Chapter 14

Testing Builds

14.1 STRUCTURE OF A DATA BUILD

The structure of a build is illustrated in Figure 50. A detailed treatment of data builds can be found in Chapter 12, but here is a brief summary. A raw data layer contains unaltered data, exactly as it was received from the customer. There are both system extract data sources and ad-hoc data sources that have been created as part of ongoing project work. Version-controlled code files then create one or more intermediate datasets as data is cleaned, enriched, and business rules are applied. The final layer is the interface layer. This contains the convenience datasets that are exposed to the analytics team to promote consistency in their work products.

14.2 AN ILLUSTRATIVE EXAMPLE

It will be helpful throughout this chapter to consider an illustrative data build. Figure 51 shows the build process from raw data through to build datasets. The example project involves analyzing the user permissions in a collection of IT systems so that we can determine duplicate users, missing user IDs, decommissioned users, and HR leavers who have not had their system accounts deactivated. This is a typical problem encountered in the area of Identity Access Management (IAM). To keep things simple and illustrative, we will imagine there are three systems involved in this project and we will name them "System 1," "System 2," and "System 3." There is an extract from each system that lists the user ID, user name, and last login date. We will also consider an HR system extract that conveniently lists all leavers by their company-wide user ID and the date they left the company.

Throughout this project, our wider team who specialize in security audit will need our support in providing details on specific systems and users, as well as views on the rate of closure of leaver accounts and the health of the systems under investigation. Once proper data checks have been done and data has been loaded, the next step is to create a data build where this evolving knowledge can be maintained. The details of the build layers are described in the following sections.

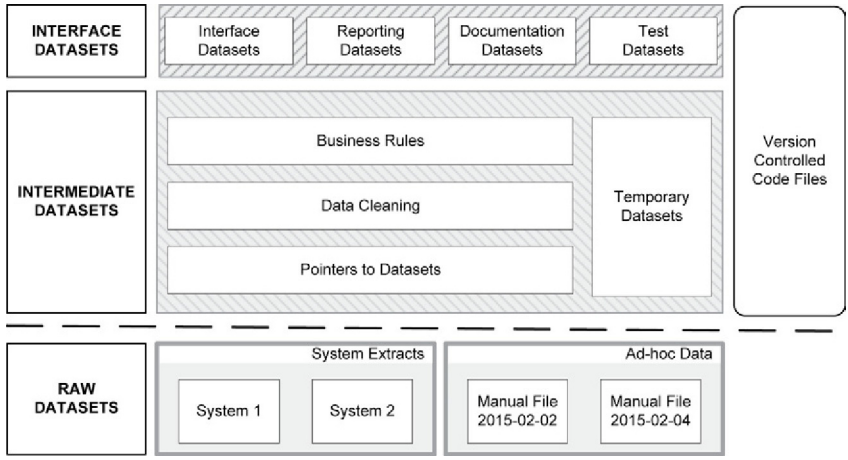

FIGURE 50 Structure of a build

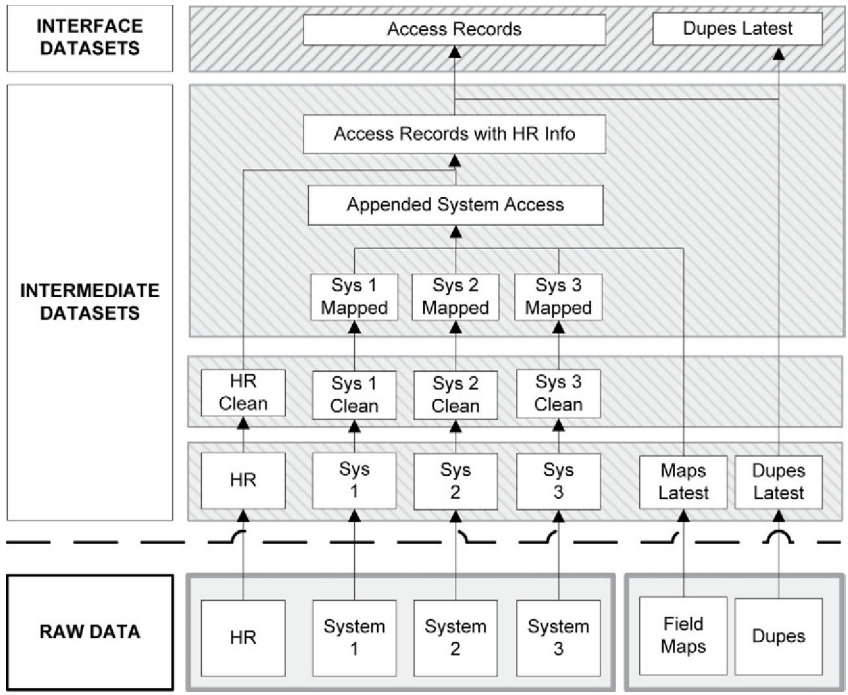

FIGURE 51 Illustrative example of a build

14.2.1 Layer 0: Raw

This is the storage area for all raw data received. It contains both the access permission extracts from three systems and the HR system extract. There are also ad-hoc extracts of duplicate reviews and system field mappings. All of these data extracts could be refreshed during the course of the project. In a live environment, IT staff will continue to add and remove users to the systems, and people will join and leave the company. This is typical of a Guerrilla Analytics project.

