Database Lifecycle Management

Achieving Continuous Delivery for Databases

By Matthew Skelton, Grant Fritchey,
William Brewer, Edward Elliott & Knut Jürgensen

Database Lifecycle Management

Achieving Continuous Delivery for Databases

By Matthew Skelton, Grant Fritchey, William Brewer, Edward Elliott, and Knut Jürgensen

First published by Redgate Books, October 2017

ISBN: 978-1-910035-16-0

Cover Image: James Billings

Typeset: Gower Associates

Table of Contents

Introduction

If you work for an organization that produces database-driven software, and the database needs frequent or periodic changes to features, functionality, or data, then this book is for you. It describes how to apply the business and technical approaches of **Database Lifecycle Management** (DLM) to make all database processes more *visible*, *predictable* and *measurable*, with the objective of reducing costs and increasing quality.

Modern organizations collect large volumes of data, in a range of formats and from a range of sources. They require that data to be available in a form that allows the business to make fast decisions and achieve its strategic objectives. The DLM approach acknowledges the dual reality that databases are becoming increasingly complex and challenging to develop and maintain, in response to the business's demand for data, and that we must be able to adapt even complex databases in response to changing business requirements.

Given these demands, traditional manual approaches to database changes become unsustainable and risky. Instead, we need to increase **automation** while retaining and using the **expertise** of people who understand our organization's data. We need team structures that encourage **communication**. We need a framework within which to make **simple, step-wise improvements** to our core database build, test and delivery processes, and then to apply the **workflow** that stitches together these otherwise disparate processes into a coherent, automated pipeline to database delivery. We need to make databases easier to maintain, by using **instrumentation**, and by considering their **support requirements** as part of the design process.

DLM will help you achieve these goals. It starts with aspects of database design and data architecture, encompasses the database development and delivery processes, and extends to the support and maintenance of the database while it is in operation. It aims to ensure that an organization has in place the team, processes, methods, business systems, and tools that will allow it, with the minimum difficulty, to design, develop, and then progressively refine even the most complex databases.

This book will explain, strategically, how to use DLM to:

- Automate many of the critical processes involved in the design, development, delivery, and ongoing operation of a database.
- Improve the organization's knowledge of the database and related applications.
- Identify opportunities for optimization.
- Encourage technical and organizational innovation.
- Ensure that the database supports your key business goals, and drives strategic decision making, within the enterprise.

The overall goal is to help you evolve databases by a process of continuous, incremental change, in direct response to the changing data requirements of the business, and to improve the delivery, operation, and overall quality of your database systems.

About the Authors

Matthew Skelton

Matthew Skelton has been building, deploying, and operating commercial software systems since 1998, and for several years led a team that built and operated large database-driven websites for clients across many industry sectors. Co-founder and Principal Consultant at Skelton Thatcher Consulting Ltd, he specializes in helping organizations to adopt and sustain good practices for building and operating software systems: Continuous Delivery, DevOps, aspects of ITIL, and software operability. Matthew founded and leads the 700-member London Continuous Delivery meetup group, and instigated the first conference in Europe dedicated to Continuous Delivery, PIPELINE Conference. He also co-facilitates the popular Experience DevOps workshop series, and is a Chartered Engineer (CEng), and can be found on twitter as @matthewpskelton.

Matthew authored Chapters 5, 7, 11, and 12.

Grant Fritchey

Grant Fritchey is a SQL Server MVP with over 20 years' experience in IT, including time spent in support and development. He has worked with SQL Server since version 6.0, back in 1995. He has developed in VB, VB.Net, C# and Java. He has authored books for Apress and Simple Talk, and joined Redgate as a Product Evangelist in January 2011. Find Grant on Twitter @GFritchey or on his blog.

Grant authored Chapters 6, 9, and 10.

William Brewer

William Brewer is a SQL Server developer who has worked as a Database consultant and Business Analyst for several Financial Services organizations in the City of London. True to his name, he is also an expert on real ale.

William authored Chapters 1, 2, and 3.

Knut Jürgensen

Knut Jürgensen is a Master Data Administrator with over ten years' experience in this field. His current employer is an engineering company, based in Germany, which designs and manufactures sophisticated, mid-sized assembly lines. Previously he was the General Secretary of the Oranje Benefit Society in South Africa. Here, he was not only instrumental in implementing the Society's first computer system, but was also involved in the development of the software for the members' database which he subsequently managed.

Knut authored Chapter 4.

Edward Elliott

Ed is a SQL Server and .NET developer based in London, UK, who has a worked for almost 15 years in a mixture of support, development, and database administration. He is passionate about SQL Server development and loves using tools such as tSQLt, SQL Server Data Tools, DacFx API and ScriptDom to deliver automated and testable solutions. Follow Ed through his blog at http://sqlserverfunctions.wordpress.com.

Ed authored Chapter 8.

1 – What is DLM?

When the different teams that are involved throughout the life of a database fail to reconcile their different roles and priorities, and so fail to cooperate, or work adaptively, the result is gridlock: databases defined as though carved in stone rather than by code and data. DLM offers an alternative that allows databases to respond quickly to business change.

DLM combines a business and a technical approach to improving database development (or acquisition), delivery and management. It aims to ensure that an organization has in place the processes, methods, business systems, and tooling that will allow it to design, develop, and then progressively refine even the most complex databases more rapidly, and with less effort.

It acknowledges the paradox that databases are becoming increasingly complex and challenging to develop, in response to demands from businesses while, in spite of this, we must be able to change and adapt even complex databases in response to business change within the time required.

