

# Data Testing White Paper

## PART 1

### Testing Data and Data-centric Applications

## PART 2

### Pragmatic Data Testing

## PART 3

### Verifying Data in Production



**PART 1**

# Testing Data and **Data-centric** Applications

*Testing data and data-centric applications is a vital step for organizations that are using their data to drive their business. This article explains what data-centric testing is, and provides an overview of a methodology that can be used to implement data-centric testing in your organization.*

## Testing Data and Data-Centric Applications

Data is critical to organizations today. Businesses depend on accurate data to determine whether their business is doing well, make decisions on new products and offerings, and evaluate the success of current initiatives. Governments use data to determine what programs are successful and which are not. And non-profits use data to evaluate the impact they are making, evaluate fund-raising programs, etc.

There are countless examples of data being used to support critical processes today. However, most of the energy and effort in testing in IT today goes into testing the application functionality that creates or uses the data, and not into verifying the end result - the data itself. Often, data-centric processes, such as data integration, extract, transform, and load (ETL) processes, and analytic applications, are not tested or are only subjected to simple manual testing. On the other hand, application functionality (like application of business rules or implementation of a calculation) are tested extensively, but at an application level only.

***Testing is the single most overlooked aspect of a project.***

Why is this? For one, the state of the art in testing has concentrated heavily on testing application logic for many years, because that's where the interest was. People were focused on developing new and better applications. They wanted to be able to

develop these applications quickly, iterate on them rapidly, and build new ones when the business drivers changed. This has required flexible and powerful testing frameworks. After all, it is very difficult to make rapid changes to an application without having a solid set of test cases that can validate that the changes you just made are actually working.

It was often thought that the application would be the only thing working with the data, so if the application was "correct" then the data must be correct as well. In practical terms, though, most data today is used and manipulated by multiple systems. Now you have to verify all the applications that may have access to the data, that they all interact with it correctly, and that there are no issues with cross-interactions. The problem is even more complex in today's self-service driven world, because new applications that use your data can be added at any time, often without you being aware.

Another reason that data-centric testing hasn't been a focus is that testing application logic is "easy", while testing data is "hard". Developers in many cases don't like testing data, because it involves outside dependencies, above and beyond their code. Many testing approaches advocate isolating the code under test – for most applications, this means testing only the code (.NET, Java, etc.) and not the data that the code interacts with. There are even frameworks used for testing that exist simply to "mock" outside objects so the tests have no dependencies. This isn't necessarily a bad approach, and is quite valuable in many application testing situations. However, it can be a drawback for data-centric applications, as the tests often verify only the application logic, and don't validate how it works with real data.

## ***Businesses are becoming more data-driven.***

Organizations are realizing that the real value is in the data they collect and manage – the applications that work with the data are subject to constant change and replacement. In many cases, the data produced from the applications is more valuable than the application itself. So, while we continue to need to test application logic, we also need to test data. This is particularly true in the following cases:

- The data is business critical or a differentiator for the organization
- The data is interacted with from multiple applications or systems
- The data is part of a data-centric application or workflow (for example, data integration between systems, extract, transform, and load, or a data warehouse)

This document presents a methodology for testing data-centric applications and data. Not every piece of the methodology needs to be adopted to realize benefits from it. Any improvement to the testing has tangible results in reducing the number of defects in your data, as well as providing a reason for the developers and consumers of a system to feel confident in the results that it provides.

There are two main areas that this methodology covers – doing data-centric testing during development, and doing data verification for production or during system testing. Many of the same testing techniques can be used in both areas. However, the focus is a little different. Data-centric testing in development focuses on the testing necessary to make sure your data-centric applications produce the correct results. Data verification testing is focused on making sure that the systems that interact with the data produce consistent, verifiable results every day (or even more frequently).

## ***Benefits***

The major benefit of testing your data and data-centric applications is confidence in your data. One of the more common reasons for business intelligence initiatives to fail is that the users lack confidence in the results. By testing and verifying both the processes and the data that you are using, you can give the consumers of the data the confidence they need to make business decisions.

***According to Gartner, less than 10% of self-service BI initiatives will be monitored for consistency.***

Another benefit arises if your organization makes use of self-service BI. According to Gartner, less than 10% of self-service BI initiatives will be monitored for consistency. That can create major issues for both the accuracy of the reporting, and adherence to regulatory requirements.

Testing data-centric applications also leads to overall cost improvements. The earlier in the development cycle that defects are discovered, the easier and less costly it is to correct them. By incorporating robust testing into the development process, the maintenance and update costs can be greatly reduced. True, it does require a little more time upfront to create the tests, but it pays off heavily.



## Challenges

*One of the biggest challenges with testing data-centric applications is that you are interacting with data.*

To test it well, you need a set of data that addresses the test scenarios. Depending on the goal of the test, you may need a small, static set of data that represents some specific expected data details, or you may need a much larger set of test data that represents your production data. Managing these data sets can be challenging, as the creation of good test data can be time consuming. Simply taking a copy of the production data for testing purposes is not an option for many organizations, due to privacy concerns and regulations.

Related to managing the test data is the problem of keeping the data and the tests synchronized. As the database schemas are updated with new columns, tables, etc. the test data sets and the tests themselves need to be updated to reflect the current state.

Another major challenge with data-centric testing is that the tools haven't progressed at the same rate as the application tools. It's difficult to automate data testing, and even with tools that support it, you may find yourself pulling various technologies together with duct tape in order to assemble a working solution.

Another challenge is the time it takes to create the tests. Often, testing is the first area to suffer when projects fall behind, and it can be easy to think that taking time from testing to complete other parts of a project will be okay. However, this often creates a downward spiral – the parts of the project that aren't being tested create larger numbers of defects and rework, which can take more time away from testing, which just repeats the cycle. In addition, data testing in particular is time consuming – managing the test data, as mentioned above, can require a lot of effort.



## Tools Used

*As mentioned in the previous section, the tools available for data-centric testing are, for the most part, lacking in several noticeable ways.*

One, most tools are targeted to a particular tool or technology, and don't provide a way to use the same testing approaches and logic across the different technologies that an organization may use. A certain amount of that is expected, as it is quite difficult to cover every possible data-centric technology available. Often the tools focus on one specific technology. As an example, there are testing tools for Microsoft SQL Server relational databases. However, you have to use a different tool, and learn a different skillset, in order to test SQL Server Reporting Services reports. This lack of technology coverage adds to the complexity of producing a full testing solution.

To some degree, you can work around this by pulling multiple tools together, and scripting the interactions between them. However, not all tools support the automation necessary for that approach, and it doesn't reduce the need to have and maintain multiple tools and the skillset necessary to use them.

Any of the "x" Unit frameworks can make a good foundation for performing data-centric testing. However, you will need to spend some time developing an additional layer of functionality to make interacting with the database and other data focused applications easier. In addition, this layer will ensure consistency in how the testing is performed.

You should also consider the people who will be developing and executing the tests when selecting

tools. If your people are familiar with .NET or general programming languages, there are a broader array of choices. On the other hand, if your people don't spend a lot of time using .NET, then you will want to use tools that provide a friendly interface for creation of the tests.

*As you are looking for tools to drive your data-centric testing initiatives, please keep the following criteria in mind:*

### **AUTOMATED**

Automated testing support is critical to any modern testing initiative. You should be able to execute most, if not all, of your tests without requiring any human interaction. This enables you to run tests while you do other things, freeing up resources and time for more critical tasks. It also means that the tests are executed consistently. Manual testing introduces the chance of human error – perhaps a tester forgets to execute a test or a set up step. Automated testing means that you get exactly the same tests executed the same way, every time.

### **TECHNOLOGY COVERAGE**

Look at what technologies you need to be able to test – do you only work with SQL Server or Oracle? Do you have ETL tools or BI tools in the mix? Which of those are important to validate? (That last one is a trick question – they are all important) Now, compare that to the tools that you are looking at. Do they support only one technology, or do they cover multiple ones? How many tools total will you have to invest in to get complete coverage?