14.2.2 Layer 1: Pointers

As discussed in the earlier builds chapter, this is a simple pointer layer to decouple raw data from intermediate build datasets and ensure that the latest version of any data extract is always being used.

14.2.3 Layer 2: Cleaning

This layer takes each raw dataset and applies cleaning rules. The typical cleaning rules are as follows.

- Convert every user name to upper case.
- Break out names into first name and last name fields. Some system extracts have done that for us. Others are in first name, last name format, and still others are in last name, first name format.
- In one system, some User ID fields contain a blank or just some dud characters. All of these fields are identified and set to NULL.
- A coherence test that joins the HR data onto each system file shows that about 10% of the system data does not have a corresponding HR record.

Hopefully you have already spotted how these data issues were identified? These issues would have arisen from the data correctness and completeness tests discussed in the previous chapter 13.

14.2.4 Layer 3: Rules

This layer applies business rules to the cleaned data.

- Each system extract's fields are mapped into a common format using the agreed field mapping that has been worked out with the customer.
- The mapped extracts are appended into a single dataset that contains a source system identifier, a user ID, a user name, and a last login date.
- HR details are added to this dataset.

14.2.5 Layer 4: Interface Datasets

Two datasets are exposed in the interface.

- A single dataset of all system access records across all systems tagged with HR information and recognized duplicates.

- The latest duplicate markup file is exposed in the interface so it can be included with work products and used in reporting. This helps customers understand where duplicates are coming from, and work with the Guerrilla Analytics team to sign off or correct any duplicates they have identified.

The next sections will describe the types of tests you can implement to find defects in this simple data build.

14.3 TYPES OF BUILD TESTS

Testing is a huge field in its own right. Reading the software engineering literature you will find mention of many types of tests and many approaches to testing. There is also ample technology support for developing, maintaining, and executing tests. Many test types and much of the literature are focused on traditional software development and are only partially suitable for Guerrilla Analytics work. Before going any further, it helps to narrow down the types of tests that need to be considered when testing an analytics build in the constraints of a Guerrilla Analytics environment.

Whittaker et al. (2012) provide a good and simple classification of test types used at Google. Their approach emphasizes scope of testing rather than the form of testing. They define the following three types of test.

- **Small tests.** These tests cover a single unit of code in a completely mocked up environment. In traditional testing circles these tests might be called unit tests.
- **Medium tests.** These tests cover multiple and interacting units of code in a faked or real environment. Again, in traditional testing circles, these tests might be called component tests (several code files that together form a service) or integration tests.
- **Large tests.** These tests cover any number of units of code in the actual production environment with real resources. These tests are about performance at scale and on real production data.

Whittaker et al. (2012) are of course writing from the perspective of software development. These three types of test go a long way toward increasing confidence in an analytics build also. The next sections will describe these tests from the perspective of testing a Guerrilla analytics data build.

14.3.1 Small Tests

Small tests cover a single unit of code (generally a single program code file). In traditional software engineering and testing, these tests would have been called "unit tests" or perhaps "component tests." But what exactly is a unit or component in a build?

A program unit in a data build is a clearly identified data step or set of data steps that perform a significant data manipulation, and result in a significant

intermediate or interface dataset. Here is a small test scenario from the illustrative build example of Figure 51.

The team is concerned that some of their Layer 2 data cleaning may be a little over zealous. They have used a function that strips all dud characters from a user name field in each of the system permission extracts. Characters such as -, $, %, # are removed. The first version of this cleaning code included apostrophes in its list of dud characters as these had been seen to occur in unusual places in users' names. However, the team later notice that some Irish and Scottish user names are badly affected by this cleaning. A valid name like "O' Malley" is being cleaned to "O Malley."

Having created a fix in a new version of the cleaning code, the team now wants to write a test to make sure this problem does not arise the next time the cleaning rule is adjusted. This test will ensure the cleaning code is not making this apostrophe replacement when apostrophes come after a standalone letter "O" between a first and a last name. Here is a high-level small test script.