The scope of DLM is, as the name suggests, the entire useful life of a database, including aspects of design and data architecture, encompassing the database development and delivery processes, and through to managing and refining the database while it is in operation. It is intended to make all the processes that make up the lifecycle more predictable and visible, with the objective of reducing costs and increasing quality. It also aims to encourage innovation, and cooperation between teams.

DLM encourages the participants in a project to look at the processes that make up the lifecycle of any database, in broad strokes and in detail, to find ways of ensuring that the right things happen at the correct time and that every team has the best information to enable them to undertake their contribution.

Foundations of DLM

DLM is most concerned with improving the *processes*, or *working methods*, used in the design, development, integration, build, test, configuration, deployment, and operational management of databases. DLM uses well-tried approaches to facilitating complex teamwork to do this. It is less concerned with specific development tools and techniques. DLM also does not require adherence to specific development methodologies, although some methodologies will be able to exploit DLM techniques more readily.

DLM is concerned with providing the best chances that a database system or process can be improved, and that failures can be remedied rapidly.

To do it, they must be:

- **Instrumented**: This requires that processes within the databases must be easily measured for both performance and accuracy, and logged. Without a baseline, it is difficult to establish that it has been improved. Without detailed evidence, it is difficult to nip problems in the bud or track down bugs.
- **Resilient:** The cost of any unreliable server-based system is reckoned in man-hours wasted and trade lost. As well as constraints and checks for system integrity and defensive coding techniques, a resilient system has a well-planned hierarchy of warning and error systems with escalating alerts to ensure that stress and anomalies that warn of impending failure can be acted on well in advance. A resilient system is easily refactored and has a clear functional structure. There must be ways of monitoring excessive technical debt and preventing the deployment of failure-prone systems.
- **Managed:** All database operations and processes must be managed. This implies that they must be:
  - **Visible:** There should be no surprises. All the different teams involved in the lifecycle of a database should be able to check on progress, so as to be able to give timely advice, and to support progress. Source must be in a development Version Control System (VCS), deployment scripts, maintenance procedures and server settings in a Configuration Management System (CMS), and all used via automated scripts regularly or, if possible, continuously, to deploy the 'working version' to the integration server.
  - **Documented:** It should be possible for development team members to join a database project without excessive initiation. All participating teams must document sufficiently but not excessively, to allow all participants to contribute their skills and knowledge without 'reverse-engineering.' Decisions such as the 'when,' 'why,' and 'where,' as well as 'what,' need to be covered by documentation.
  - **Tested:** Before a system is released it must be tested against the release criteria. Before deployment, it must be tested against the criteria established *a priori* by the governance activity in conjunction with the business, end-users and operations. DLM uses automated testing where possible, but acknowledges the importance of manual testing.

- **Measured:** Where possible, all teamwork and actions such as approvals, test, sprints, and phases should be measured in terms of time, cost, and quality. Metrics from techniques such as static code analysis for technical debt, and performance (see 'Instrumented' above) would be useful aids to support judgment on quality. With baselines and measurements, it is much easier to prove improvements and refine processes.
- **Repeatable:** This means that the same steps can be repeated accurately in the same order to produce the same result. If a change must be made only once, it should be idempotent, in the sense that it has no additional effect if it is called more than once with the same input parameters. It also means that documentation, source, and archiving of information about the system is of sufficient quality that, with an impending problem, urgent remedial action can be guided sufficiently for the 'first responder' rather than having to waiting for the expert.

When a number of processes have been refined to a managed level, it becomes possible to employ, where appropriate, an iterative approach to development that allows a working database to be delivered in short cycles, and deployed incrementally to a production-like environment. This approach is helpful for a number of reasons, but is not essential in order to benefit from a DLM approach. It is a visible proof of having a managed process, but not necessarily an end result.

DLM relies on automated scripting and third-party tools to improve the speed, reliability and repeatability of each process from initial planning, through development and production support. However, it is not enough simply to improve the efficiency of individual processes. DLM also works to integrate critical processes within the database lifecycle. For example, in DLM, bugs reported to Support teams in production or test environments should flow directly into a central bug-tracking system, which itself integrates directly with the version-control system. When a developer changes the affected code in response to the bug report, he or she includes the bug ID in the code header so that the tester can verify that the change eradicated the bug.

Integrated processes and software tools by themselves are not sufficient to make DLM work. Effective DLM relies on very clear documentation of these processes, located on a central information hub for everyone associated with a particular database, to allow easier communication among those working on that database. It introduces and encourages workflow that is capable of conditional branches to make the review process easier to monitor, so that it becomes easier to integrate requirements such as regulatory compliance, resilience, data quality, production monitoring, and downstream reporting.

Data is central to business change, so DLM requires organizations to change the way that the business works with IT. Instead of a project-by-project creation of data-domain silos, an enterprise should maintain a data architecture (aka data model) that documents, validates, classifies and verifies its data across the enterprise from source right through to reporting, and establishes a vocabulary for naming that it understood by all. Databases must be designed in a way that facilitates change. While DLM will improve individual working methods for databases in a way that benefits any systematic development methodology, it is most effective as a phased, iterative process, because this encourages development cultures and supporting processes that facilitate change. It is no longer enough to assume that businesses are content to have databases that are unchanging in their design. Instead of current database applications locking the business data processes in place due to the difficulty of making changes, databases must reflect the data architecture as it evolves, in the light of changes to the organization they support. This means that databases must be easier to modify.

What DLM delivers

DLM is intended to make all database processes more predictable and visible, to improve the organization's knowledge of the database and related applications, to identify opportunities for optimization and to support decisions within the enterprise.

