then copied to the target data, with the results recorded. The two sets of results (actual and expected) are then compared, to validate that the data has been correctly transformed.

## The Root Issue: Complexity & Testability

The underlying issue behind such manual validation is that ETL routines, by their very nature, rapidly become highly complex. As the business grows, and the variety and volume of data it collects increases, the ETL rules grow in order to handle it. In the so-called 'information age', this growth is happening faster than traditional testing methods can handle. In fact, the sheer amount of information being collected by data driven organizations has grown so fast that 90% of the data in the world was collected in the last two years alone4, while the amount of data collected by the average organization is doubling each year.[5]

The complexity of the systems designed to collect, transfer, operate on, and present this data grows exponentially with every added decision. This includes ETL rules, and there are numerous factors which can impact the complexity of the transformations:

- The number and variety of data sources involved, including relational and non-relational database types, and flat files;
- The number and variety of data targets;
- The number and variety of simple and complex transformations, from simple look ups, to active unions and normalization;
- The number of re-usable transformations, code snippets, and joins;
- The number of tables created.[6]

All of these factors are exacerbated by the current focus on near real-time solutions, and the added complication they bring.[7]

## The Documentation Doesn't Help

This growing complexity directly impacts the testability of ETL routines. It is especially problematic for ETL testing, as the transformation rules are typically stored in poor documentation, lacking explicit expected results. The rules are often designed during the development phase itself, and are frequently stored in written documents or spreadsheets – or worse, they might not exist outside the developers and testers' minds.[8] In this instance, no real documentation exists from which test cases (i.e., the shadow code) can be confidently derived.

At one business intelligence team we worked with, requirements were stored as written documents, while test cases were stored in spreadsheets. This static documentation presented a "wall of words", from which the logical steps of the ETL routines had to be deciphered. The documents were "happy path" focused and contained no negative conditions, so that around 80% of the possible logic which needs to be tested in the average system was omitted. Such incomplete, ambiguous documentation meant that testers had no way to easily or accurately understand the ETL routines.

Too often, testers were left filling in the blanks, but when they got it wrong, defects enter the ETL routines. Invalid data was then copied to the target, even though the code and test cases reflected a plausible interpretation of the requirements documentation.

## "Garbage in, Garbage Out" – Manually Deriving the Test Cases and Expected Results

In fact, a massive 56% of defects which make it through to production can be traced back to ambiguity in requirements documentation.[9] This is partly because test cases and expected results are manual derived from the poor documentation: an immensely time consuming process which usually leads to poor test coverage.

### Quality

Manual derivation is ad hoc and unsystematic, and usually leads to the creation of whichever test cases occur in the minds of testers. Given the discussed complexity of ETL routines, combined with the poor documentation on offer, it is unfair to expect even the most talented tester to create every test needed to validate the possible data combinations. For example, if a simple system with 32 nodes and 62 edges is designed in a linear fashion, there can be 1,073,741,824 possible routes through its logic – more than any one person can think up.

Ad hoc derivation therefore leads to massive under-testing and over-testing, where only a fraction of the possible logic involved in an ETL routine is tested. Negative testing will be a particular challenge, and the test cases, like the documentation, usually focus almost exclusively on happy paths. However, it is these outliers and unexpected results which must be tested against, as it is imperative that the ETL routines reject such "bad data".

One financial services company CA Technologies worked with, for example, relied on 11 test cases with just 16% test coverage. This figure is fairly standard, and our audits have found 10-20% functional test coverage to be the norm. Another project at the company was over-testing by a factor of 18 times, as they piled up test cases in an effort to fully test the system, but still did not achieve maximum coverage. The 150 extra test cases cost them $26,000 to execute using an outsourced provider.[10]

The result of such poor coverage is defects enter the code, where they are expensive and time-consuming to fix: studies have found that it can cost 40-1000 times more resources[11], and 50 times more time12 to fix a bug during testing, compared to finding it earlier. Worse, bugs might go undetected, so that invalid data is copied to the live target, where it can threaten the integrity of the system. What's more, when faced with static documentation, testers have no reliable way to measure the coverage of their test cases - they simply cannot say with confidence how much, or how little of a given routine they are testing, and nor can they prioritize tests based on criticality.