## *Look for tools that support the 3 A's: Arrange, Act, Assert*

### **SUPPORT FOR THE THREE A'S (ARRANGE, ACT, ASSERT)**

A very common pattern in testing is the three A's – Arrange, Act, Assert. Arrange involves the setup of the necessary conditions for the test. Act involves invoking the actual code or application being tested. And Assert is where you verify assertions about the state of things after the code has been executed. This is a common pattern because it works very well and there are many resources on successfully using it. Look for tools that support it.

### **TEST DATA MANAGEMENT**

Since data-centric testing is, well, data-centric, managing test data is a vital part of the process. Unfortunately, most tools today do not offer this as an integrated function. You may be able to use other tools to manage the test data, but this again increases the number of different tools you have to integrate.

### **RESULT REPORTING**

Finally, for data-centric testing and data verification, reporting the results of the tests often goes beyond the typical test tool approach. Particularly for data verification, the consumer of the test results may not be in IT, and may need a friendlier way to view and process the results.

The methodology discussed here can be implemented using a variety of tools. However, you will find that some tools are better suited to it than others. The samples shown in this series of articles will use LegiTest (<http://pragmaticworks.com/Products/LegiTest>), which is a tool developed with the methodology in mind, so it fits very well. However, as mentioned, the approaches discussed in the articles can be implemented with other tools and a bit of ingenuity, they may just require more work to set up and use.



## *People / Roles*

There can be a wide variety of people involved in testing. In the context of Pragmatic Data Testing, though, you will focus on a few key roles. Please note that these roles do not have to be different people, though each role has a specific focus to the testing.

## *Involve these roles in your testing strategy: Developer, Development Tester, QA*

### **DEVELOPER**

In some organizations, it's felt that developers shouldn't be involved in the testing process. Instead, they should just focus on producing code and let the Quality Assurance (QA) group handle testing. This is a good way to produce lots of code that nobody has tested. Developers are integral to the testing process, because they are the only ones that know what code they have written. At a minimum, they need to work with the testers to ensure that everyone has a clear understanding of the requirements and the implementation, so that the tests can accurately exercise the system.

In many organizations, particularly those adopting test driven development (discussed further in the next article), there is a trend towards developers actually creating their own tests. An additional benefit you may find is that when creating automated tests, developers are often the best equipped to

do that well. In data-centric testing, it is often necessary to have a developer who really understands the data participate in the test creation, or at a minimum, educate the testing team on working with the data. If you are really focused on improving your data-centric testing, you are likely to have at least a portion of your developer's time spent on testing.

Developers would still be primarily involved in development testing for functionality, at the unit and system testing level. These will be defined in a later section of this series of articles. Data verification is typically not in their area of responsibility.

### **DEVELOPMENT TESTER**

This is a more specialized role in organizations that focus on having extremely thorough automated test coverage. These are testers who are focused on testing and quality, but develop automated testing to



verify the systems they work on. They differ from developers in that they are typically not adding new functionality to the systems, instead, they are writing automated tests that verify the new and existing functionality of the system. This is a role that fits very naturally with the Pragmatic Data Testing approach. Development Testers have much the same responsibility as developers, in that they focus on testing functionality, through unit, integration, and system testing.

## QUALITY ASSURANCE

Quality Assurance encapsulates the traditional testing in many organizations. Often the people in QA focus primarily on “black box” testing – that is, they don’t know the internals of the system, but rather what goes in and what should come out for the application. Particularly when it comes to data-centric applications, they make focus on the application side, and not test details of the underlying data. What data testing is done is typically done manually.

Adopting a testing approach for data-centric applications tends to change this role more significantly than the other roles. The focus for your QA resources becomes a) understanding the data requirements of the application, b) developing automated test scripts for that data, and c) testing the bigger interactions of the data-centric application or system under test. The QA role is usually responsible for testing the system functionality at a macro level, rather than smaller units of code. They should be involved in testing at the system level, as well as performance and load testing. In addition, the QA role is heavily involved in data verification testing, which will be defined in a later section in this series.

## Conclusion

This has covered a brief introduction to data-centric testing. It also explained why it is a critical factor in today’s data-driven world. The quality, accuracy, and reliability of the data your organization works from is not something that can be left up to chance, or the hope that “nothing will go wrong”. Instead, you need to be able to have confidence in your data, and be able to prove that it is accurate, and adheres to the organizational requirements for your data.

The next sections of the series will go into more details on the Pragmatic Data Testing methodology. It will focus on how you can adopt data-centric testing as part of your development processes, along with the different types of testing that you can consider as part of your development of new and enhanced functionality and data. You will also see how to apply data verification testing to data throughout your organization, which can increase your confidence in the data you work with every day.

---

**FOR MORE INFORMATION ON DATA-CENTRIC TESTING AND TO REQUEST A DEMO OF OUR PRODUCT LEGITEST, PLEASE VISIT [PRAGMATICWORKS.COM](https://pragmaticworks.com).**

**PART 2**

# Pragmatic Data Testing

*Testing data and data-centric applications is a vital step for organizations that are using their data to drive their business. This article explains what data-centric testing is, and provides an overview of a methodology that can be used to implement data-centric testing in your organization.*

## Testing Data-Centric Code in Development

In this section, you will learn more about the types of testing that can be used with data-centric applications when they are under development. For the most part, these line up with traditional application testing approaches, but there are some differences to accommodate the data focus.

### WHAT IS CODE?

When you think of code, you may picture a monitor full of C++, C#, Java, or another programming language. However, code has a much broader application. SQL, SSIS packages, SSRS reports and many other languages and tools that you use in data-centric applications are also considered code.

*Code can be any set of instructions to the computer or an application that produces an output.*

When testing code for data-centric applications, we need to define what code actually is. As mentioned, you might think of code as .NET or Java code, something that is compiled into an executable order to be run by the computer. But in a more general sense, code can be any set of instructions to the computer or an application that produces an output. From a testing standpoint, if we have an input (the instructions) and an expected output (the results), we have something to test.

So what types of computer instructions can this include? It certainly includes traditional application code, but it also includes database code, and instructions to the computer for other, specialized applications, like data integration tools and business analysis and reporting tools. When you create an

SSIS package, or an SSRS report, you are creating a set of instructions to an application that specify how you would like to retrieve data, manipulate it, and then either store it somewhere (for SSIS) or display it to a user (for SSRS). A stored procedure would be a set of instructions on how to retrieve, combine and return data to the user.

The Data Definition Language (DDL) that you use with databases is also a set of instructions to the database engine. The DDL details the tables, views, foreign keys and other objects that should be created, altered, or deleted in the database.

SQL Server Analysis Services uses a similar language, XML for Analysis (XMLA), which supports creating, modifying and deleting objects on an Analysis Server. There is also the Multi-Dimensional Expression language (MDX) and Data Analysis Expressions (DAX) which allow for querying and shaping results from Analysis Services.

When you look at it from that perspective, most, if not all, of the things that we create for data-centric applications would be considered code. And since they are code, they should be tested. The rest of this section will discuss how to do this.

## Types of Testing

There are several types of testing that are done on traditional applications. Most of these are directly applicable to testing data-centric code as well, though you might find it necessary to tweak the approaches a bit to make them work well.

### UNIT TESTING

Unit tests focus on small units of work (logical groupings of code) in the system under test and check assumptions about the behavior of that code. Generally, unit tests are implemented by

the programmer in conjunction with the development of the code. These are tests that you would run against the code you have just completed to ensure it works as expected. Once completed, you would keep the unit tests to form the backbone for regression testing, and to act as a verifiable check on whether the code performs as expected.

Automated unit testing is a standard practice in application development. In application development unit testing, efforts are made to isolate the unit of work being tested from any outside dependencies, including the database or the file system. This is challenging for data-centric applications, and you will find that it can create additional work to abstract away the external systems the code interacts with. In some cases, the tools you use for data centric applications don't support this level of isolation. SSIS, for example, is very difficult to unit test in a fully isolated manner, as some components require a connection to a database in order to function. Rather than getting too wrapped up in debates about whether this truly meets the definition of a unit test, we prefer to take a practical approach and work with what we have. If you like, you can refer to unit tests that interact with outside dependencies as micro-integration tests.