DLM will improve visibility of the database project to all participants in the database project, including:

- the final customers of databases, whether end-users, sponsors or teams responsible for downstream processing and analysis, or other applications
- application management and development teams
- the project managers with the responsibility of allocating resources
- the teams tasked with deployment, support, maintenance and decommissioning
- IT management responsible for compliance, quality, strategy and architecture.

It will optimize communication both within and across teams, allowing earlier and faster resolution of any potential conflicts, and preventing the need to make drastic design changes late in development.

DLM supports the several participating teams by overcoming the challenges of managing workflow and communications, and dealing with the issues of disparate priorities, processes, metrics, and infrastructures.

This is done in order to:

- reduce development time
- coordinate separate projects better
- improve the collection and reuse of domain knowledge and software techniques
- give better predictability to all parts of the process
- increase quality
- lower production costs
- drive team collaboration across development and operations teams.

DLM aims to use automation, workflow, and monitoring wherever appropriate, to improve the time-intensive processes of build, integration, release, test, and deployment, in order to reduce the workload required and allow more frequent and more reliable delivery of changes and enhancements. DLM supports database continuous integration and delivery where it is required.

The DLM landscape

The database lifecycle is usually referred to as:

- **New** (being designed or developed, or given the green light)
- **Emerging** (in production but in pilot form)
- **Mainstream** (in active production use, and being actively maintained)
- **Contained** (in production only for limited or legacy use)
- **Sunset** (scheduled for retirement)
- **Prohibited/Retired** (no longer used).

DLM extends much further than just the development lifecycle, covering three major concurrent streams of **governance**, **delivery**, and **operations** throughout the phases of DLM. These are activities that may map to several different teams but, in smaller businesses, be managed by a one or two teams. A stream does not relate directly to a team, and many believe it shouldn't do so, because of the importance of coordination.

For a project to succeed, all these activities must be done right. If a project gets the initial governance aspects wrong, it is unlikely to provide much business value no matter how well it does the development and operations. Similarly, however successful a database project may

be in tackling the problems of the development process, it is doomed if it neglects operational issues, such as training, support, or providing enough resources to run the application reliably. Once again, the business value this investment provides won't be as large as it should be. DLM techniques can provide a broad view of database applications to help organizations avoid problems like these.

New	
Governance	Identify business needs and data architecture. Determine mix of bought-in, commissioned and internal development work. Identify business priorities for the database. If in-house development, work with development teams on the planning role for development. Identify all enterprise-wide data and processes shared by applications and plan data interfaces and feeds. Ensure the participation of all IT expertise in the organization for security, compliance, etc. Agree a resilience plan with the operations activity in conformance with the standards of the organization.
Delivery	Work with governance to plan out all development work sufficiently to allow cost estimation for development to the 'emerging' level. Provide roadmap for iterative development. Execute all necessary development work, and plan out the detail of all supporting Extract-Transform-Load (ETL) and reporting systems and services. Ensure that development is a managed process. Determine all database requirements to comply with the legislative framework.
Operations	Provide detailed plans for hardware platform, along with server and network configuration. Work with development team and governance to host the pilot version of the database. Advise governance on the requirements of maintenance, monitoring, support and training. Establish likely demands for scaling and evaluate alternatives. Establishment of a CMS archive. Work with governance to make sure that the hosted production database meets Disaster Recovery (DR) plans and High Availability (HA) Service Level Agreements.

Emerging	
Governance	Work with the business, development team(s) and all downstream consumers of the data, e.g. Business Intelligence (BI), to ensure that the database meets the contract for quality and specification. Check that the database meets requirements for operational support and maintenance.
Delivery	Regular or continuous deployments to deliver new and changing functionality. Work with operations to develop features to improve database instrumentation and resilience, and general maintainability.
Operations	Training for users, and for support, maintenance and high availability. Implement whatever replication, warehousing, high-availability DR scheme is required. Monitor performance for index problems and poor algorithms. Work with governance on security issues.

Mainstream	
Governance	Check for compliance with the regulatory framework, and plan for changes that are a consequence of business change (e.g. mergers or acquisition). Check for compliance with corporate or company-wide data policies and standards.
Delivery	Deployments to deliver new and changing functionality in line with changes in the business and the legislative framework. Also, changes in consequence of database version and OS version upgrades.
Operations	Work on ensuring that all likely points of stress or failure are matched with documented processes for correction. Monitor and test all DR strategies. Improve CMS archive to ensure all maintenance tasks are possible without special knowledge or heroics.

Contained	
Governance	Continue to monitor for changes that are required as a result of security concerns, legislative and business changes.
Delivery	Deployments only for required changes as specified by governance.
Operations	All operations such as maintenance and monitoring done as a routine, relying on documentation and CMS archive.

Sunset	
Governance	Ensure that all requirements previously met by the database that are still relevant are now met by other systems and migrations are in course. Continue to assess security risks.
Delivery	Provision of systems to provide any continuity in functionality that is required.
Operations	Continue all maintenance as a routine, relying on documentation and CMS archive.

Retired	
Governance	Plan for archiving of data.
Delivery	Ensure the retention and long-term archiving of all development source code.
Operations	Checks that media can still be read while data retention is required.

Governance

Governance encompasses all of the decision making, planning and project management for a database throughout its life, with the aim of making sure that the application continues to provide what the business needs. This role is likely to be supervised by different activities within the organization at different stages of the life of a database, from initially determining the business case, through to eventual decommissioning. Once a database becomes part of the portfolio of applications that are used by the organization, the governance activity then provides ongoing assessments of the database's benefits, risks and costs and determines when revisions and updates are necessary.

Governance starts at project conception, identifying the business needs and objectives and continues to the end of the lifecycle to ensure that the requirements are met and the database application is then correctly maintained, reviewed, and updated as needed.