### Time and effort: Testing Simply Cannot Keep Up

Writing test cases from such documentation is also immensely time and labour intensive. In the previous example, it took 6 hours to create the 11 test cases, while the rampant over-testing at the company drained even more time. This time wasted on manual test case design is made worse by the time which then has to be spent comparing the actual and expected results.

Comparing the vast individual fields to the expected results is highly time-consuming, given the amount of data produced by a complex ETL routine, and the fact that the source data will often be stored in a diverse variety of database and file types. It is also highly difficult, as the transformed data has to be validated on multiple levels:

- Testers have to check for data completeness, making sure the counts of the data source and target match;
- Data integrity must be ensured, checking that the target data is consistent with the source data;
- The transformation must match the business rules;
- Data consistency must be guaranteed, identifying any unexpected duplicates;
- Referential integrity has to be maintained, with any orphan records or missing foreign keys spotted.[13]

Sometimes a compromise is made, and only a sample dataset is validated. However, this also compromises the thoroughness of the ETL testing, and so the reliability of the transformations is impacted. Given the role of many ETL routines in business critical operations, such a compromise is unacceptable. Quality is further affected by the error-prone nature of manual comparisons, especially when the expected results are poorly defined, or, worse, are not defined independently of the shadow code used in testing at all. In this instance, testers tend to presume that a test has passed, unless the actual result is especially outlandish: without pre-defined expected results, they are likely to assume that the actual result is the expected result[14], and so cannot confidently determine data validity.

## The Data Problem

So far, the issues encountered deriving the tests (shadow code) needed to validate ETL rules has been focused on. However, once the test cases have derived, testers need dummy source data to pump through the system. This is another frequent cause of bottlenecks and defects.

### Do You Have All the Data Needed to Test the Complex ETL Routines?

Having enough "bad" data is critical for effective ETL testing, as it is paramount that, when in operation, an ETL rule rejects this data and sends it to the appropriate user, in the appropriate format. If it is not rejected, then this bad data is likely to create defects, or even system collapse.

"Bad data" in this context can be defined in a number of ways, which correspond to the ways in which data must be validated by testers. It might be data which, based on the business rules should never be accepted – for example, negative values in an online shopping cart, when no voucher is present. It might be data which threatens the referential integrity of a warehouse, such as missing inter-dependent or mandatory data, or data missing among the incoming data itself.[15] The test data which is pumped through an ETL validation rule must therefore contain the full range of invalid data, in order to allow for 100% functional test coverage.

Such data is rarely found among the production data sources still provisioned to test teams at many organizations. This is because production data is drawn from "business as usual" scenarios which have occurred in the past, and so is sanitized by its very nature to exclude bad data. It does not contain the unexpected results, outliers, or boundary conditions needed for ETL testing, and instead will be "happy path" focused. In fact, our audits of production data have found just 10-20% coverage to be the norm. The irony is, the better a routine has been built, the less "bad data" will have been allowed through, meaning that there is less data of sufficient variety to fully test ETL rules going forward.

### Is the Data Available When You Need It?

Another major issue for ETL validation is the availability of data. Source data might be drawn from 50 different sources across an enterprise. The problem for ETL testing, and testing in general, is that it is viewed as a series of linear stages, so that test teams are forced to wait for data while it is being used by another team.

Take the example of a banking migration chain, where data is being taken from one bank and converted into the other's systems, using a reconcile tool. At each stage, the data needs to be validated, to check that it has been correctly converted into the financial control framework, that the account number was retrieved, that there was hour to hour correctness, and so on. This process might fall into several distinct phases, from the basic input, through de-duplication and preparation, to propagation and the data being booked in. There might further be multiple teams involved, including both ETL teams and non-ETL teams, particularly those working with the mainframe.

If the data from every data across the enterprise is not available to every team in parallel, then delays will mount as teams sit idly waiting. Testers will write their ghost code, and then not have the source data to validate an ETL rule, as it is being used by another team. In fact, we have found that the average tester can spend 50% of their time waiting for, looking for, manipulating, or creating data. This can constitute a massive 20% of the total SDLC.

### What Happens When the Rules Change?

Manually deriving test cases and data from static requirements is highly unreactive to change. ETL routines change as quickly as a business evolves, and the volume and variety of data it collects grows with it. When such constant change occurs, however, ETL testing cannot keep up.