When creating unit tests, you should control the inputs to the code. The tests should verify that the output of executing the code delivers the results you expected. In some cases, the same unit tests may be driven through a variety of inputs, so that the same unit of code can be tested with many different inputs. This verifies that the code produces the correct results for all the tested inputs. These are often referred to as data driven unit tests.

Unit tests should also be isolated from each other. You should create unit tests so that any unit test is atomic and can be run independently of other unit

tests. This isolation means that the tests can be run in smaller subsets easily, even down to a single unit test, and that you do not have to run them in any particular order. This does require that each test sets up the appropriate preconditions for the test, creating any necessary data prior to the test execution and cleaning it up afterward.

Unit tests should ideally be fast to execute. The longer the unit tests take to run, the less likely you are to execute them. Since much of the benefit of unit tests comes from running them frequently, you should ensure that your tests run as fast as possible. You can accomplish this by making sure your unit tests are done against small sections of code and that they do not cover too much of your application at once. If the tests are created properly, you can also run them singly or as a smaller subset to get faster feedback on the section of code you are testing. Another key point is that data-centric unit tests should focus on small sets of data. The point is to exercise the functionality of the code unit, not to performance test it.

#### **WHAT TO INCLUDE IN UNIT TESTING**

Often, the reason that developers object to including external dependencies in their unit tests relates to performance. Typically, external resources like databases or file systems are orders of magnitude slower than the same operation carried out in memory. However, you can work around this in many cases by following the guidance in the preceding paragraph. In addition, running subsets of the unit tests when you are testing interactively, and the full test suite when doing a full integration, can lead to a better experience with resource constrained tests.



## TEST DRIVEN DEVELOPMENT

Test Driven Development (TDD) is a practice in which unit tests and code are written in conjunction with each other. As a developer, you would write small, incremental tests, then write the code to satisfy those tests and ensure they pass. You start by creating a test that implements a specific test case. This test will fail initially, so you write the code necessary to make the code pass. Then you refactor the code until it is clean and elegant, while maintaining the passing status of the test. You would then repeat the process for the next set of functionality, until the code delivers the expected results.

This approach has a number of benefits. One, since you are creating tests in conjunction with the code you are writing, test coverage of the code is much higher. Two, it keeps your efforts focused on implementing the code that meets the requirements. Three, one of the most important benefits it offers is increased confidence as a developer. When you develop using a TDD approach, you always know where your code stands. Because you are working in small increments, you are never very far away from a system that passes all the tests. If all tests are passing, then you know all implemented code is working as designed. If you make modifications, you will get immediate feedback on whether the change has impacted other functionality in the system. This makes it much easier to make updates and refactor code.

***When you develop using a TDD approach, you always know where your code stands.***

## INTEGRATION TESTING

Integration tests generally span multiple units of work, and verify that larger portions of the system work together correctly. This may involve interacting with multiple subsystems, for example, verifying that a report can correctly retrieve information from a database, perform calculations on the result, and then display that to the user. Integration testing further ensures that code or modules developed by one developer works properly with code from another developer, and that it doesn't have unintended impact on other parts of the system.

As noted above, often testing of data-centric applications falls into the category of integration testing, as it can be difficult to isolate the application being tested from the underlying data. In many cases, isolating the application from the data can actually hamper the effectiveness of the tests, as the data is central to the requirements for the application. Rather than getting too concerned about what type of testing is being performed, we prefer to take a pragmatic approach (no pun intended) and focus on creating the tests that best verify the system under test.

Integration tests generally take a block box approach to the code, that is, the tests don't assume knowledge about the internal implementation of the code itself. Instead, they focus on providing inputs that model the requirements and expected inputs of the system, and verify that the output from the system matches the expected results.

Creating integration tests for data-centric applications is much like creating unit tests, in that you generally have to set things up for the test, invoke the part of the system under test, and then assert that the new state of the system matches an expected result. However, it focuses on larger sets of functionality. To create an integration test, you

would define a usage scenario for the application, the expected end state, and test data that supports the scenario. For example, a scenario for an ETL process for a sales data mart might look like the one in Figure 1 - Sample Integration Scenario.

**Scenario:** A customer places a new order.

The customer was recently married, and as part of placing the order, the customer notes that both their name and address have been changed.

**Application Functionality:** The Load\_DimCustomer package should be executed to pick up the changes from the SalesStage staging database and load them into the SalesDM datamart.

**Expected Results:** The customer name change should be handled as a Type 1 change – all historical customer records should be updated to reflect the

new name. The address change should be handled as a Type 2 change – a new version of the customer dimension record should be created with the new address, and marked as the current record.

Just as can be done with unit tests, integration testing can be automated. In many cases, the same framework or harness that is used for unit testing can also be leveraged for integration testing, as the general structure of the tests is very similar. The primary difference is in the granularity of what is being tested, and how hard you try to isolate the code being tested from other systems. Using a framework also enables you to assemble integration tests into suites that can be run together, and the ability to include your integration tests as part of the build process. You will find that using an automation approach to your integration tests provides an immense amount of value, and is required to take advantage of integration tests for regression testing.

**Source Data:** The source data for the test.

ACCT. ID	NAME	ADDRESS	CITY	REGION	POSTAL CODE	CURRENT
75	Jane Smith	123 Elm Ln	Tampa	FL	33601	N
75	Jane Smith	<b>111 Oak Rd</b>	<b>Tampa</b>	<b>FL</b>	<b>33601</b>	<b>Y</b>

**Target Data (before):** The data in the dimension before the load is executed.

ACCT. ID	NAME	ADDRESS	CITY	REGION	POSTAL CODE	CURRENT
75	Jane Smith	123 Elm Ln	Tampa	FL	33601	N
75	Jane Smith	<b>111 Oak Rd</b>	<b>Tampa</b>	<b>FL</b>	<b>33601</b>	<b>Y</b>

**Target Data (after):** The data in the dimension after the load is executed.

ACCT. ID	NAME	ADDRESS	CITY	REGION	POSTAL CODE	CURRENT
75	Jane Smith	123 Elm Ln	Tampa	FL	33601	N
75	Jane Smith	<b>111 Oak Rd</b>	<b>Tampa</b>	<b>FL</b>	<b>33601</b>	<b>Y</b>

Figure 1 - Sample Integration Scenario – bold indicates expected changes

## SYSTEM TESTING

System testing tests the system as a whole. It generally focuses on validating that the system meets the overall requirements for the solution, and often includes user interface, usability, and load and performance testing. For data-centric applications, system testing may need to take on some additional steps to truly validate the system. For example, it becomes much more important to validate the underlying data in the system when dealing with data-centric applications.

Since individually reviewing each row of a table in a database isn't practical, you will need to apply tools to this problem. Good tools are capable of comparing expected data with the actual data, and ideally will have the capability to do this against either a comparable, known good database, or against control totals. Control totals are things like a customer count, the total amount of sales for the month of December, or some other aggregated value that gives you confidence that if the aggregate matches, the underlying details are likely to match as well.

### TAKE NOTE


Be careful with tools that only allow you to do a row-by-row, column-by-column comparison. Often, when dealing with changes to data-centric applications, updates to the system require modifications to the data structures. When that happens, it can break the functionality of many data comparison tools. Rather, you should look for tools that support both a tabular comparison, as well as the ability to compare aggregated values.

## REGRESSION TESTING

Regression testing is testing done to validate that new changes to a system have not adversely affected existing functionality. In basic terms, this is something that most of us have seen when we have fixed one problem, only to see something that we thought was unrelated suddenly stop working in another part of the system. Regression testing is all about finding unintended consequences. It also is used to ensure that corrected issues do not resurface in later versions of the system by continuing to validate those fixes for subsequent versions.

Regression testing is a problem spot for many organizations, because it doesn't involve testing what has changed, it involves testing everything that has not been changed. People are not very good at anticipating the side effects of their changes. In addition, in cases where testing is done manually, it can be easy for people to not test as thoroughly for areas of the system where they don't expect to find issues.

If you are using Pragmatic Data Testing approach, you will get regression testing without having to do any additional work. By creating automated unit and integration tests, you establish a baseline of functionality testing that can be easily re-executed as needed. So for subsequent changes, you can continue executing the same tests that you have already created, verifying that nothing unexpected has changed in the system. This does mean that you will need to make sure any new changes to the system are also covered by automated tests, particularly any defects that are corrected. Once you have a test that validates that a particular defect is fixed, you can have confidence that if it shows up again, you will catch it during testing, rather than in production.