Planning

The planning role is part of governance. The definition of the product requirements is based on the viewpoints and requirements of customer, company, market, and regulatory bodies. A database specification is produced from which the database's major technical parameters and architecture can be defined.

This has four important aspects.

- **Remember:** Check previous projects to see where existing domain knowledge and software techniques can be reused. Learn from mistakes in previous projects, and within the industry.
- **Innovate:** The conception stage is the ideal point to check whether new frameworks, tools and techniques can reduce the delivery time, yet fit the requirements.
- **Specify:** The requirements of the database must be documented, and the main processes identified. Organizations are required to identify and understand the key initiatives, pain points, and business problems they are trying to resolve by changing the database. All upstream data feeds and applications, and downstream reporting and analysis applications, must be identified along with any restrictions that are likely in movement of data. Before delivery can be attempted, there must be a thorough understanding of data right through to its source, its nature, constraints and characteristics, along with estimates of volume and distribution. Is this structured or *fluffy* data? What are its lifetime and retention requirements? What audit requirements are there? How mutable is it? What naming is in place within the user's domain to refer to the data?
- **Plan:** At the conception stage, planning involves the creation of architectures, chiefly the domain or business architectures, preferably modeled diagrammatically and given a name so that the delivery process can adopt naming conventions at the object or entity level. These architectures are then used to identify the logical areas of the database and to provide rough estimates of timescales and the likely effort required to deliver the database. There should be a plan for the overall design that is sufficient in detail to allow the delivery process to start. There should be sufficient high-level and detailed use-cases to provide a foundation for documentation, training material, and test plans.

Delivery

Although delivery is split into design, development, release, and deployment, it is possible with certain types of database, particularly when close-coupled to an application, to automate the bulk of the repetitive development and testing processes that teams use to deliver, manage, and maintain the database and application. This allows continuous delivery from the point of version-controlling changes, to deploying them to different environments, and, when ready, opting to deploy to production. If done well, it can help teams to reduce risk and increase both efficiency and reliability in the database delivery process. Whether or not this can be adopted, the processes, and the requirements for cross-team communication and automation, remain similar.

Design

In any relational database, Entity-Relationship (ER) modeling, assisted by Unified Modeling Language (UML), is important. The ER model should be maintained so that it reflects the database definition in code, as it evolves. Although code can be generated from an ER diagram, it is normally merely first-cut code that is then developed at the object and schema level.

Increasingly, large databases are broken down into more manageable logical components and the implementation of a logical component could occasionally involve other types of databases, such as network databases or document databases. In such cases, all components of the database must have interface-specifications, with the aim of minimizing interdependencies. In this way, the database can be developed in phases and deployed independently, so that business value can be delivered as soon as possible.

Part of the design process involves the security model, preferably based on schemas and interfaces. At this stage, the requirements of the production team for support, database administration, reporting, monitoring and compliance are recorded. Policies for naming conventions, code formatting, documentation, and security will be identified and agreed.

Development

The development phase of the lifecycle begins soon after the stage of refining and approving requirements. Development is best done iteratively, with a Lean, frequent-release cycle. As well as responding to new requirements from the governance process, the developers will need to respond to the need for changes to any version of the database in production, to deal with bugs and incorrect indexing.

Design and development work should be continuously validated against the user-requirements, and prototyped in a way that encourages participation. The validity of the work of each database development phase, for each logical component, should be continually checked against the output of the previous phase. All changes should be synchronized and verified continuously, with the accent on testing. Smaller, more frequent database changes can allow development teams to integrate their work sooner, see testing results and feedback earlier, and act on any issues much more easily. This makes it possible to release database changes frequently and more quickly.

All development work should be verified for accuracy and performance by means of automated testing, through simple unit test, before check-in of changes. *Mocks* are created where necessary for upstream and downstream dependencies.

Regularly, ideally daily, the team should build the current working version of the database, via scripting from the canonical version in source control, and then compare and synchronize it with the version being used for development work.

This version should be subject to automated integration testing for performance and accuracy, and the results of this testing made easily accessible to the developers.

Release

In DLM, the release of a database version is part of a change order, specifying all changes to the design, interfaces, and build requirements. Where relevant, the release will also include references to the precipitating change request, or bug ticket.

The release manager / change analyst, (often a project manager), is responsible for expediting the change order through the release process, with the objective being to safely manage and monitor the release through to deployment to the customer.

The database scripts are, at this stage, object-build scripts, but any data-migration scripts that have been developed are included. Along with the database build scripts, the release process will result in a *build artifact* that contains everything required for the build. This can be in a central CMS repository as files or a zipped NUGet or DacPac. This will have been validated in a continuous integration environment as well as in testing and Quality Assurance, but not yet against a production server.

At this stage, several checks or *approval gates* are required. This will probably include hardware platform requirements, implications for compliance, performance testing and review, and confirmation of user acceptance. It will need to be checked for compliance with production requirements for maintenance, support, and monitoring.

The release code and components are archived in a central configuration management archive along with all scripts required for deployment to test and staging. Normally, final tests are made for compatibility with all dependent systems and data feeds. Where scalability testing is difficult to achieve as part of the development cycle, it is delayed until this stage. Where database tables are changed, migration scripts must be created that preserve the existing data and assign it correctly within the new table structure. These are tested against a database server which is as close as possible to production. Rollback strategies are devised and tested at this stage.