- Run through data records in the raw user dataset from "System 1" and using a regular expression count the occurrences of "O" followed by an optional one or two spaces followed by an apostrophe. This is the test's expected number of names with the O apostrophe characteristic, and we expect that these should not be cleaned out of the raw data.
- Run through the cleaned user data from "System 1" and using a regular expression to perform the same count. This is the test's actual number.
- Compare the expected number and actual number.
- If the expected and actual differ, write a test result file that lists test details such as run time, tested dataset, expected value, actual value, and a result of "FAIL."

What the team have done here is wrap some expectations around a single code file to check that it does not "over clean" the data feature of user name. As other problematic cleaning features are discovered, the team will add further tests to capture those additional functionalities. Here are a few suggestions.

- Some fields may have trailing spaces because of file format differences. These, while often invisible to the developer and user, can cause problems when matching field contents. The field "Joe Bloggs " is not the same as "Joe Bloggs." A test could check that all trailing spaces and multiple adjacent spaces within fields have been removed.
- Often blank fields need to be converted to NULL. A test could check that no blank fields remain after cleaning.
- Date fields should be within a sensible range. A test could check there are no dates before some business relevant point in time.

There are many possible tests depending on what is expected of the cleaned datasets. The main point to take away here is that small tests wrap around a single code file and modular functionality.

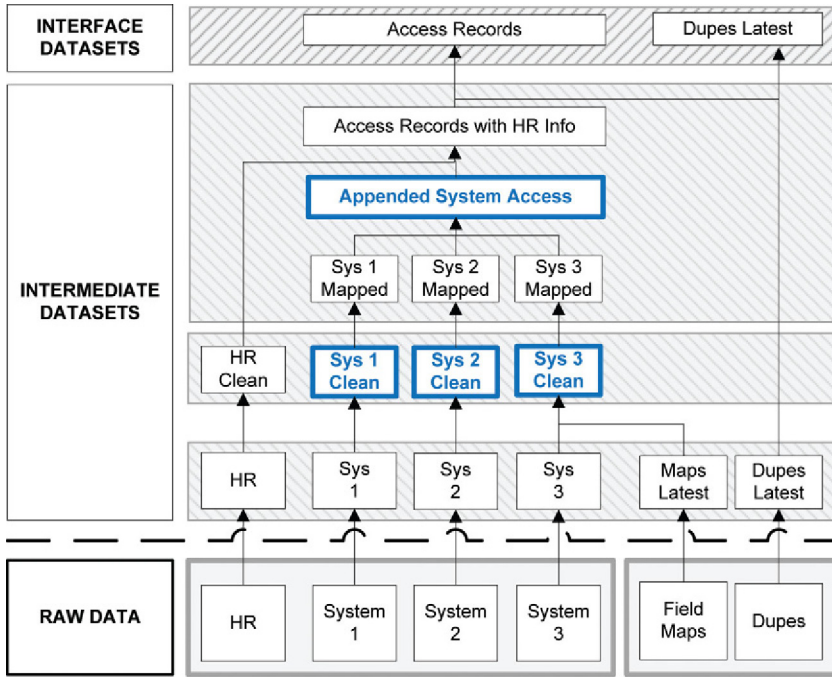

FIGURE 52 An illustration of a medium test

14.3.2 Medium Tests

Medium tests cover several code units that work together to produce an intermediate or interface dataset. Let us look at a medium test from the illustrative build example. The build data flow is reproduced in Figure 52 with the datasets from the relevant data segment highlighted.

Several datasets interact to produce the "Appended System Access" intermediate dataset. The relevant code files are as follows.

- "Sys 1 Clean," "Sys 2 Clean," and "Sys 3 Clean" are passed through the mapping dataset to produce equivalent mapped datasets where fields have been renamed to a common agreed set of field names.
- Once the system extracts have a common format, "Appended System Access" unions together the mapped datasets to produce the dataset "Appended System Access."

One key feature requiring a medium test is whether fields have been correctly renamed given the definitions of field mappings. A priority field to test would be the user name field. A medium test (covering the interacting code files above) could look like the following.

- Find the distinct list of IDs and user names in the Appended System Access dataset.

- Find the distinct list of user IDs and user names in the three clean system extract datasets.
- Check that all ID and username combinations are the same in the "Appended System Access" dataset as they are in the individual cleaned system access datasets.

This medium test is very similar in principle to the small test. An expected value of user ID and name is calculated from the input system access files. An actual value after mapping is calculated from the "Appended System Access" dataset. Both are compared to test that the mapping application did not break the combinations of user IDs and user names.

14.3.3 Large Tests

Large tests are run against significant marker points in the build datasets, ignoring all the intermediate datasets and code units between those marker points. They are really a type of end-to-end test.

Returning to the illustrative build one last time, a large test could look like the following, illustrated in Figure 53. The aim is to test that no users are dropped at any point in the build's data flow.