Arguably the single greatest cause of project delays in this instance is having to check and update existing test cases when the routines change. Testers have no way to automatically identify the impact of a change made to the static requirements and test cases. Instead, they have to check every existing test case by hand, without any way of measuring that coverage has actually been maintained.

With the business intelligence team mentioned above, where requirements and test cases were stored in written documents and spreadsheets respectively, change was especially problematic. It took one tester 7.5 hours to check and update a set of test cases when a change was made to a single ETL rule. At another organization we worked with, it took two testers two days to check every existing test case when a change was made to the requirements.

**Section 3**

## The Viable Alternative: Fully Automated ETL Testing

It is clear, then, that as long as test cases are manually derived, and the results manually compared, ETL testing will not be able to keep up with the rate with which business requirements constantly change. Below is a possible strategy for improving the efficiency and efficacy of ETL testing. It is a model-based, requirements based strategy, designed to shift left the effort of testing, and build quality into the ETL lifecycle from the very start. Such a Model-Based approach introduces automation across every stage of testing and development, while making ETL testing fully reactive to constant change.

## 1) Start with a Formal Model

Introducing formal modelling into ETL testing offers the fundamental advantage that it shift lefts the testing effort, where all subsequent test/dev assets can be derived from the initial effort of mapping an ETL rule to a model. The formal model therefore becomes the cornerstone of fully automated ETL validation.

However, formal modelling also helps to resolve the more specific issue set out above, of ambiguity and incompleteness in requirements. It helps to maintain testability in spite of the ever-growing complexity of ETL rules, so that testers can quickly and visually understand exactly the logic which needs to be tested. They can easily know both the valid and invalid data which should be inputted to fully test a transformation rule, and what the expected result should be in each instance.

A flowchart model, for example, breaks down the otherwise unwieldy "wall of word" documentation into digestible chunks. It reduces the ETL into its cause and effect logic, mapping it to a series of "what if, then this" statements, linked into a hierarchy of processes.[16] Each of these steps in effect becomes a test component, communicating to the tester exactly what needs to be validated. Modelling ETL routines as a flowchart therefore eliminates ambiguity from the requirements documentation, working to avoid the 56% of defects which stem from it.

As the ETL routines grow more complex, the flowchart serves as a single point of reference. In contrast to "static" written documents and diagrams, additional logic can be easily added to the model. What's more, the abstraction of highly complicated routines is possible using subflow technology, thereby increasing testability. Lower-level components can be embedded within master flows, such that the numerous routines which make up a highly complex set of ETL rules can be consolidated into a single, visual diagram.

In addition to reducing ambiguity, flowchart modelling also helps to combat incompleteness. It forces the modeller to think in terms of constraints, negative conditions, restraints, and boundary conditions, asking "what happens if this cause or trigger is not present?" They must therefore systematically think up negative paths, accommodating the negative testing which should constitute the majority of ETL validation. Completeness checking algorithms can further be applied, as the formal model is a mathematically precise diagram of the ETL rule.

This eliminates missing logic like "dangling elses", so that test cases which cover 100% of the possible combinations of data can be derived (it should be noted, however, that there will almost always be more combinations than can be viably executed as tests, and so optimization techniques will be discussed later). Another major advantage is that expected results can be defined in the model, independent of the test cases. In other words, with a flowchart, the user can define the model to include the boundary inputs and propagate their expected result to different end points in the model. This clearly defines what should be accepted and rejected by a validation rule, so that testers do not wrongly assume that tests have passed when the expected result is not explicit.

It should be noted that adopting Model-Based Testing for ETL validation does not require the full adoption of a requirements-driven approach to testing and development enterprise wide. It does not demand a complete sea change, and, in our experience, it takes as little as 90 minutes to model an ETL routine as an "active" flowchart. This model can then be used by the ETL or test team for the sole purpose of testing, and re-testing that routine itself.

## 2) Automatically Derive Test Cases from the Flowchart Model

The introduction of Model-Based Testing can automate one of the major, manual elements of ETL testing: test case design. Tester's no longer need to write ghost code, or manually copy SQL from the source to target database. Instead, the paths through the flowchart become the test cases, which can be used to pump data through the transformation rules. These can be systematically derived, in a way not possible when writing code from static requirements.