*Use of automated tests for regression tests is incredibly valuable, particularly if the system you are working on experiences a lot of change.*

Use of automated tests for regression tests is incredibly valuable, particularly if the system you are working on experiences a lot of change. It also means that your investment in automated tests gets more valuable every time you make a change – just look at all the time you are saving over having to manually re-execute tests, or the costs of having a regression in functionality make it through testing unnoticed.

## **LOAD AND PERFORMANCE TESTING**

Load and performance testing is testing to determine if the system handles operations at the expected volume of the production system in an acceptable timeframe. Load and performance testing generally assumes that the functionality is correct, and focuses primarily on timeframes and volume. Pragmatic Data Testing doesn't focus specifically on load and performance testing, as existing approaches for this type of testing work well for data-centric applications. However, it can be helpful to use a test framework that allows you to easily time operations that are being performed.

One item to note is that load / performance tests should not be combined with tests that verify functionality. Developer focused tests need to

run quickly, so that the developer doesn't spend time waiting on them to complete. Load tests in particular, and most performance tests, require a large significant volume of operations, so the tests tend to take more time. This doesn't mesh well with quick functionality tests. You will find it much easier to manage if you keep a clear separation between these types of testing.

## ***Specific Technologies***

The information below contains information on the types of functionality that you should consider testing for data-centric applications. It also provides information on the how-to of actually automating test cases that verify this functionality. However, we can not document all the possibilities around that for this article, due to space constraints, so it keeps things at a fairly generic, pseudo-code level. We will use the common Arrange, Act, Assert pattern for the pseudo-code. When looking at a test framework, it should be capable of handling the requirements of the scenarios below. LegiTest was developed with the Pragmatic Data Testing approach in mind, so it enables the below scenarios. Other testing frameworks can be used as well, though some of them may require additional work.



## SQL

When working with SQL databases, you will want to test the structure, the data and the various ways that the data can be manipulated.

This primarily includes the tables and views in the database. From a testing standpoint, you want to verify that the object exists, and that it contains the correct columns with the correct data types. You may also want to verify that calculated columns and check constraints are set up properly, as shown in Figure 2- Test for table creation.

### ARRANGE

- Create an empty **Sales** database to test **Customer** table creation.

### ACT

- Using **Sales** connection
  - Execute **CreateCustomer.sql**
  - Execute **InsertTestCustomer.sql**

### ASSERT

- That the **Customer** table contains a column **FirstName** (Type: VARCHAR, Length: 50)
- That the **Customer** table contains a column **FirstPurchaseDate** (Type: DateTime, Default: GETDATE())
- That the **Customer** table contains a column **YearsAsCustomer** (Type: Int, Calculation: DATEDIFF(year, FirstPurchaseDate, GETDATE()))
- That the **Customer** table contains a row where **FirstName** = "John", **LastName** = "Smith", **FirstPurchaseDate** = "2012/01/01", and **YearsAsCustomer** = 3.

Figure 2- Test for table creation

You will also want to test stored procedures, triggers and functions that are used in the database. Verifying that these are properly implemented requires that you set up the necessary prerequisites, inputs, and verify the outcome of executing the routine. An example of testing a stored procedure is shown in Figure 3- Test for a stored procedure. The requirement for the stored procedure is that it evaluates the customer records looking for possible duplicates based on names and addresses. It requires that there are customer records to evaluate, that a threshold for similarity be provided as input, and that a list of customer that exceed the threshold be provided as output.

### ARRANGE

- Using **Sales** connection
  - Execute **CreateCustomer.sql**
  - Execute **CreateCustomerDeduplicatorProcedure.sql**
  - Execute **InsertTestCustomerRecords.sql**

### ACT

- Using **Sales** connection
  - Execute **EvaluateCustomers** procedure (**Threshold** = 0.90).

### ASSERT

- That the result contains 3 rows where **FirstName** = "John", **LastName** = "Smith", and **Similarity** >= 0.90.

Figure 3- Test for a stored procedure

## SSIS

SSIS can be among the most heavily used component of the BI stack, and thus requires rigorous testing. An area for special focus is the Data Flow Task, which handles the bulk of the work in most packages. Two types of tests for the data flow are fairly standard, validating the number of rows loaded, as well as comparing the loaded data against the source.

Further, your requirements may also dictate performance levels that a package must achieve. These too should be validated to ensure, as the example in Figure 4- Example SSIS Package Test illustrates, the package executed within a predefined run time. As noted earlier in the performance and load testing section, though these should be split into separate tests if they are performing load or volume testing, so they can be isolated as necessary.

### ARRANGE

- Create a connection to the **LoadCustomer** package.
- Create a connection to the **Sales** source system.
  - Insert test data from **CustomerTest.sql** script.
- Create a connection to the **SalesMart** target system.

### ACT

- Execute the **LoadCustomer** package.

### ASSERT

- That the **Sales Customer** table row count matches the **SalesMart DimCustomer** table row count.
- That the Sales Customer table rows match the **SalesMart DimCustomer** table rows.
- That the **ExecutionTime** property of the **LoadCustomer** package is  $\leq 60$  seconds.

Figure 4 - Example SSIS Package Test

Additionally, the Execute SQL Task often needs validation as it has the ability to modify data, or to return data that can have an impact on the other tasks in the package. Figure 5 - Validation of an Execute SQL Task Which Returns Data covers a test for an Execute SQL Task which runs a query and returns a value which is placed in a variable.

#### ARRANGE

- Create a connection to the **LoadCustomer** package.
- Create a connection to the **SalesMart** target system.
  - Insert test data from **DimCustomerTest.sql** script.

#### ACT

- Execute the **GetMaxID** Execute SQL Task.

#### ASSERT

- That the value of the **NextID** variable is = 6.

*Figure 5 - Validation of an Execute SQL Task Which Returns Data*

In addition to Execute SQL Tasks which return data, many also perform commands against the target system. These can be tested by querying the target database. Figure 6 - Validation of an Execute SQL Task Which Alters Data is an example of a test that verifies an Execute SQL Task truncates a target table.

#### ARRANGE

- Create a connection to the **LoadCustomer** package.
- Create a connection to the **SalesMart** target system.
  - Insert test data from **DimCustomerTest.sql** script.

#### ACT

- Execute the **Truncate DimCustomer** Execute SQL Task to truncate the target table.

#### ASSERT

- That the **DimCustomer** table row count is = 0 (zero).

*Figure 6 - Validation of an Execute SQL Task Which Alters Data*

## SSAS

Analysis Services has three main areas that you will want to test: dimensions, measures and calculated members. With dimensions, there are a few different aspects to consider. First, you should ensure that expected members exist in the database. Next, you should validate the count of those members. Finally, you should test the accuracy of any calculated members of the dimension. Figure 7 - Test for SSAS Dimension walks through an example scenario of testing a product dimension.

### ARRANGE

- Create a connection to **SalesCube** SSAS database
- Create a connection to the **SalesMart** source system
  - Insert test data from **ProductTest.sql** script.

### ACT

- Process the **Product** dimension to ensure data is valid and up to date

### ASSERT

- That the **Product** dimension contains an **Unknown** member.
- That the **Product** dimension contains a **Widget123** member.
- That the **Product** dimension member count matches the **Product** table row count.
- That the **Product** dimension **Discontinued Products** member **Sales Amount** = **\$100,000**.

Figure 7 - Test for SSAS Dimension

The second area of focus for Analysis Services testing revolves around measures. These tests tend to be straightforward, comparing the aggregated values from the cube against similar aggregations from the source systems. Figure 8 - Test SSAS Measures illustrates this scenario.

### ARRANGE

- Create a connection to **SalesCube** SSAS database
- Create a connection to the **SalesMart** source system
  - Insert test data from **SalesTest.sql** script.

### ACT

- Process the **Sales** cube to ensure data is valid and up to date

### ASSERT

- That the cube **Sales Amount** measure equals the table **FactSales SalesAmount** column total.