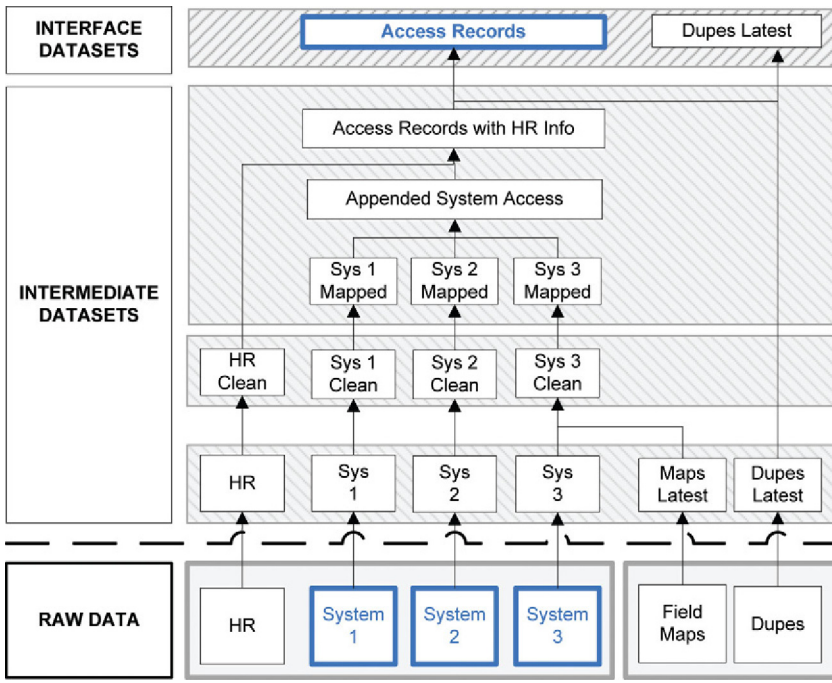

FIGURE 53 An illustration of a large test

- Append together all user IDs from the raw data files. This is the expected list of user IDs in the population.
- Compare this list to the user IDs in a build interface dataset. This is the actual list of user IDs.

As you can see from this example, there is no reference to or concern with any intermediate datasets and data steps on the journey from raw to interface. Also, this test is run at scale across the entire data population as opposed to a data segment or intermediate dataset.

14.4 TEST CODE DEVELOPMENT

Implementing a test involves writing a test code file. There are several tips and tricks to make this code easy to understand and share amongst the Guerrilla Analytics team.

14.4.1 Practice Tip 77: Use a Common Layout in Test Code Files

14.4.1.1 Guerrilla Analytics Environment

Test code files are no different from analytics code files. They can become complex and they require maintenance. Without a proper strategy for developing test code, the Guerrilla Analytics team can end up overwhelmed by a large number of poorly understood test scripts that are more trouble than the code they are supposed to be testing.

14.4.1.2 Guerrilla Analytics Approach

We have seen how all tests follow a similar pattern.

- **Setup.** Gather an expected dataset or calculate an expected value from the data.
- **Test.** Compare the expected value to the actual value in the data.
- **Store results.** Write a test result dataset.
- **Report failures.** If the test fails, write a test fail details dataset to help investigate and understand the failure.

This pattern is needed regardless of the details of the test, which can vary significantly.

More established software engineering test frameworks help a developer by having clear methods for setup, teardown, messaging, assertion, and failure code (Tahchiev et al., 2010). Guerrilla Analytics borrows from these fields. Within a test code file, the layout of setup, test, and result should be clearly visible from the test code layout and comments.

14.4.1.3 Advantages

The advantages of a clear test code layout in a Guerrilla Analytics environment are the following.

- Code is easier to understand, review, and share between team members. This is important to avoid the potential chaos of a dynamic Guerrilla Analytics project.
- Common code functionality for setting up and tearing down tests can be developed and used across the test suite. This saves the effort of reinventing the wheel for every test and so gives precious time back to the Guerrilla Analysts.
- Maintenance of test code is easier when all test code files are similar in form. This helps when the provenance of test results is challenged and when tests begin to fail.

14.4.2 Practice Tip 78: Create a Common Test Results Dataset Structure

14.4.2.1 Guerrilla Analytics Environment

Similar to when testing data, build tests will also write a test results dataset to record the outcome of every test execution. If this results dataset is different for every test, then it becomes time-consuming and error prone to inspect results and determine which tests are failing. In a Guerrilla Analytics project where code should be executed and tested often, this can result in delays in the publication of new builds that the team need for their work products.

14.4.2.2 Guerrilla Analytics Approach

Test result datasets should be developed with a structure that is common across all tests. Figure 54 shows a suggested test result structure. This provides information such as:

- **Test unique identifier (UID):** This is the UID of the test.
- **Description:** This is a simple reporting level description of what the test does.
- **Test type:** This simply states whether the test is small, medium, or large.