This automatic derivation is possible because the flowchart can be overlaid with all the functional logic involve in a system. Automated mathematical algorithms can then be applied, to identify every possible path through the model, generating test cases which cover every combination of inputs and outputs (cause and effect or homotopic analysis can be used to do this).

As the test cases are linked directly to the model itself, they cover all the logic defined in it. They therefore provide 100% functional coverage, so that using a flowchart model to work towards having complete documentation is equivalent to working towards completely testing an ETL routine. A further advantage of this method is that testing becomes measurable. Because every possible test case can be derived, testers can determine exactly how much functional coverage a given set of test cases provides.

### Optimization: Test More in Fewer Tests

Automated optimization algorithms can then be applied, to reduce the number of test cases down to the bare minimum, while retaining maximum functional coverage. These combinatorial techniques are made possible by virtue of the logical structure of the flowchart, where a single step in the cause and effect logic (a block in the flowchart/test component) might feature in multiple paths through the flow. Fully testing the ETL routine then becomes a matter of testing each individual block (operator), using one of multiple existing optimization techniques (All Edges, All Nodes, All In/Out Edges, All Pairs). A set of 3 test cases, for example, might in fact be enough to fully test the logic involved in 5 paths.

At the financial services company mentioned above, this proved extremely valuable for reducing over-testing, shortening test cycles, and improving the quality of testing. Including the time taken to model the flowchart, for example, it took 40 minutes to create 19 test cases with 95% coverage – in contrast to the 150 test cases with 80% and 18 times over-testing used before. In another project, it took 2 hours to create 17 test cases with 100% coverage: a drastic improvement on the 16% coverage achieved in 6 hours previously.

## 3) Automatically Create the Data Needed to Execute the Tests

Once test cases have been created, testers require data which can cover 100% of the possible tests in order to execute them. Such data can also be derived directly from the model itself, and can either be created automatically, or drawn from multiple sources simultaneously.

A synthetic data generation engine like CA Test Data Manager offers multiple ways to create the required data when using Model-Based Testing to drive ETL testing. This is because, in addition to functional logic, a flowchart can also be overlaid with all the data involved in a system. In other words, as the ETL rules are modelled, output names, variables and default values can be defined for each individual node. When the test cases are created, data needed to execute them can be automatically generated from default values, along with the relevant expected results.

Alternatively, using CA Agile Requirements Designer (formerly Grid Tools Agile Designer), system data can be quickly created using the Data Painter tool. This provides a comprehensive list of combinable data generation functions, seed tables, system variables and default variables. These can then be used to create data which covers every possible scenario, including "bad data" and negative paths. As each path is simply another data point, the data required to systematically test an ETL routine's ability to reject invalid data can be created wholly synthetically.

Finally, existing data can be found from multiple back-end systems in minutes, using automated data mining. This uses statistical analysis to discover patterns in databases, so that groups of patterns, dependencies or unusual records can be extracted.

## 4) Provision the Data in Minutes, "Matched" to the Right Tests

Usually, a combination of existing, production data and synthetic data will be preferable, where synthetic generation is used to bring the coverage up to 100%. What's critical for efficient ETL testing is that the "Gold Copy" data is stored intelligently, so that the same data sets can be requested, cloned and delivered in parallel. This, then eliminates the delays caused by data constraints.

The first step to intelligent data storage is the creation of a Test Mart, where data is "matched" to specific tests. Each test is assigned exact data, while data is matched to defined, stable criteria, not specific keys. Test Matching data can eliminate the time otherwise spent searching for data among large production data source, as data can be retrieved automatically from the Test Data Warehouse, or can be mined in minutes from multiple back-end systems.

The Test Data Warehouse serves as a central library, where data is stored as re-usable assets, alongside the associated queries needed to extract it. Data pools can then be requested and received in minutes, linked to the right test cases and expected results. The more tests which are run, the larger this library becomes, until practically every data request can be performed in a fraction of the time.