Deployment

The deployment phase aims at setting all aspects that are required to make the database operational. To do this, the build artifact from the central repository must be deployed to a staging database set up as a copy of production, in order to generate an upgrade script for deployment. This may involve setting the appropriate infrastructure, deploying the database in its production environment with appropriate data, or preserving the current data. It might also require associated frameworks or libraries. For a complex database, it might require delivering training materials and documentation, and using these to train operations staff. Operations may also require hotline support to be set up for complex issues.

The DBA is likely to have to deal, in script, with issues that are beyond the remit of the developers, such as access control, hardware specification for data, replication, messaging, alerts, ETL systems, and interfaces to downstream systems dependent on data.

Different teams will have different requirements for this process and there are a range of third-party tools that can help to automate the process that is in place. The size of the production system will dictate the alternatives that are suitable.

As well as confirming and implementing the plans for security, resilience and availability of the database, the DBA will want to review the whole artifact, together with the upgrade script, to check whether it is production-ready, and detect whether there have been any changes to the version of the database in production, since the developers generated the validated script that was used with the build artifact (*database drift*).

All databases that are involved in a deployment keep a record of all the changes that have been successfully applied to them. There is also a full record of hardware specification, software installation, and database configuration for all servers and databases involved in production.

There is likely to be an approval gate, with restricted access, as part of the process of deploying to a live production environment. This helps to implement the separation of duties between development and operations when this is a requirement of the system. By maintaining a central CMS archive, database administrators and other participants in the deployment process can use source control and workflow to track, and report on, all changes, issues and sign-offs that mark the progress from development into testing, QA, pre-production, and production environments.

Operations

The operations activity has relevance at every point in the database lifecycle. In particular, the team will have direct responsibility for the functions below.

Access control

Access control must be closely allied with domain-wide access control, and require as little DBA intervention as possible. This is achieved by means of schema-based access control using role-based security.

Security

Operations will probably set up intrusion-detection systems and put in place systems for detecting malicious version drift. There will be auditing systems planned in conjunction with governance. These should be designed with a long-term view of simplicity for the *contained*, *sunset*, and *retired* phases of a database where security checks cannot be relaxed.

Maintenance

As well as supervising the regime that is designed to provide resilience and high availability, and ensuring that scheduled processes complete successfully, there will be the task of fixing application defects, applying hotfixes for missing or inappropriate indexes, reporting software bugs and determining temporary workarounds, and managing the addition of new functions to the application.

Support

It is likely that there will be some requirement for multi-level operational support of a database system, unless this is entirely provided by the applications: For hotline support, all the likely FAQs that are relevant to the database (e.g. database messages and warnings viewed on the users' screens, or connection problems) need to be documented. If this requires participation from development teams, then it needs to be fed into the governance process as a production requirement. When an issue happens, several levels of hotline and on-site support will be required, depending on the complexity of the issues. These issues will vary from those easily dealt with by hotline, to those which require a new database version. Some support teams will need guidelines that allow them to decide to what level to escalate a support question.

Training

Training is required for any major change to a system. Normally this is supported with materials supplied by the application development teams and doesn't directly affect the database. However, training materials for production staff that are required to perform maintenance for the business aspects of the database must usually be provided. Where there is a rapid deployment of changes to a database system over several months or years, there will soon be a mismatch between the production system and the training materials. Ideally, changes in training materials and support materials should be kept in synch with changes in the database and applications.

Monitoring

It is important to be able to head off likely failures when they are only visible as trends within a monitored system, and before they develop into problems or failures. This requires monitoring systems that can produce alerts on trends away from established baselines. As much of this as possible needs to be automated. Although monitoring systems can help "off the shelf" they cannot measure or monitor business-specific trends within the database, such as business transactions or functional usage. For this, the database itself requires instrumentation. It is likely that an effective monitoring system will be a mix of monitoring tools and scripted processes.

Resilience

All systems for providing high availability and disaster recovery will be agreed with the business and implemented. After implementation, they will be tested to ensure that they are effective and up to date.

DLM techniques

DLM is impossible to achieve without being able to automate many of the time-consuming routine processes. The success of DLM also relies on a handful of core working methods that underpin the manageability of the development process. None of these are unique to DLM.

Source control

This makes it easier to share code, and make it easier to inspect. It ensures that no changes can get lost and gives a history of changes to code, explaining when, what, and why a change was made. It allows a deployment to always be made from a known state or version. It makes it easier to check, supervise, approve, and test code as new functionality is developed, and ensures that the right version of code is worked on. It also allows developers to work on different branches of the database, while ensuring that modifications to the core functionality of the database are preserved in all branches.

Preservation of existing data

Existing data in a database that is the target of a deployment is preserved by means of a migration script. Although a synchronization tool such as DACFx or SQL Compare can normally determine how data should migrate after a table change, a hand-cut migration script allows, where necessary, a more precise way of determining how schema and data changes are made: it is much more easily tested because it automates the change from one version to another. Furthermore, for some types of database changes, the synchronization tool would be unable to interpret the precise intention of the developer and would risk loss of data as a result (table-renaming, splitting or merging columns, or changing the data type / size of a column). Checking in migration scripts also helps developers to specify exactly how to make specific database changes, especially where a more obvious method would lock essential tables for a significant time. Migration scripts will often involve more than one object, so should be stored separately. The build process needs to figure out where the migration script needs to be applied.

Deployments can be made without synchronization tools where all changes are specified by migration scripts. In this case, source control contains immutable migration scripts rather than object-build scripts, and the build is done by progressive recapitulation of the migration scripts in their original order. This is more appropriate for relatively small databases, unless each release is consolidated into a single script from the individual changes.

Bug tracking

Where database developers are working closely with testers, and are running regular automated testing, it pays to relate bug fixes with changes to code. A change should, where relevant, refer to the bug it fixes. This should also be true of feature requests and feedback from user acceptance testing. This is particularly useful where documentation can be updated to pull together information from the separate systems.