TEST ID	DESCRIPTION	TYPE	DATASETS	TIME	RESULT
50	Appended system access mapping	MEDIUM	APPENDED_SYSTEM_ACCESS, SYS X CLEAN...	2014-04-17 10:03	FAIL
50	Appended system access mapping	MEDIUM	APPENDED_SYSTEM_ACCESS, SYS X CLEAN...	2014-04-16 11:49	PASS
34	Dud character removal	SMALL	SYS_1_CLEAN	2014-04-16 11:47	PASS
49	Deduplication of names	MEDIUM	ACCESS_RECORDS, DUPES LATEST	2014-04-16 11:46	PASS
20	Trailing space removal	SMALL	SYS_3_CLEAN, HR...	2014-04-16 11:46	PASS

FIGURE 54 A test result dataset structure

- **Datasets:** This is a list of the dataset or datasets being tested.
- **Time:** This records when the particular test was run and the result was recorded.
- **Result:** This records whether a particular test run passed or failed.

While this is a suboptimal data model, remember that these datasets will be inspected by users to understand test failures and so there is a compromise between efficiency of data model and user friendliness.

14.4.2.3 Advantages

If the test results dataset has a common structure, then it becomes very easy to write further code across the test suite to provide functionality such as the following.

- Collect and report on the latest test result from every test result dataset.
- Create summary statistics across all tests or some subset of tests to show coverage, last execution time, total run time, etc.

14.4.3 Practice Tip 79: Develop Modular Build Code to Help Testing

14.4.3.1 Guerrilla Analytics Environment

The temptation in a pressurized Guerrilla Analytics environment is often to write code that performs several key data manipulations at once because this is perceived to be clever or more efficient. But small and medium tests cannot be wrapped around this functionality if the build code and associated data steps are not modular. A balance needs to be struck between having fewer intermediate data steps and having lots of data steps that can be isolated and tested.

A good indicator that the build code is not sufficiently modular is that you have to go to significant effort in a test script to prepare datasets for the test. In extreme cases, parts of the build end up being rewritten in the test code to facilitate testing. Imagine how more difficult it would be to write a medium test around the field mapping if the example build's data flow looked like that of Figure 55. There is no easy way to pull apart each system's mapping or inspect the appended dataset before HR information is added to it.

14.4.3.2 Guerrilla Analytics Approach

At a minimum, identify critical marker datasets that need to be written into intermediate datasets from the build process. Implement build code with testing in mind.

14.4.3.3 Advantages

Small and medium tests can easily be wrapped around critical data steps and intermediate datasets to test important parts of the build data flow. Testing code is kept to a minimum and tests can execute quickly.

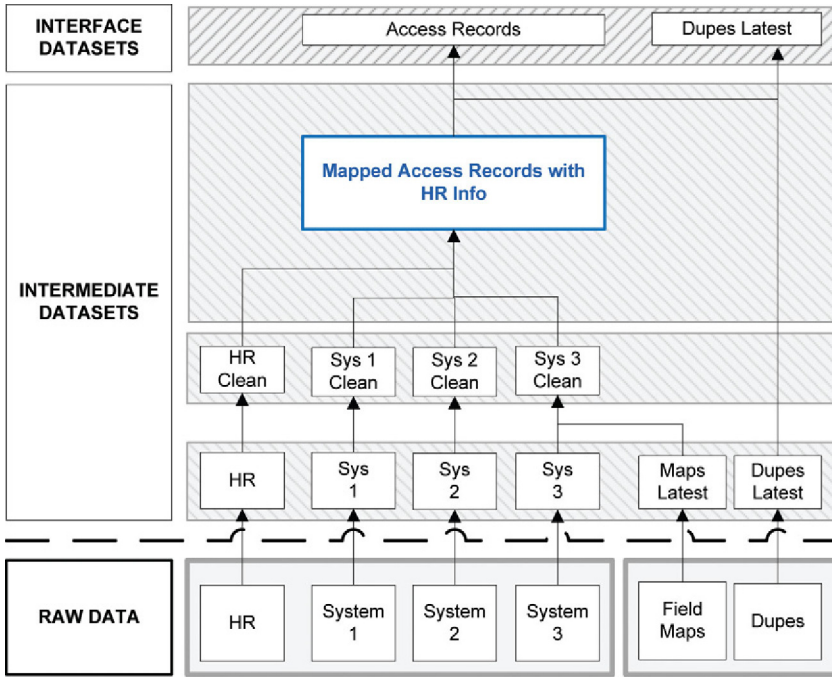

FIGURE 55 Modular build code makes testing easier

14.5 ORGANIZING BUILD TEST CODE

Tests datasets have to be stored somewhere. The test code also has to be stored somewhere. In build testing, only small tests map one-to-one with build code files and build datasets. More complicated medium and large tests may cover several code files and build datasets. This raises questions of how to clearly associate test code with both the build code and the build datasets under test. The following practice tips show how to best organize test code for this purpose.

