



"turning data into dollars"

Tom's Ten Data Tips – May 2010

### Data Warehouse Testing

Data Warehouse (DWH) testing is a new and relatively immature discipline. There remains considerable disagreement whether DWH testing is any different from "normal" software testing, or whether there is merit in a DWH-specific testing approach.

The nature of business intelligence (BI) is that source systems (usually) are consolidated in some central "hub" (the DWH), and from there passed on to end-users via front end applications. All three touch points, source systems, DWH, and front end applications, potentially contribute to errors. Where these *truly* reside is generally not knowable to end users. This reality, that is pretty unique for BI and data warehousing, needs to drive testing plans.

#### 1. End-Users Don't Care Where Problems Originate (And Neither Should They)

There's a significant difference between data warehouse (DWH) testing, and testing of other applications. Because you migrate data from source systems to the DWH, and then extract information from the DWH through front-end applications, there is a compound effect. You have a source to DWH, and DWH to front-end mapping. Because both stages can cause errors, there's a genuine risk of confounding. And of course the source systems themselves can contain errors.

End-users can't see, and nor do (or should) they care which of these layers caused the error. All *they* see is that some number in a report is wrong. Or that there remain duplicate addresses in a direct mail selection, etc. To disentangle these effects, a purposely chosen mix of whitebox and blackbox testing is called for (see also tip# 2).

#### 2. DWH Testing Requires Both Whitebox And Blackbox Testing

Whitebox testing is when the tester is "allowed" to look at the program code to resolve errors. We talk about blackbox testing when testers are "naïve" and have *no* access to program code while performing tests. Both methods have pros and cons as has been well established in the testing literature.

Due to confounding between the extract, transform, load (ETL) layer and the front-end layer (see tip# 1), together with a combinatorial explosion of possible test scenarios, you will *always* need both whitebox and blackbox testing to get an economic coverage of relevant test conditions. Even millions of test sets would be insufficient for adequate test coverage, so merely blackbox testing is prohibitively time consuming.

### 3. Waterfall (or "V") Models For Testing Result In Time vs. Reliability Tradeoffs

"Traditional" software development (and DWHs are no different) has suggested to follow fixed steps: requirements-design-coding-unit test-integration test-system test-acceptance test. "V" testing means you follow these same steps where the second half are devoted to testing (hence the "V"). Now if you get it "right" the first time around, all is well. But that's not always how it goes, in particular in DWH projects where the requirements tend to have considerable ambiguity (see also tip# 4 from a previous newsletter on Agile Software Development).

The challenge when your DWH project doesn't go exactly as planned is that you need to commit resources early, long before implementation with its inevitable surprises. By the time you get to later testing stages, it is often too late to make big changes. The project then is faced with an ugly choice: either to delay the release, or accept a mediocre product (a time versus reliability tradeoff). Such realizations have driven support for iterative approaches. See also a previous newsletter on Agile Software Development. It is not about persisting along the *path* you have chosen, but rather pursuing your initial *goal*. It's futile attempting to plan your DWH development in great detail, but you *should* keep an eye out on your business case at all times.

### 4. ETL Testing Benefits From An Incremental Approach

During DWH development, the ETL phase tends to take disproportionately long. Gartner estimates it typically consumes 60%-90% of project resources. This is also where the largest testing risks lie. So naturally, this is where you will focus a lot of your testing effort. What makes ETL extra complex is that source to target mappings can (often) contain a flow of multiple, sequential operations. Any attempt to test such complex mappings in one "big bang" is bound to fail. Because of sequential processes, there are almost always "blocking faults" that preclude comprehensive testing (see also tip# 5).

The way to circumvent such risk, and avoid uncontrollable (late) release schedules (due to an unpredictable number of iterations) is some incremental approach. When you release parts of code *early*, and keep releasing *often*, you keep the effort more manageable, and the test product more transparent. As an added bonus you acquire a lot of data points with regards to the amount of rework needed “on average” before ETL code is ready for production. This gives you the most powerful empirical information to estimate your releases. And preferably, you apply a data model (like a Data Vault) where complexity of testing grows linearly with the size of your DWH (see tip# 5).

### 5. Data Vault Architecture Helps Control The Size/Complexity Dynamic

As systems grow in size and complexity, the challenges of managing this grow in a non-linear fashion. Testing faces the exact same problems. Your author asserts that exactly *this* dynamic is the culprit in the overwhelming majority of derailed ETL projects. In particular when the number of interfaces grows, and the complexity of mappings and transformations. This often leads to large, fundamentally *sequential* process flows with intricate interdependencies. As a result, when an error occurs this blocks testing of downstream flows. If you want to get a grip on a large project, you need to “divide and conquer”: design your architecture so that multiple “chunks” of ETL can be run and tested independently (and in parallel!).