Figure 8 - Test SSAS Measures



Finally, it is important that you test the accuracy of the cube's calculated measures. This could involve calculating the expected value by hand, then hard coding it into the test. Alternatively, you may wish to recreate the calculation in a query against source data. Figure 9 - Test SSAS Calculated Measures represents the basic workflow for validating calculations.

#### ARRANGE

- Create a connection to **SalesCube** SSAS database
- Create a connection to the **SalesMart** source system
  - Insert test data from **SalesTest.sql**

#### ACT

- Process the **Sales** cube to ensure data is valid and up to date

#### ASSERT

- That the Sales cube **Avg Cost of Sale** measure matches the calculation  $\text{SUM}((\text{UnitPrice} * \text{Qty}) - (\text{UnitCost} * \text{Qty})) / \text{COUNT}(\text{SalesId})$  from the **FactSales** table.

Figure 9 - Test SSAS Calculated Measures

## SSRS

Testing around Reporting Services falls into two areas, validation of values and ensuring reports executed successfully in a predefined time period.

As part of the test, you will need to execute the report. After executing the report, you would retrieve a value from the report, and then compare that to a known value, whether it is manually set or calculated from the source data. Most commonly, the grand totals of the report are used for this purpose.

Retrieving values from SSRS reports can be complicated. You will often find it easiest to do this by exporting the report to an XML format. Once you have it in the XML format, you can use XPath queries to locate specific values in the report.

Figure 10 - Test an SSRS Report Value summarizes these steps for a sales report.

#### ARRANGE

- Create a connection to the **Sales** report.
- Create a connection to the **SalesMart** source system

#### ACT

- Execute the **Sales** report.

#### ASSERT

- That the **Sales Amount** grand total value from the Sales report is equal to the **FactSales** table **SalesAmount** total.

Figure 10 - Test an SSRS Report Value

Many reports have performance requirements associated with them. You can validate that the report runs in the expected timeframe by capturing the runtime for the report, and comparing that to an expected value. Again, as noted in the performance and load testing section, these tests should be split into separate tests if they are performing load or volume testing, so they can be isolated as necessary.

#### ARRANGE

- Create a connection to the **Sales** report.

#### ACT

- Execute the **Sales** report.

#### ASSERT

- That the **Sales** report execution time is  $\leq 60$  seconds.

Figure 10 - Test an SSRS Report Value



On the next page, we explore how to **manage test results**.

## Managing Test Data

One question that often comes up when testing data-centric applications is “How do I manage test data?” There are a number of possible ways to handle this, and which will work best depends on your environment and the tools you have available. Before getting into that, though, you should make sure you understand where and what types of test data are needed for your solution.

You need test data anywhere that you expect external input to your solution. Most data-centric applications deal with one or more databases. Each of these is a potential area where you will need to create or load test data. In addition, some data integration processes deal with text files or other non-relational sources — you will also need test data that represents these inputs. If your data-centric application is a data mart or warehouse, you may be wondering if you need test data for the warehouse itself — after all, you can just load the data from the source system as needed.

While this is an option, it’s recommended that you have separate test data for each data store / input source you deal with. The reason for this is it is difficult to create effective unit tests if you rely on processes unrelated to the unit of code under test to set up the test data. It’s very common that these upstream processes can be modified in a way that breaks downstream unit tests, so keeping them isolated is a better approach.

***It’s recommended to have separate test data for each data store/input source you deal with.***

When it comes to types of test data, it generally falls into two categories: specific data sets to validate known scenarios and large volumes of data that represents a broad sample of the types of data the solution may possibly encounter. The specific, well known data sets are most often used for unit and integration testing, and generally represent a small number of rows that are designed to exercise any conditional paths in the solution. These data sets cannot be randomly generated, as they need to provide specific values to make sure the test conditions are met. By the same token, extracting this data from production data generally doesn’t result in a data set that exercises all the conditional paths. Often, these data sets have to be created by hand. It’s very useful to leverage a test framework that has support for creating and managing these targeted data sets.

Large volume data sets are used for load and stress testing. They are also useful for sanity checks on the solution — verifying that out of range data is handled or that a sample of production data can be processed successfully. Random data generators and extracting data sets from production are valuable approaches to creating these types of data sets. If you do extract production data for test purposes, there will often be a requirement to mask or strip certain pieces of information. In these cases, it can be helpful to combine random data generation with the extract process, to fill in any gaps in the extracted data.

Regardless of the type of test data, look for test frameworks that include or can be modified to include the management of the test data sets. You will find creating tests much easier if you have simple ways to load test data, reset the data sources, and persist the test data outside of the database itself.

## ***Integration into Continuous Delivery***

Continuous integration is a development practice in which any changes checked into the team's source control system are immediately compiled, analyzed and tested, so that the developers get immediate feedback on the state of the solution. Continuous delivery builds on this with a set of processes that enables teams to build solutions in short iterations, while keeping it in a state that it can be released at any time. This approach can lead to much faster time to value, and enables users to see and respond to changes in the solution much more quickly. It is quite popular in traditional application environments, but there are some challenges in implementing it for data-centric applications. One of the challenges is that, in order to ensure the solution is in a releasable state at all times, you need to be able to test any changes quickly and efficiently. If extensive manual testing is required, it becomes very difficult to do this. This is why many people feel that automated testing of a significant portion of the solution is mandatory to truly implement continuous delivery.

***Continuous Integration & Delivery leads to much faster time to value.***

Once you have an automated suite of tests, it becomes much easier to have your data-centric applications participate in continuous integration and continuous delivery processes. Since the tests are automated, they can be added to integration and build processes. Most automated testing frameworks support this capability, at a minimum by using a command line tool. Some feature the option for direct integration of the test execution and result evaluation into the build tools.

You don't have to do continuous integration or delivery to use the Pragmatic Data Testing approach. However, you will find it much easier to implement continuous delivery if you follow the approach of automating the tests for your data-centric applications. Continuous delivery can add significant flexibility to your team's ability to deliver useful business results in a timely manner, and should be something that you evaluate when looking at improvements in your delivery process.

## ***Conclusion***

In this part of the series, we covered some important concepts on what sort of things you need to consider testing during development of your data-centric applications and what types of testing you should consider. There are also several examples of test cases that you can look at implementing in your automated testing framework of choice.

The next part of the series will discuss how testing your data and data-centric applications is still important even after development has finished, and the solution is in production. It will also cover several techniques that you will find useful for this type of testing.



**PART 3**

# Verifying Data in *Production*

*Testing data and data-centric applications is a vital step for organizations that are using their data to drive their business. This whitepaper explains what data-centric testing is, and provides an overview of a methodology that can be used to implement data-centric testing in your organization.*

## Verifying Data in Production

*In this section, you will learn about verifying data in production and how this is different than testing performed in the development process. Verifying data in production is a critical step for organizations that rely on their data for decision making.*

This step is performed after data integration processes are executed or after any changes to the data of the system. This verification includes several important aspects, but they can be summed up in the three R's: Is the data reconciled? Is it related? And is it reasonable?

- The reconciliation of data is performed by comparing counts, totals and balances between sources. For example, if the daily sales transactions in your source system total \$100,000, then the daily sales in your data warehouse should also be \$100,000.
- The relationship of data is verified by examining the categorization of the data and how it relates to other data. Continuing the example above, your data warehouse total sales might match the source system, but if 80% of the sales are recorded against an "unknown" product, the data isn't related in a useful way for decision making.
- The reasonableness of data is determined by looking at trends, history and tolerances. If historically your data warehouse daily sales total within +/- 10% of \$100,000 (a range of \$90,000 to \$110,000), seeing daily sales of \$200,000 would trigger some additional investigation to ensure the results were valid.

In the sections below, the different types of verification will be discussed in more detail.

These three types of verification are often implemented as queries against the data stores and applications in a data-centric system. There can be a large number of these queries to execute in a production system to verify all the data. These may not all be queries in the traditional relational or SQL sense, as you will often find the need to interact with systems that have non-SQL interfaces or use web APIs. In all likelihood, you will want to automate the process of interacting with these heterogeneous systems, running the queries and comparing the results, whether you use a homegrown system or implement a commercially available package like LegiTest (<http://pragmaticworks.com/products/legitest>). Key features to look for (or implement, if you are creating your own) are the ability to store the test results and to access and query the breadth of data sources that you use.