14.5.1 Practice Tip 80: Use One Test Code File Per Test

14.5.1.1 Guerrilla Analytics Environment

The Guerrilla Analytics environment will require several build projects, each with their own build versions. This will result in many build code files and build datasets. Over time, a suite of tests will build up in the code base. This leads to confusion over which test code files perform which tests. The Guerrilla Analytics team do not want to waste time trying to coordinate and understand their own tests and code.

14.5.1.2 Guerrilla Analytics Approach

When discussing work product development and build development in previous chapters, it was recommended to have one code file per data output. This one-to-one mapping between code files and datasets helps to easily identify the code that created datasets in the DME, and encourages modular development that is easier to review, test, and report on. Testing is similar. It is best to have one test file per test.

14.5.1.3 Advantages

The advantages of one test file per test are as follows.

- **Debugging.** When a test fails, you can quickly locate the test file that reported the failure.
- **Maintenance of tests.** Test files are modular and easy to understand because they are small and do only one thing. A secondary benefit here is that if a single test script is becoming overly large and unwieldy, it is probably an indicator that the build is not sufficiently modular.

14.5.2 Practice Tip 81: Make Test Code Easily Identifiable with a Test Register

14.5.2.1 Guerrilla Analytics Environment

One of the most important advantages of writing a test is to help guide development. That is, a build developer should be able to write some build code and then execute the associated test code to gain confidence that his changes to the build have not broken the build. Testing is different to general analytics code where one code file creates one dataset. In testing, a particular code unit can be covered by multiple small and medium tests, each of which has their own test code file. This makes it difficult for a developer to know which tests cover their code, and quickly locate those test scripts to execute them.

14.5.2.2 Guerrilla Analytics Approach

The easiest way to solve this problem is to have a test register dataset that maps test UIDs to build code units and datasets. The creation and maintenance of this lookup dataset is owned by the test script writer. This means that when a contribution is made to the build code, the contributor should also be writing any necessary test scripts and registering them in the test register dataset.

Figure 56 shows what a test register dataset could look like for this chapter's illustrative build. Four tests have been registered in this example. Test 50 covers two datasets as it is a medium test. Three small tests cover dud character removal in the three system extract datasets.

Test registration should be done as part of the test's setup code so that it sits right beside the test script under version control. In practical terms this could be achieved in an RDBMS with UPDATE statements into the test register dataset.

TEST ID	DESCRIPTION	TYPE	DATASETS
50	Appended system access mapping	MEDIUM	APPENDED_SYSTEM_ACCESS
50	Appended system access mapping	MEDIUM	SYS_1_CLEAN
34	Dud character removal	SMALL	SYS_3_CLEAN
35	Dud character removal	SMALL	SYS_2_CLEAN
36	Dud character removal	SMALL	SYS_1_CLEAN

FIGURE 56 Test register dataset

14.5.2.3 Advantages

With a test register in place, the challenges of a fast-moving project with associated changes in the build are addressed. Specifically, build developers can easily identify the tests that cover their build code. This is achieved with a minimum of documentation where the only obligation is on test writers to register their test script.

14.5.3 Practice Tip 82: Make Test Code Part of the Build Code Base

14.5.3.1 Guerrilla Analytics Environment

Tests need to evolve with the build. As new features are added and some features are repaired or changed, so new tests need to be added or existing tests need to be changed. This means that test code only makes sense in the context of the version of the build code it is testing.

14.5.3.2 Guerrilla Analytics Approach

Test code should be under the same revision control system as the build code and build versions should contain the latest test code. Test code is as much a part of the code base as the build code.

14.5.3.3 Advantages

Taking this approach, a repaired bug in a particular build version can be verified to have been closed and covered with an associated test. A report can be run to list all tests and their descriptions from the test register along with their latest test results. This type of report can then accompany build delivery.

14.6 ORGANIZING TEST DATA

Test code uses and generates test data. Three types of data are associated with tests.

- **Test input datasets:** Some tests require a mocked up input dataset that contains data characteristics being tested. For example, if you were writing a

small test around deduplication functionality, you would create a test dataset that deliberately contained duplicates and then pass this mocked up data to the code under test.

- **Intermediate datasets:** When a test executes, it may need to create some intermediate datasets of its own. This helps simplify and debug test code.
- **Test result datasets:** Ultimately, the result of the test and any details of a failed test need to be archived somewhere for future reference and for investigation.

The following tips show how best to organize these types of test datasets.

14.6.1 Practice Tip 83: Retain Intermediate Test Datasets

14.6.1.1 Guerrilla Analytics Environment

We already discussed how tests will often need to prepare supporting test datasets in a setup phase. These supporting datasets are essential to understanding why a test failed. If tests do not store these datasets then it can be time-consuming to investigate why a test failed.