The way star schema DWH's are loaded essentially precludes independence of loading streams and hence testing. The reason for this is that all the dimension tables need to be loaded *before* you can load the fact table. Workarounds in this architecture use temporary “filler” records to enable loading, but they still block proper testing. And like with all workarounds, it is fraught with difficulties. One of the unique strengths of Data Vault modeling is that independence of loading is *always* possible, and at negligible cost of rework.

### 6. Testing *Early* Is Cheaper

*Any* stage during the life cycle of development lends itself to testing. The cost of finding and fixing errors goes up dramatically as development progresses. Adjusting requirements prevents building low value solutions and preempts changes and fixes. When development is underway, each programmer can find and fix his own errors, which is much cheaper than having testers detect, report, play back, and then

retest the next release. And after a solution goes “live”, all these costs go up another order of magnitude.

Other reasons to begin testing early (even in light of incomplete documentation – see also tip# 10) are:

- You are rarely, *if ever*, in a position to do all the testing you ought to do, even the testing you sincerely believe you *must* do;
- In particular towards the end of a release, a lot of problem reports “simply” get deferred to future releases. The reason for this is that it may be too late to make changes, and planning changes is more likely to disrupt other concurrent plans, *in particular once get into a squeeze*.

### 7. Stress Test Your “Maximum” Load

Performance issues are perennial problems for data warehousing. Although hardware performance has gone up dramatically in recent years, users will *always* be able to dream up new and bigger queries that will get the DWH on its knees. And they will fire these queries that dim the light, too (although sometimes unintentionally...).

A crucial part of user acceptance testing (UAT) therefore is to test the DWH at its limit, *and beyond*. Have 2-3 times the expected number of users fire concurrent heavy queries, and see how the system responds. For sure, clever DBA tuning can help enormously, but don’t let that be an excuse to accept poor performance.

### 8. Testers Should *Never* Have A Release Veto

Sometimes a distinction is made between ordinary “testing” and Quality Assurance (QA). The latter is sometimes considered the keeper of quality, and therefore they might be granted a veto right over releases. As Jerry Weinberg has pointed out (Perfect Software, 2008), many of the organizational (political?) problems testers run into are directly attributable to misalignment of accountabilities.

Deciding when to release (or not) should be the responsibility of the project manager (or business owner). *They* bear the brunt of releasing too early and dealing with unsatisfied customers as a result. Or gain additional profits from being first (or early) to market. Testers should limit themselves to pointing out what tests they have performed, and what the results are. And provide all information that can help business owners make rational decisions about quality and schedule tradeoffs.

## 9. Testers Are Not The “Enemy” Of Developers

In many shops there is animosity between testers and developers. Yet they both aim to deliver quality software that provides value to their customers. Then why is this antagonistic dynamic so prevalent?

Testers often need to deliver “bad news”: we all know not to shoot the messenger, but still... Depending on their practices of writing bug reports and pin-pointing of errors, testers (themselves under time pressure and overburdened) are sometimes perceived as not adding sufficient value. They only tell what’s broken (and so they should).

Testers need excellent “soft skills” in order to maintain a professional relationship, and earn developers’ trust. Besides that, management needs to be aware of this dynamic, and act accordingly. And last but not least, testers should stick with their job: they are testers, period. They do not assure quality, they don’t make the product any better, they report *what* they tested, what the *outcomes* were, and maybe add some analysis of risks involved (to end-users) if you leave the product unchanged. That’s all (see also the previous tip# 8).

## 10. You Can’t Afford To Wait For “Perfect” Specifications

Some testing experts assert that testers should refuse to begin their work, until they get (good) specifications. The rationale being that unless you know what “good” system behavior is, there is no way to know when a result is “good.” However, DWH specifications are not always crystal clear. On top of that, during the course of a project changes are made to the requirements, as well as the solution, which makes it very difficult (near impossible) to keep all documentation up-to-date.

Waiting for specifications is one strategy, beginning work under severe ambiguity another. Testers don’t often have sufficient clout to be able to “claim” accurate specifications before starting. You don’t always get what you want. It often makes sense to do the best you can in suboptimal conditions, anyway. This is a profession, not a hobby. So go about it like a pro.