Configuration management

In the same way that the DML and Data Definition Language (DDL) source of the database is stored in source control, all materials that are shared between ops staff should be held in a central repository. This is particularly important for the scripts and structured documents that are used for the automated deployment of databases to a range of servers.

Documentation

Documentation is not only helpful for delivery, but it is also the key to speeding the work of readying a release for deployment. It supports a closer integration between development and operations, and makes hunting down bugs easier. To save repetition or cut and paste, all documentation should have a single canonical source, whether it be in source control, the code itself, the database, the bug-tracking system, the workflow system, or as a separate document in source control. The assembly of the documentation needs to draw the appropriate documentation from all the separate sources, using automation wherever possible.

Logging and log management

Logging is useless without good ways of parsing and analyzing the logs so that they can be searched rapidly. Ideally, logging information should be stored in a searchable repository from which graphs, reports, and alerts can be easily generated. This is particularly important with database performance and scalability testing, where a large amount of timing data and counters have to be retrieved and analyzed.

Collaboration

There are a range of collaboration tools to make teamwork easier to coordinate. These range from simple, team-based, chat systems to full workflow systems, but there is no consensus on what represents the ideal way of coordinating across teams and ensuring that tasks such as sign-offs are done in a timely manner. There seems to be a preference among developers to use as lightweight a collaboration system as possible that is sufficient to allow remote working and sufficient visibility to other teams.

Monitoring

The delivery process as well as the database itself must be instrumented and monitored. This applies to the progress of the release process through to delivery and deployment. Obvious candidates for monitoring would include completed workflows, source-control actions, documentation items, static code analysis metrics, bug resolution, failures in automated integration tests, and performance metrics for the database under load tests and scalability tests.

Automation-assisted tests

There are many points where tests have to be run. Test assertions on compilation of functions or procedures, tests for table constraints, unit tests for any routine, integration tests for processes, performance tests, scalability tests, security and access-control tests and many others. These tests will supplement manual testing and user acceptance testing but are an integral part of the build and deployment process. They require data that simulates real production data in its distribution, characteristics and volume. This data must be machine-generated, or obtained via obfuscation of real data.

Tests can be done via a framework such as **`tSQLt`**, or a generic test system. It can also be done via PowerShell scripting as long as care is taken with presenting results clearly.

Scripting and workflow

Although tools such as Puppet/Chef, and build/release tools such as Jenkins can help, PowerShell scripting is likely to be the bedrock of any windows-based DLM system because it has support for SQL Server, WMI, Active Directory, performance counters, Server Management Objects (SMOs), DacFx and a host of others. Because of its support for remote operation and Workflow, PowerShell is able to script sophisticated processes within the Windows domain.

Octopus Deploy provides PowerShell pre- and post-deployment scripts that can be executed on remote machines outside the Windows domain. Most build and release servers allow functionality to be extended via PowerShell scripting. PowerShell scripts must be kept in a CMS archive along with server setting and other configuration files.

Summary

The best way that database systems can meet the requirement for increasingly complex databases, and ever more rapid change, is to adopt the same principles that revolutionized manufacturing in the late twentieth century, and which were used effectively for Application Lifecycle Management in the past decades. It basically means paying attention to making the techniques, processes, and tooling for database more efficient and better integrated. Although these techniques are primarily aimed at making the business of creating, developing, and maintaining databases more manageable, repeatable, and visible, DLM has the side-effect of improving quality and reducing time to delivery. DLM is quite different in many respects from Application Lifecycle Management (ALM) because of the complications of data and the general connectedness of databases, as well as the greater requirement for up-front planning and compliance with organizational policies and standards. Data represents the most valuable asset of many enterprises, and has unique requirements as a consequence.

2 – DevOps and DLM

DLM aims to provide a roadmap of what is required, and when, to deliver and maintain effective databases that can respond quickly to business change. How does the DevOps movement, as it applies to databases, fit into this?

This chapter explains how DevOps provides the necessary organizational change between the delivery and operations activities to make the more important parts of the process easier to introduce.

What is DevOps?

DevOps, originally known as "Agile Infrastructure," started as a groundswell of frustration at the entrenched segregation and autonomy of the three application-oriented activities within IT: **delivery**, **operations** and **governance**. There was no established methodology that dictated such an "Iron Curtain." On the contrary, methodologies such as Catalyst had been advocating a more cooperative cross-specialism approach for over 20 years.

None of the development methodologies, such as Structured Systems Analysis and Design Method (SSADM), advocated this silo mentality either. No "waterfall" approach had insisted on a Chinese wall between operations and delivery. However, IT practices had become entrenched in a particular way of working. It was a cultural problem rather than a technical one, but the growing need for a more rapid and predictable delivery of change to ecommerce sites brought matters to a head.

DevOps is all about bridging the cultural and technical gap between different teams so that practices such as CI and continuous delivery (CD), which demand a high level of cooperation between teams, become possible.

Done well, DevOps represents a transfer of effective practices and methods between operations and delivery teams, infecting ops people with some of the best development practices such as source control, versioning, and issue tracking, and automated processes like testing. In turn, operations teams give insights to the delivery teams about concerns such as infrastructure monitoring, immutable architecture, real-time alerts, configuration and integration management, and the requirements underlying high availability, delivery, and hosting.

In some cases, it may involve operational people working "embedded" within delivery teams, or developers working on operational tasks, but it doesn't require this. The focus is on the exchange of ideas and expertise. It does not tackle the sometimes-conflicting concerns between governance and delivery or operations, nor does it encompass the very earliest stages of planning a database, or the mature system that is no longer being actively developed.