14.6.1.2 Guerrilla Analytics Approach

A Guerrilla Analytics team should store intermediate datasets in an appropriate DME location when a test fails.

14.6.1.3 Advantages

- The team can quickly identify a failed test and locate its intermediate datasets.
- Time is not wasted having to rerun a test step by step to inspect what it does and understand why it failed.

14.6.2 Practice Tip 84: Store Test Results and Test Failure Details

14.6.2.1 Guerrilla Analytics Environment

Tests will be executed many times and every execution will produce a test result. Sometimes that result will be a fail and sometimes it will be a pass. If a log of test results is not maintained then it becomes difficult for the team to determine when a test began to fail and therefore determine the changes that might have caused this failure. If the data that caused the failure is not stored then it is difficult to investigate the cause of the failure.

14.6.2.2 Guerrilla Analytics Approach

Test results should be stored in a DME location. Again, in making results easy to find, it is best to store results in a namespace that contains the test UID. Every execution of a test writes its test results into its associated result dataset. Thus the result dataset is effectively a log of all executions of that particular test.

Different tests will fail in different ways. For example, a large test comparison of the user IDs in the raw data and the build interface might fail when one

or more user IDs are lost in the build. A test failure would need to list these missing user IDs. When further detail on a test failure is useful, then this should be should be stored in a failure details dataset. While the results dataset has the same format across all tests, the details dataset can be allowed to differ depending on the specifics of the test that is failing.

14.6.2.3 Advantages

The Guerrilla Analytics team can quickly determine two things that greatly help the investigation of a failed test.

- **Time of failure.** By inspecting the results dataset, the team can see when a test began to fail.
- **Cause of failure.** By looking through the specific data records that caused a failure, the team can more easily determine why the test reported a fail and adjust their code or test accordingly.

14.6.3 Practice Tip 85: Establish Utility Data Structures in Setup

14.6.3.1 Guerrilla Analytics Environment

Some tests will rely on setup datasets before they can perform their test checks. For example, a large test checking that all user IDs are in a build interface dataset will have to first generate a list of all user IDs from the raw datasets feeding into the build. If every test creates its own setup datasets, space is wasted and tests take longer than necessary to execute.

14.6.3.2 Guerrilla Analytics Approach

Guerrilla Analytics takes the approach of identifying common utility datasets that serve several tests and building these datasets at the start of a test suite. Tests can then use these utility datasets instead of building them from scratch.

In the user ID example, a list of all raw user IDs is probably a useful dataset that could be availed of by other tests in the test suite.

Figure 57 shows a test suite code folder that contains setup scripts and the corresponding utility datasets for the illustrative test example. The need for two utility datasets has been identified. The User_IDs dataset is a list of all raw user IDs input to the build. The HR_IDs dataset is a list of all raw HR system IDs. There are several points to note about the organization of the code and datasets.

- There is now familiar one-to-one mapping between a utility dataset and the test script that creates the dataset.
- Setup scripts are stored at the top of the test script folder. They are the first scripts to execute because subsequent test scripts will use their output datasets.
- Subsequent test scripts then have their own folder that is named after their test UID.