Crucially, data can be fed into multiple systems simultaneously, and is cloned as it is provisioned. This means that data sets from numerous source databases are available to multiple teams in parallel. ETL testing is no longer a linear process, and the long delays created by data constraints are removed. The original data can be maintained as changes are made to the model, enabling teams to work on multiple releases and versions in parallel. This 'version control' also means that changes made to the ETL routines are automatically reflected in the data, providing test teams with the up-to-date data they need to rigorously test transformations.

## 5) Execute the Data Against the Rules and Automatically Compare the Results

Once testers have the tests needed to fully test an ETL routine, and the data needed to execute them, the validation itself must also be automated if ETL testing is able to keep up with the changing requirements.

### Using a Data Orchestration Engine

One way to do this is to use a test automation engine. CA Technologies offers the advantage of doubling up as a data orchestration engine, meaning that the data, matched to specific tests and expected results, can be taken, and pumped through a validation rule. This is "Data-Driven Automation" where, for example, a test harness created in a data orchestration engine can take each row of an XML file, defined in the flowchart, and execute it as a test.

This will then provide a pass/fail result, based on the expected results defined in the model. This automates both the execution of tests and the comparison of the actual versus expected results. Testers no longer have to manually copy scripts from the data target to the data source, and also avoid the painstaking, error-prone process of having to compare every individual field produced by the transformation.

### Using CA Test Data Manager

Alternatively, once the expected results have been defined, CA Test Data Manager can be used to automatically create pass/fail reports based upon them. This automates the otherwise manual data comparisons, but SQL still has to be copied from source to target.

First, synthetic data is defined in the source system, using the various techniques set out above. A Data Pool can then be created to simulate the ETL process, copying data from the source to the target. A valid set of data might be copied, to test that it is not rejected, as well as a data set with errors, to make sure invalid data is rejected. By creating a further table to store the Test Conditions and results, and creating a Data Pool-Based Table with Test Cases and Data Conditions, the expected results can be automatically compared to the actual results. Any missing or erroneous data can then be identified at a glance.

## 6) Automatically implement change

One of the greatest advantages of Model-Based Testing for ETL validation is the ability to react to high velocity change. Because the tests cases, data and requirements are so closely linked, a change made to the model can be automatically reflected in the test cases and associated data. This traceability means that, as ETL routines rapidly grow more and more complex, testing can keep up, and does not create bottlenecks in the Application Delivery pipeline.

With flowchart modelling, implementing a change becomes as quick and simple as adding a new block to the flowchart. Completeness checking algorithms can then be applied, to validate the model, and make sure that every piece of logic has been properly connected into a complete flow. If using CA Agile Requirements Designer, the Path Impact Analyzer can be used, to further identify the impact of the change on the paths through the flowchart. The affected test cases can then be removed or repaired automatically, with any new tests needed to retain 100% functional coverage generated automatically.

This eliminates the time wasted checking and updating tests by hand, while automated de-duplication has been proven to shorten test cycles by 30%. At the business intelligence team mentioned above, Model-Based Testing introduced massive time savings into the ETL process. Whereas it had taken 7.5 hours to check and update test cases when a single ETL rule changed, it only took CA Agile Requirements Designer 2 minutes. What's more, of the total test cases, only 3 had actually been affected and so needed to be updated.

As described, version control of data also means that testers are provided with the up to date data they need to fully test an ETL routine, even after it changes. Because test data is traceable to the model, when the requirements change, the changes are automatically reflected in the relevant data sets. Data is available across releases and versions in parallel, while the data needed for efficient regression testing is preserved in the Test Data Warehouse. Interesting or rare "bad data" can also be locked, to stop it being eaten up by another team, and to prevent it being lost during a data refresh.

**Section 4**

## Summary

Introducing a higher degree if automation into ETL testing is imperative for any organization striving for the Continuous Delivery of high quality software. An especially high degree of manual effort remains in ETL validation, from manually writing ghost code from static requirements, to sourcing the required data and comparing the results. Model-Based Testing and intelligent Test Data Management can be used to automate each and every one of these tasks, while allowing numerous teams to work from the same data sources in parallel.

Model-Based Testing "shifts left" the effort of ETL testing, concentrating the bulk of the work into the design phase. From there, every asset needed for ETL testing can be automatically derived in a fraction of the time. Test cases which provide 100% functional coverage can be automatically generated, and are linked to independently defined expected results. Each test is further "Matched" to the exact source data needed to execute them which, if stored in the Test Data Warehouse, can be provisioned to numerous teams in parallel.