The supporting infrastructure needed to perform data verification effectively will be discussed further later in this whitepaper. First, however, we will discuss the differences between testing in development and verifying data in production.

## Testing in Development vs. Verification of Data in Production

*There are a number of similarities between testing in development and verifying data in production. They can leverage the same techniques and the same tools. However, there is a significant difference – and that is in your focus.*



**Testing in Development Goal**  
*Ensure code is working successfully*

**Verifying Data in Production Goal**  
*Verify the data manipulated is valid*

When you are testing in development, the goal is to make sure that the code is working successfully, so you typically control the inputs to the processes by using known test data. This gives you the ability to verify that the process produces the expected results. So, in this case, the focus of the testing is on the code functionality.

When you are verifying data in production, however, you don't control the inputs. You are now dealing with real, production data. So the focus shifts from verifying the process or code itself, to verifying that the data manipulated by the process is still valid and reasonable. You do this by verifying that the input matches the output, with any appropriate transformations or modifications accounted for. Since you no longer control the input, you also have to validate that the output is reasonable, based on past history.

In development testing, your tests will often deal with individual records, comparing expected and actual values at a detail level. However, since the data volumes in production systems can be quite large, and the data is expected to be transformed as it moves between systems, you don't usually focus production data verification on a line-by-line reconciliation. The performance impact from doing row-by-row validations in a production system usually outweighs the benefits. Rather, you would focus initially on testing aggregates and rollups of information. If these show that the data is incorrect, then you would go to a more detailed level of data, to identify the underlying issues that caused the incorrect data.

## Types of Verification

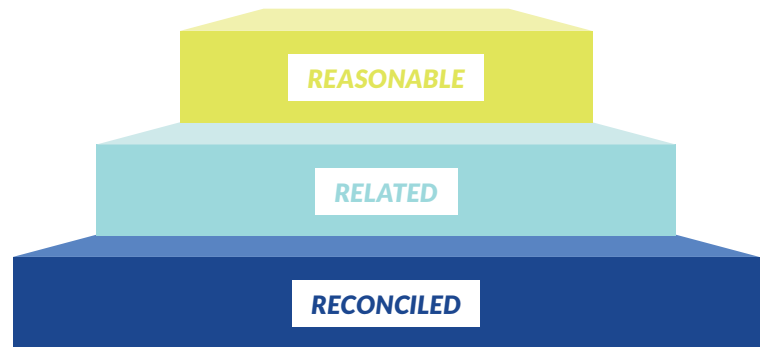
*As mentioned above, when you are considering data verification, there are many considerations. We like to sum these up as the 3 R's: Is the data reconciled? Is it related? And is it reasonable?*

The 3 R's are interconnected and build on each other. Reconciliation is a fundamental verification step and should be your first goal to implement. It provides a baseline of confidence in the data, as you will know that the totals and key values in each system matches. Verifying data relationships is the next logical step and adds a significant amount of value, as it ensures that data is related in a way that is useful for the business. Finally, you will want to verify that your data is reasonable. Implementing this depends on having appropriate reconciliation and relationship tests in place, as you can't verify that data is reasonable if it can't be reconciled. The ability to find and be alerted to exceptional scenarios by reasonableness testing can be invaluable, particularly since this type of verification can catch problems that you haven't even considered.

### RECONCILED

Reconciliation of data involves comparing the sources of your data to the targets, and verifying that the data matches. This is rarely as simple as just comparing rows of data from the source and the target systems. Data structures are usually different between source operational systems and reporting data structures, and the data is often cleaned, transformed and otherwise modified when it is moved from repository to repository. Reporting applications can add their own layers of transformation, including new calculations and filtering, that have to be accounted for in the reconciliation.

### The 3 R's



The primary measures of reconciliation are totals, counts and balances. You can calculate these easily from data storage by running aggregate queries against the data store. For example, if you are verifying a SQL Server repository, you might use a query like the following to get the count of orders and the totals for sales:

```
SELECT
    COUNT(DISTINCT OrderId) AS OrderCount,
    SUM(UnitPrice * ItemCount) AS TotalSalesAmount,
    SUM(ItemCount) AS TotalItemsSold
FROM
    Sales.OrderLine
```

A similar query can be executed against the data warehouse to retrieve the same numbers, and the results can be compared to validate that they match. Any differences should trigger additional research to discover the source of the mismatch. These queries may also need to be filtered for specific time ranges. It usually isn't necessary, nor wise for performance, to process your entire history for the totals each time you run your data verification. Rather, you will likely want to filter the results to data that has been modified or moved since you last ran the verification. Another common approach is run the verification for a specific time window — for example, records modified in the last 24 hours or the last 7 days.

## RELATED

When verifying the second 'R', that data is related, you are checking the relationships in the data. For example, you expect sales data to relate to a number of other pieces of information. The record of a sale could relate to a product, a customer, a date or a number of other entities. If those relationships are missing or invalid, the data may lose a significant amount of value.

One scenario where you can encounter the potential for unrelated data can be found if you work with non-relational data stores, like file based systems or NoSQL stores. These types of data stores don't enforce referential integrity, so there is nothing to guarantee that records are properly related.

If you work primarily with relational databases, you may not think you have much to worry about here. However, it's not uncommon for relational databases to be missing the referential integrity checks that they need to properly enforce relationships. In addition, many data warehouses intentionally don't enforce referential integrity for performance reasons.

It is not uncommon for relational databases to be missing the referential integrity checks needed to enforce relationships.

Even when you are working with a relational database where referential integrity is enforced, you may need to verify that your data is related. For example, in data warehouses, it is common to have an "Unknown" record in dimensions and

fact records that don't match existing dimensions may be assigned to these unknown records. These records are valid in that they pass referential integrity checks, and a certain volume of "unknown" relationships may be acceptable for your business scenarios. However, all organizations have limits to how many unknown relationships they can have in their data before it becomes unusable for decision making.

For example, when processing product sales, it may be acceptable to have 5% of sales map to an "unknown" customer location (geography), because it either wasn't provided from the customer or the provided information is invalid (the provided region and the postal code don't match so the geography can't be determined accurately). However, if the volume of "unknown" geography relationships for sales grows to 10%, that may be unacceptable to your business users for reporting accuracy. This tolerance will be different for other details — most business users would be pretty unhappy about 5% of their sales relating to "unknown" products.

To verify that data is related you will be testing counts and totals, but they will be subdivided by the relationships. In the case that you are checking a specific relationship, like an "unknown" record of a dimension, you may use a query that filters results to just that subset of data, like this:

```
SELECT
    SUM(ExtendedPrice) AS TotalUnknownSales
FROM
    DW.SalesOrder
WHERE
    CustomerId = -1
```

This allows your data verification tests to do specific checking for certain relationships and report any unexpected variances.



Another technique for validating relationships can be to look at specific groupings and the distribution across groups. This can help you to quickly identify outliers and unusual relationships.

```
SELECT
    Product.Category,
    SUM(ExtendedPrice) AS ExtendedPrice
FROM
    DW.SalesOrder
    INNER JOIN
    DW.Product ON SalesOrder.ProductId =
    Product.Id
GROUP BY
    Product.Category
```

The above query would let you easily compare categories to see if sales were abnormally high in any particular category, which would indicate that the categories were being related incorrectly. By using a second query that leveraged the results from the categorized sales query, you can test for anomalies in the distribution.

```
SELECT
    MAX(ExtendedPrice) - MIN(ExtendedPrice)
    AS ProductCategoryHighestToLowestDelta
FROM
    CategorizedSales
```

These values can also be useful in the next step, checking the reasonableness of the data. This is another place where being able to persist the values and results from the verification tests is beneficial.

## REASONABLE

The 3rd 'R' is that data is reasonable. This verification is a bit more subjective and will require you to work with your business users to determine what data scenarios are expected and normal, and which data scenarios are abnormal and require someone to be alerted. It's very helpful to know the trends for key metrics in your historical data and to have that available for discussions with business users. Generally, you can consider data reasonable if a business user would look at it and not find it surprising.