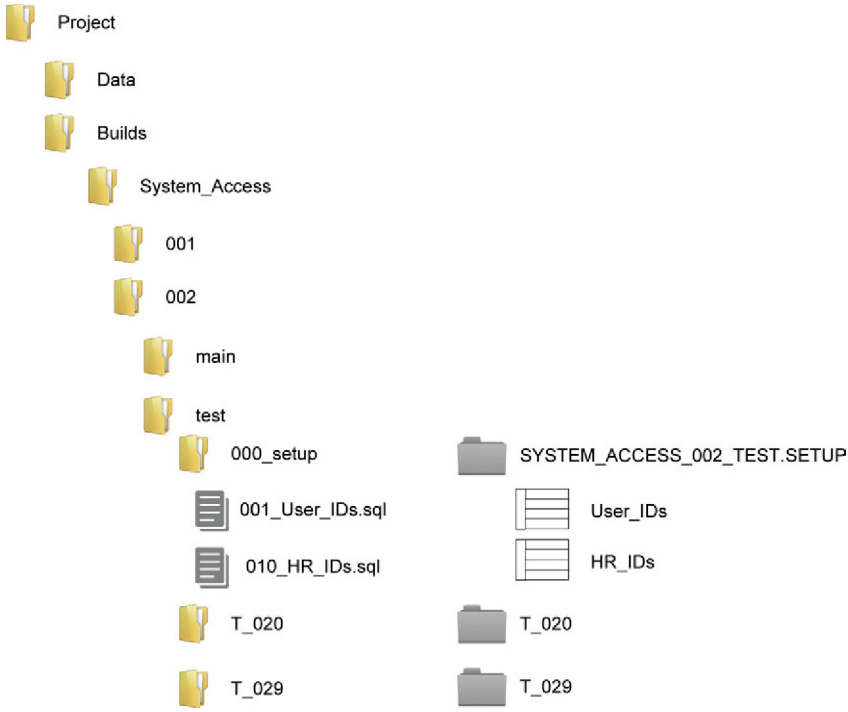

FIGURE 57 Setup datasets for test suite

14.6.3.3 Advantages

- Having utility datasets available avoids having to repeatedly generate them in the many subsequent tests that rely on them. This reduces test run time.
- Test scripts are simplified and this reduces the chance of bugs appearing in test scripts.

14.6.4 Practice Tip 86: Automate Build Test Execution and Reporting

14.6.4.1 Guerrilla Analytics Environment

As the test suite grows, it becomes increasingly cumbersome to manually execute all the test scripts and check for passes or fails. In a Guerrilla Analytics project, the build will change frequently and so tests should be executed and reported on. If test execution and reporting is a manual ad-hoc process, then it is time-consuming and expensive to do and hold up the release of new builds to the broader team. Furthermore, a key advantage of testing – to guide and accelerate development – is lost.

14.6.4.2 Guerrilla Analytics Approach

The obvious solution is to automate executing tests and checking results for failures. Test execution covers:

- Tearing down and rebuilding of the test suite setup datasets.
- Execution of each test script.
- Reporting of test scripts that have failed.

Automation includes not only the capture of test results, but also the first-level reporting of those results. As far as possible, tests should do the comparison of expected to actual, so an analyst need only get involved when a test starts to fail.

The build code will have a build code file that executes all the build code and produces fresh build datasets. This build file should also have the option to execute the build's test suite. In this way, testing is integral to the build and is run every time a new build is created.

14.6.4.3 Advantages

With test execution and reporting automated, several advantages are realized.

- Testing can be an integral part of the build process, improving quality, and identifying build defects early.
- The test suite can grow to include many test scripts without a significant operational cost for the team.

War Story 15: Better Late, Then Never

William was a test manager whose responsibility was to test a data build in a Guerrilla Analytics team. Unfortunately for William, the build did not have a very modular design. The build team struggled to implement significant changes to their code because a single feature change involved many changes throughout the build data flow. Because testing had not been tightly integrated into the build, William was always waiting for the build team to declare they had finished their work before William could begin his testing.

Testing was effectively put on hold for several days while the build was developed and executed. Testing always ended up lagging behind the build and pressure came on the build team to release results before they had been fully tested. When testing was eventually executed, scripts had to first unravel the tightly coupled build structures before testing them. In effective, much of the build code was being refactored in the test scripts. The whole project began to hinge on when the build was ready.

William decided to take a Guerrilla Analytics approach to testing. He met with the build manager, explaining where his test code required significant refactoring and how some minor changes to the build structure would benefit both teams. Build code would be more modular and testing would have to do less reverse engineering of the build data structures.

Once this build refactoring was complete, William then pushed for test code to be held within the main build code base. This meant that tests executed at the same time as the build code executed. There was no more waiting for the build to complete or worse still, results being issued before testing had completed. The build team benefited because they could publish a build accompanied by a verified set of passed tests.

14.7 WRAP UP

In this chapter, you have learned about testing data builds. In particular, this chapter has covered the following.

- A reminder of the structure of a data build with an illustrative example of building and deduplicating an analysis of system access permissions.
- The three categories of build tests.
  - Small tests wrap around a single code file or dataset.
  - Medium tests wrap around a data segment or several code files from the build.
  - Large tests wrap around the entire build and test its end-to-end processing.
- Test code development described tips for writing test code that is clear, traceable, and maintainable.
  - Test code should have a clear uniform layout for all tests.
  - Tests should store their results in a common results dataset structure.
  - Build code should be sufficiently modular to facilitate wrapping tests around priority segments of the build.
- Organizing test code described tips for storing test code so that it is easily identifiable.
  - There should be one test code file for each test.
  - Each test should have its own UID.
  - Each test should register itself in a central test register.
  - Test code should reside in the build codebase and be executed as part of the build.
- Organizing test data described tips for storing input datasets, intermediate datasets, and results datasets.
  - Input test datasets should be stored under their associated test UID.
  - Intermediate datasets from a test execution should be retained so that test failures can be investigated.
  - Test results should be stored in a common result dataset structure.
  - Test failure details should be stored separately.
- Executing tests covered operational tips for testing builds.
  - Tests should be executed often.
  - Where possible, common test datasets should be identified and created as utility datasets that other tests scripts can use.
  - Test execution and reporting should be automated so it is easy to run tests often and test results need only be inspected when there is a test failure.