Both delivery and operations have aspects of responsibility that are of little outside interest. Ops teams have a responsibility to maintain many processes, applications, and databases that aren't being developed in house, or aren't being actively supported at all. Delivery teams have responsibility for implementation detail that is of no interest to either operations or governance. DevOps is relevant where the interests of delivery and operations meet.

The difference between DLM and database DevOps

How does the DevOps movement, as it applies to databases, differ from DLM? The short answer is that database DevOps focuses only on one part of the database lifecycle; the intersection between delivery and operations. It is, to be frank, a very interesting part, but DevOps for the database doesn't aim for a broad view of the whole life of a database. Database DevOps is more concerned with enabling good cooperation between development and operations to facilitate the rapid delivery of database changes.

DLM, on the other hand, aims to make sure that the development, use, and replacement of a database system achieves a measure of quality and scientific method, rather than relying on individual heroics. It encourages teams to evolve their way of working from the chaotic to the managed. It sheds light on all the necessary activities of providing a technical service to users, even those that are of no interest or concern to most developers.

Is DevOps necessary?

As Internet commerce took off spectacularly in the "noughties," a new breed of application became more common, one that was not only the public face of the organization but was intrinsic to the organization.

The organization *was*, effectively, the application, but unlike the classic applications of commerce, these Internet applications had to evolve continuously in the face of competition and demand from users. They did not conform to the accepted lifecycle of previous applications, and could not afford any relationship with the business other than total involvement.

The traditional divisions and practices of established IT simply weren't appropriate any more. A new way of team-working had to evolve to make it possible to develop applications rapidly and deliver new functionality with as little delay as possible. DevOps in this context wasn't just a nice idea, it was essential.

Today, most organizations aren't in this predicament. Instead of focusing on a handful of applications under continuous development, with a plethora of BI satellites, most applications have a considerable operational task with clear-cut objectives and constraints. In many cases, the delivery activity is focused on relating the data-flow between bought-in applications, and on both reporting and BI. The creation of new business applications is far less crucial to the business.

In this sort of setting, DevOps isn't essential, but does it represent good practice? The answer must be that the DevOps movement has developed techniques to solve the challenges of rapid delivery, and these techniques have turned out to be generally useful. It is particularly useful in helping an organization move from big intermittent releases toward Continuous Delivery.

It makes it easier to introduce version control for all aspects of hosting applications in organizations. It is likely to make applications easier to support and maintain, through making developers aware of operational requirements earlier in development, and by "shifting-left" many of the test, security and compliance chores that were customarily postponed until the deployment pipeline.

To be effective, though, DevOps needs to go hand-in-hand with a renewed spirit of cooperation and culture-sharing with the governance function: otherwise, we are in danger of delivering the wrong functionality, more quickly and efficiently.

DevOps and the database

There was an idea that gained a lot of traction when DevOps was maturing; the database, like the application, was "just code," and should be treated as part of the application. A database, in this way of thinking, is just a slightly weird component of the application that can be coded, built, tested, and deployed alongside the application code.

The truth is rather more complex. It is true that we can generate a build script for everything within a database, and these build scripts can reconstitute the database. This is true for both the data and metadata, the DML and DDL.

However, database developers and DBAs use several ways, including ER diagramming tools, wizards, or interactive table-design tools, to develop a database, and the code is often generated retrospectively. Each build must bring all the development and test databases in line with the current version, while preserving the existing data. Where changes affect base tables in ways that change the data, this requires a migration script, or changes to the data import process.

There are also server settings that either affect the way the database works, such as server configuration, or server-wide resources, such as Agent jobs or alerts, which are not part of the database but are required for the functioning of the database. There are also database settings that can be enshrined in code, such as file locations, and these must be changed as appropriate for the target server.

The database can certainly be expressed as code but the build process, where the database is created from code, is only part of the construction of the working database. It has to be configured, it requires scheduled tasks and could be participating in replication. It needs a maintenance regime. A database will be taking upstream feeds, and supplying downstream applications and databases with data. All of this means that even relatively small databases within an organization require input from many different people, from both operations and development backgrounds, to make them happen. In short, it is easier to work in isolation with application development than with database development: the latter is always a team-based activity.

For any business grappling with the issue of how to evolve applications rapidly in the face of competition, another problem is that relational databases aren't amenable to rapid change once they have become established. Relational database practice assumes that the data is well understood before the database is designed. Changes to the schema of a complex database

can be tricky. It's not only the difficulty of making the schema design changes, and corresponding changes to every dependent database object, but also in adapting the data-migration processes. Databases don't easily fit the Agile paradigm, though Agile does acknowledge the need for a good understanding of the business domain before development starts.

Initially, NoSQL offered the tempting hope that, by being schema-less, one could get all the advantages of relational databases without the pain. Sadly, they had forgotten that RDBMSs provide ACID-based data and transactional integrity, and these qualities couldn't be fudged by promising "eventual consistency." NoSQL databases were fine for the type of data for which they were developed, but were dangerous for transactional operations. Some spectacular Bitcoin Exchange collapses (see https://www.wired.com/2014/03/bitcoin-exchange/) convinced even the NoSQL die-hards that transactionality was required for any commercial activity.

Microservice architectures, developed from the ideas of service-oriented architectures, also offered the hope of avoiding the need for an enterprise-scale RDBMS to manage the transactional integrity of business processes. Architectures that are dependent on a large "uber-database" to provide transactional safety and data interchange were held by many developers as being restrictive, because they seemed an unnecessary restriction. Developers talked of the delays inherent in an institutionally-oriented operational culture that was resistant to fast-moving changes to applications. Here again, the problems of managing distributed transactions reliably had been grossly underestimated. However, where transactionality isn't an issue, this architecture can cut down on interdependencies and therefore make development easier.