*Reasonable verification will require you to work with your business users to determine what data scenarios are normal or not.*

A simple example of reasonableness is looking at daily sales. Imagine your organization typically sells 10,000 widgets a month. After loading the current day's data into your data warehouse, you find that the data shows that you've exceeded 10,000 widgets for the month and it's only the 4th day of the month. While that's certainly exciting data (it's a record sales month!), you may not consider it reasonable that sales have skyrocketed that quickly. It's certainly possible that the data is legitimate, but you would want to be alerted and do some research to confirm those numbers, particularly before telling your CEO to buy a new house.





While that might be an extreme example, the same reasonableness criteria can be applied in other scenarios. If you pay sales people commission on your product sales, there's likely a specific range that you expect the commissions to fall into when you look at as a percentage of the sale. Doing a reasonableness verification on orders to ensure that commissions fall into the appropriate range can be very beneficial.

Simple reasonableness verification can be done by getting expected values for key metrics from the business, and querying to validate that they fall within an acceptable range. For example, if you wanted to verify that your daily sales were within 10% of an expected value of \$5,000, you could use the following query:

```
SELECT
    (TotalSalesAmount - 5000) AS Variance,
    CASE
        WHEN TotalSalesAmount
            BETWEEN (5000 * .9) AND (5000 * 1.1)
        THEN 'Valid'
        ELSE 'Invalid'
    END AS Result
FROM
    (SELECT
        SUM(UnitPrice * ItemCount) AS TotalSalesAmount
    FROM
        Sales.OrderLine) SalesToday
```

As currently implemented, this test is useful, but doesn't deal well with the normal variances that most organizations experience. Perhaps Fridays tend to be your busiest days or there are seasonal factors that lead to significant differences in the data for different time periods.

This type of test gets much more powerful when you combine it with historical results and expectations. If your data verification framework enables you to store test results, you can implement verification tests that compare the current results to historical data and trends. You can also leverage other sources of historical data to implement trend comparisons. With the proper setup, these tests can automatically adjust to changing business scenarios and trends.

One way to leverage historical information is to use standard deviation. In the example query batch below, the historical test results for total sales are used to determine a baseline standard deviation value. The difference between the current day's sales and the previous day's sales is then calculated and compared to the standard deviation. If it exceeds the calculated standard deviation, an alert would be raised.

```
DECLARE
    @CurrentDate DATE = GETDATE(),
    @Deviation MONEY,
    @Yesterday MONEY,
    @Today MONEY,
    @Delta MONEY

SELECT
    @Deviation = STDEV(TestValue)
FROM
    TestResult
WHERE
    TestCase = 'OrderAmount'
    AND ExecutionDate < @CurrentDate

SELECT
    @Today = TestValue
FROM
    TestResult
WHERE
    TestCase = 'OrderAmount'
    AND ExecutionDate = @CurrentDate

SELECT
    @Yesterday = TestValue
FROM
    TestResult
WHERE
    TestCase = 'OrderAmount'
    AND ExecutionDate = DATEADD(dd, -1,@CurrentDate)

SET @Delta = ABS(@Yesterday - @Today)

SELECT
    @Deviation AS Deviation,
    @Delta AS Delta,
    IIF(@Delta <= @Deviation, 'Valid', 'Invalid') AS Result
```

Note that this example assumes the use of a database table to store and retrieve the test results. Different implementations may take different forms. In the case of LegiTest, test history storage and retrieval is built into the test framework.

Reasonableness verification testing is a powerful technique, as it can be used to provide consistent sanity checks to your data. It can also be used to monitor for exceptional data situations, that might otherwise go unnoticed, as they don't always show up as errors or data reconciliation issues.



## Applying Data Verification

Data-centric systems often contain multiple data stores and applications. There can be numerous separate processes that move or transform the data. So how do you go about verifying that it's happening correctly? At what points should you apply the three R's of verification? There are two approaches that are commonly used, and frequently, these approaches are used in conjunction with each other for additional validation.

### STEP-BY-STEP

One approach is step-by-step validation. In this case, you look at the major components of your data infrastructure. Within the infrastructure, you identify each transition point for your data – where does it move from one system to another, or where is it transformed? You would then implement testing for each of these points.

Testing for reconciled data works very well in the step-by-step approach and it's highly recommended that you do reconciliation verification at each transition point. This helps you quickly identify spots where some of the data may have been missed or not transformed correctly. Checking for related and reasonable data is usually not as critical for each step, though you may want to implement these types of verification testing at specific transition points.

Step-by-step validation is particularly useful for identifying where in the overall data infrastructure a problem was introduced. However, it is targeted toward technical users who have a good understanding of the complete system. It may not work well for business users, who are often less concerned with where the problem occurred than the end result. Operations people do find the finer grained approach beneficial.

### Step-by-Step Validation

*Identify transition points for your data and test these points.*

### End-to-End Validation

*Compare the starting values in your source systems to the final values in reporting data structures, reports and dashboards.*

### END-TO-END

End-to-end verification focuses comparing the starting values in your source systems (one end) with the final values in your reporting data structures, reports and dashboards (the other end). The goal of this validation is to look at the macro level and major end points of the system, without considering the individual transition points. End points are usually the places where the data crosses the boundaries of your data-centric systems. End points can be originating, where data enters your data-centric system, or terminating, where data flows outside the control of your data-centric system. In some cases, an end point might function as both.

Some data-centric systems may have many end points. For example, systems that include point-of-sale applications may have hundreds or even thousands of end points. In this case, you have to identify a reasonable place to start the testing. If there is an easy way to replicate the tests over multiple end points, that may be an option. If not, you may consider working a little further up the flow of data to a point where the systems are combined into a common store.

Out of the three R's, end-to-end tests usually focus on reconciliation and reasonableness. You will typically want to reconcile the data at your end points, and verifying the reasonableness of the data

in your end points is critical to having confidence in the data. Relationships can be validated in these verifications too, but it's more commonly found in the step by step verification.

End-to-end verification works very well as a way of report system status for business users, who are generally more interested in the high level picture. Often, the business user really wants to know "Can I run my reports this morning and have confidence in the results?". An end-to-end test that reconciles

the source system totals to the values in the data warehouse can fill this need nicely.

End-to-end tests are also useful as quick smoke checks. Because end-to-end verification usually has less test cases and focuses on high level aggregates, they can be run more quickly than the step-by-step verifications. Many organizations benefit from having both – they run the end-to-end verifications to determine if there are any major issues, and then run the step-by-step validations to identify specific problem areas.

---

## Data Verification Example

For an idea of how you might apply this to a real life system, consider the following example. You are working on a data infrastructure that contains the following systems:

- An OLTP order processing system
- An OLTP customer relationship management (CRM) system
- A star-schema data warehouse (DW), which includes a staging area (Stage)
- An OLAP analysis repository
- A series of reports and dashboards that report on information from the data warehouse and OLAP repository

## TRANSITION POINTS

The transition points where data is moving or being transformed in this system are listed below, with details on example verifications for each.

	RECONCILED	RELATED	REASONABLE
Order Processing to Stage	●		●*
CRM to Stage	●		●*
Stage to DW	●	●	●
DW to OLAP	●	●	
DW to Reports	●		
OLAP to Reports	●		

Table 1 - Transition Point Verification

*\*Optional, but recommended for most scenarios*

### **Order Processing to Stage**

Very little transformation is done at this point. The primary goal is to move the data to the staging area for additional processing.

To validate accurate data movement, you would implement reconciliation that compared the key metrics from the order processing system to the staging area. This would include total orders, total sales amounts and current inventory levels. The queries for retrieving this information would be very similar, since there aren't significant changes to the data structure in this step.

Relationship and reasonableness verifications are optional at this step. In many scenarios, it would be beneficial to implement some reasonableness verification that looks at the count and total of the data transferred to determine if it matches the historical trends.

If the data movement is done incrementally, the verifications should be filtered so that they look at the current increment. For example, if the sales information for the previous day is being moved to the staging area, you would want to ensure that your queries were filtered to include only that day's information. That helps narrow the scope of any identified data mismatches.

### **CRM to Stage**

Much like Order Processing to Stage, this transition is focused on movement of data, not transformation. The validation logic is very similar, but you would focus on customer counts, total amounts for quotes, etc.