Because of the close link created between tests, data and the requirements, the tests needed to re-test an ETL routine can be rapidly executed after a change is made. The time spent creating the initial model is therefore quickly outweighed by the time saved compared having to manually create, update and re-run tests. Model-Based generation also offers the substantial benefit of re-usability, where test components can be stored as sharable assets, linked to data and expected results, in the Test Data Warehouse. The more tests which are run, the larger library grows, until testing new or updated ETL rules becomes as quick and easy as selecting from existing components.

ETL testing no longer creates bottleneck in application delivery, and can keep up with the pace with which data-driven businesses grow. The testability of ever-more complex routines is maintained, so that testing can handle the variety and volume of data collected, and does not prevent the Continuous Delivery of quality applications.

Connect with CA Technologies at ca.com

CA Technologies (NASDAQ: CA) creates software that fuels transformation for companies and enables them to seize the opportunities of the application economy. Software is at the heart of every business, in every industry. From planning to development to management and security, CA is working with companies worldwide to change the way we live, transact and communicate – across mobile, private and public cloud, distributed and mainframe environments. To learn more about our customer success programs, visit **ca.com/customer-success**. Learn more at **ca.com**.

1 Jean-Pierre Dijcks, *Why does it take forever to build ETL processes?*, retrieved on 24/07/2015 from **https://blogs.oracle.com/datawarehousing/entry/why_does_it_take_forever_to_bu**

2 Alan R. Earls, *The State of ETL: Extract, Transform and Load Technology*, retrieved on 21/07/2015 from **http://data-informed.com/the-state-of-etl-extract-transform-and-load-technology/**

3 Grid-Tools, *Synthetic Data Creation at a Multinational Bank*, retrieved on 21/07/2015 from **https://www.grid-tools.com/resources/synthetic-data-creation-multinational-bank/**

4 IBM, retrieved on 20/07/2015 from **http://www-01.ibm.com/software/data/bigdata/what-is-big-data.html**

5 Jacek Becla and Daniel L. Wang, Lessons Learned from managing a Petabyte, P. 4. Retrieved on 19/02/2015, from **http://www.slac.stanford.edu/BFROOT/www/Public/Computing/Databases/proceedings/**

6 **http://etlcode.com/index.php/utility/etl_complexity_calculator**

7 Jean-Pierre Dijcks, *Why does it take forever to build ETL processes?*, retrieved on 24/07/2015 from **https://blogs.oracle.com/datawarehousing/entry/why_does_it_take_forever_to_bu**

8 ETL Guru, *ETL Startegy to store data validation rules*, retrieved on 22/07/2015 from **http://etlguru.com/blog/?p=22**

9 Bender RBT, *Requirements Based Testing Process Overview*, retrieved 05/03/2015 from **http://benderrbt.com/Bender-Requirements%20Based%20Testing%20Process%20Overview.pdf**

10 Huw Price, *Test Case Calamity*, retrieved on 21/07/2015 from **https://www.grid-tools.com/test-case-calamity/**

11 Bender RBT, *Requirements Based Testing Process Overview*

12 Software Testing Class, *Why testing should start early in software development life cycle?*, retrieved on 06/03/2015 from **http://www.softwaretestingclass.com/why-testing-should-start-early-in-software-development-life-cycle/**

13 datagaps, *ETL Testing Challenges*, retrieved on 24/07/2015 from **http://www.datagaps.com/etl-testing-challenges**

14 Robin F. Goldsmith, *Four Tips for Effective Software Testing*, retrieved on 20/07/2015 from **http://searchsoftwarequality.techtarget.com/photostory/4500248704/Four-tips-for-effective-software-testing/2/Define-expected-software-testing-results-independently**

15 Jagdish Malani, ETL: *How to handle bad data*, retrieved on 24/07/2015 from **http://blog.aditi.com/data/etl-how-to-handle-bad-data/**

16 Philip Howard, *Automated Test Data Generation Report*, P. 6. Retrieved on 22/07/2015 from **http://www.agile-designer.com/wpcms/wp-content/uploads/2014/10/00002233-Automated-test-case-generation.pdf**