### **Stage to DW**

Stage to DW is primarily a transformation step. As part of this transition, the data is restructured to fit the data warehouse data, existing values are summarized and new values are created. Data that is determined to be invalid would also be excluded at this step.

You would implement all of the three R's at this transition point. Reconciliation verification would be used to ensure that totals and counts were not impacted by the transformations being applied. Relationship verification would be applied to validate that categorization of the data hadn't been impacted as facts were related to dimensions. Finally, reasonableness verification would compare the loaded values to ensure that they matched the historical trends and that there weren't unusual data scenarios encountered during the data transformation processing.

Validation of this step involves a good understanding of the business requirements. To implement the validation, you need to understand the data scenarios that may be encountered, as well as how they are expected to be handled. The validation would still be focused on the key metrics identified in the previous steps (total orders, sales amounts, customer counts), but the queries might be more complex, as the data has changed during the transition.

For example, you may have sales that are filtered out for missing or invalid data during processing. Validation would ensure that all totals from the original staging tables is compared to totals that include both the data inserted into the data warehouse, as well as any data that has been excluded as invalid.

DW to OLAP

This transition point is primarily focused on changing the storage of the data, rather than modifying it. Typically, the data is retrieved from a relational store and written to a multidimensional store with different physical storage properties. As part of the OLAP storage, additional calculations may be provided.

Since the data and business rules aren’t typically coming into play, the verification for this transition point is focused on reconciliation and relationships. The total amounts and counts should not change, and the data categorization should remain the same. Your verification tests should focus on ensuring that the totals match, that the associated facts and dimensions are still aligned, and that any new calculations implemented in the OLAP store return correct values.

DW to Report

This transition point enables the visualization of the data for end users. Normally, the data isn’t heavily transformed in this step, though it is often aggregated and summarized differently for reporting

purposes. Many times there are additional calculations implemented in the reporting tool.

As you implement verification for this transition point, your primary concern should be reconciling that the totals displayed on the reports match the totals from the data warehouse. Relationships are not typically modified in the reporting step, and the reasonableness of the data should be verified in a previous step.

If the reports are simple and straightforward, with direct pass through to the underlying storage, verification tests may not be necessary at this step. However, new calculations and transformations for display purposes are frequently implemented in the reporting layer. In these cases, reconciliation verification is important to ensure that the reporting layer isn’t presenting an invalid display of the source data.

OLAP to Report

The verification for this transition point matches the DW to Report transition point. In this case, the only difference is the data source for the reports.

END POINTS

End points are the places in your data-centric system where data is introduced from outside, or where the data flows out of your system. The following items are the end points for the example above.

	RECONCILED	RELATED	REASONABLE
Order Processing	●		●
CRM	●		●
DW	●		●
OLAP	●		
Reports	●		

Table 2 - End Point Verification



## Order Processing

Order processing is an originating end point. Data flows into this system from users or other systems interacting with the order processing system. End point verification at this point would consist of reconciliation steps, as well as reasonableness verification.

For reconciliation, your primary goal for this end point is to gather the totals, counts and balances that will be used for reconciliation with other end points. These values should be stored, so that you have a history of the values. The reasonableness verification should be based on these historical values. For example, you might capture the total order amounts and item counts for reconciliation. Your reasonableness tests would compare those totals to the historical trends, and if the variance was greater than 15%, an alert would be triggered.

## CRM

The CRM, like order processing, is an originating end point. You would apply the same verification steps to it. The metrics would be values for customer counts, outstanding quotes, geography breakdown of customers, etc.

## DW

The data warehouse is considered a terminating end point, as some users report directly from the warehouse, and other applications and systems use it as a source of data. For verification, your primary concerns are reconciliation and reasonableness.

The reconciliation verification should compare the amounts and counts with the originating end points. In this case, that would mean comparing the data warehouse total sales amount and item counts with the values captured from the order processing system, and the customer counts and quote amounts with the CRM system.



This is another point where you will want to persist the values for the recompilation verification, so that you can test the reasonableness of the data. Even though the data was tested for reasonableness at the source, and the data reconciles, the processes for loading the warehouse can be complicated, and the reasonableness check in this case verifies that the data matches trends, and that unusual scenarios (like a single salesperson making all of the sales for the day) are caught.

## OLAP

Since the OLAP system is another point where data is consumed by other systems and users, it is also considered a terminating end point. In this case, you would focus on reconciling to the CRM and order processing systems, using the same metrics as described above for the data warehouse.

## Reports

Reports and dashboards are often the terminating end point for users, though, thanks to the prevalence of Excel and the ease of exporting most reports to it, the data may still flow to other downstream uses. For this end point, reconciliation to source systems is the most important verification step. Particularly since this may be the only aspect of your data-centric system that end users interact with, it is important to make sure that the values on the report reconcile to the source systems and to the data warehouse or OLAP store, as appropriate.

## Infrastructure

As mentioned earlier, doing complete data verification can involve interacting with a large number of systems and running a large number of queries against them. Whether you choose to build your own or use a commercial package like [LegiTest](#), there are some key features you should look for:

### Automated execution of queries

As mentioned, there can be a large number of queries to execute, and if you want to test regularly (which is certainly recommended), manual execution will get very time-consuming, very quickly.

### Support comparison of results across disparate systems

Your test framework should handle the need to compare data across different systems that may treat the same nominal data type differently. For example, SQL Server, Oracle and JSON all use slightly different representations for date values. Your test framework should be able to account for this and still make accurate comparisons.

### Alerting

When tests fail, you will want someone to be notified. Make sure your test framework supports this.

### Flexible query result checking

Once you've run the query, you need a way to verify that it returned the expected data. This should be part of the automated execution, so that you get a pass/fail message for each test case. Bear in mind that some queries may return single values and some may return tables of information. Your test framework should handle either.

### History

Storing a history of test results is beneficial for reporting purposes, as well as seeing trends in test results. It's also vital to doing effective reasonableness verification, so this is a must have feature.

### Support multiple types of data stores

Very few organizations work with a single application interface or data storage technology. Your test framework should support all the tools you work with and be flexible to handle future ones. Ideally, it will support an extensibility model so that new data providers can be added easily.

### Dashboards/Reports

In addition to alerting, some users will want the ability to see current verification status of their data-centric systems, and to drill into details about test failures and issues. Ideally, this should be web-based or in a format that is easy to share with your users.

Having a proper framework in place means that you can focus on implementing the business logic for your tests, rather than worrying about how to run a query. While you can do some level of data-centric testing without a framework, it is strongly recommended.

*This whitepaper introduced data verification testing and discussed how it differs from testing data centric applications in development. The three R's of data verification (reconciled, related, and reasonable) were defined, as well as the approaches for implementing data verification tests.*

With the examples provided, you should be able to start implementing data verification testing in your organization.

This series of whitepapers has focused on data-centric testing, both from a development standpoint and for production data verification. You've seen the new concepts involved defined, as well as been presented with some of the benefits to be gained by adopting a data-centric view of testing in your organization. With examples of how to implement data centric testing at both the development and production level, you have an introduction to the approaches and techniques you can leverage.

Our goal for this series of whitepapers has been to help you to evaluate your approach, tools and techniques for dealing with the data-centric nature of organizations today. Data is a vital part of business, and is only becoming more important. We want to enable you to be better equipped to deal with not only today's challenges, but to make sure you are ready to face the challenges of tomorrow.

We believe that it's necessary to go beyond saying that your data is good – you should be able to prove it. If you can show, through repeatable tests and verification, that your data is reconciled, related and reasonable, it gives you confidence in your work and it gives your users confidence to make the decisions they need to make. And at the end of the day, everyone working on your data-centric systems will be happier and more productive. We want to help you get there.

---

**PRAGMATIC WORKS CAN HELP YOU BUILD A MORE RELIABLE DATA-CENTRIC ORGANIZATION. FOR MORE INFORMATION ON DATA-CENTRIC TESTING AND TO REQUEST A DEMO OF OUR PRODUCT LEGITEST, PLEASE VISIT [PRAGMATICWORKS.COM](https://pragmaticworks.com).**