The most promising solution to this problem of how to accommodate fast-developing applications that need to deploy continuous changes to a changing market, is to understand the nature of the data involved (see Chapter 4, *Managing Data as Part of DLM*). This makes it possible to use graph databases, document databases and other specialized database systems for unstructured or semi-structured data, relational databases for transaction processing, and OLAP/Cubes for business intelligence. In other words, a heterogeneous data platform is adopted, rather than a single platform.

Is there such a thing as a DevOps tool?

Before DevOps, there has a temptation for some of the major players in the industry to exploit a change, or new initiative, in the approach to software development as an opportunity to tie users into a "Godzilla Integrated Development Environment (IDE)," i.e. an all-encompassing solution for the entire application delivery process. This would include the obvious components such as build servers, source-control system, bug tracking, provisioning and so on. Rather than try to integrate existing applications, behemoths were created that solved the problem of integration and communication by not having to communicate at all, as theirs was the only necessary application.

Ordinary people working in IT were even less inclined to buy that dream than senior management were. Over the past two decades, software tools have become more specialized and component-oriented. Delivery and operations people demanded that software had to work together, not only on Windows or Linux, but across platforms. It meant that tools needed to cohabit, rather than dominate.

For DevOps to work, a delivery team and operations team needed to be able to pick and choose from a whole range of specialist tools, often open source, each serving to automate a specific process, such as building, testing, packaging, releasing, configuring or monitoring. They then had to rely on all these tools working together in a command-line interface (CLI) "toolchain," with each one taking their input from the previous tool in the chain.

For network intercommunication, components can expose their services as Representational State Transfer (RESTful) services. The accent is on simplicity and ensuring that tools that support development processes can, where necessary, conform to the demands of the toolchain.

Microsoft was late into this change in the way of managing processes, but with a fairly clear field and little need for backward compatibility, they were able to introduce a scripting technology, PowerShell, which was able, not only to provide the CLI-based toolchain, but also to allow objects rather than documents to be passed along the chain, and use all the power and features of .NET as well. PowerShell is close to the old command-line batch scripts, but also takes inspiration from Bash scripts.

Windows DevOps tools tend to be PowerShell-aware, or even shipped as PowerShell cmdlets, and they can also be useful participants in a PowerShell script by merely using their CLI.

Can a relational database be Agile?

By introducing DevOps work practices, can we make database development and delivery more Agile? The short answer is "Yes," but it requires some compromises in the way that data is accessed from front-end applications. Agile relational database techniques are well-established but rarely practiced.

For rapid database deployments to be acceptable to the business, they need to be as risk-free as possible. By default, a new release immediately exposes the latest functionality to all users. However, if it is possible to preserve the functionality of the original interface intact, and expose the new functionality only to a few users while allowing immediate rollback, then this risk can usually be reduced to the point that it doesn't affect the business significantly. Where the delivery team are working on the deployment alongside the ops team, it becomes easier to manage an operation that requires a lot of knowledge of the system. If, for example, SSIS needs to be changed because a data source changes its network address, the task needs both knowledge of the application and the production environment, and the developers can use the insights they gain from it to automate the task in future. Where the staging and production servers have different drive configurations, the job of altering the build scripts accordingly needs to involve both teams, and both dev and ops have skills that can be brought to bear.

Databases have a particular problem with the rapid delivery approach, which is the need to be certain of preserving all the changes to data while allowing a controlled delivery of new functionality. It requires that the database is designed in a particular way to achieve this, and relies on the same discipline, in insulating the database base tables from the application(s), behind an interface defined at the database level. Base tables do not represent a satisfactory interface!

There are some features in the SQL Standard, such as table-valued functions and synonyms, which lend themselves well to aspects of Agile development, such as the use of feature toggles. It is perfectly possible to expose different versions of the database to different users while keeping the underlying data up to date, and to change the version of the database exposed to a certain user, merely by changing the user's Windows group.

A release that relies on feature toggling needs a lot of testing, and defensive development. It is best to use feature-switching techniques that can be controlled simply by making changes to user access, so schema-based security is ideal. The downside to this is that schema-based

access requires considerable re-engineering of an existing database. Feature-switching via code changes is ill-advised. The worst mistakes come from switching features via global toggle variables, especially if they are recycled (see Phil Factor's article at www.simple-talk.com/blogs/a-knights-tale/).

Summary

There has always been a gap between good practice and organization, and the conventional structure of the average IT department. In terms of the lifecycle of the application and database, organizational convention often dictates a strange wall of non-communication, sometimes even non-cooperation between the delivery activity and operations. This has never been sanctified or condoned by any known methodology and gets in the way of delivering updates to an application or database.

To be sure, a delivery team is focused on their project, whereas an operational team has a wide-ranging responsibility for the whole range of current applications, databases, and processes, and the way they interact. They are very different perspectives. This means that cooperation and the exchange of ideas and techniques is valuable but the two teams can only merge into a single team in the unusual case where an organization has just one application that requires operational support.

DevOps is an initiative that comes from frustration felt with an organizational system that has evolved in an adaptive way. It stems from individual professional people caught up in the resulting confusion. It makes it easier to introduce many of the techniques that underlie the best DLM practices, especially the delivery pipeline and database provisioning, and it focuses on part of the database lifecycle. It is not concerned with the entire sweep of DLM, and has little to say about governance or the operational support of databases that are no longer being actively developed in house.

It has, however, been a very positive influence in making it possible to take part of the database lifecycle, the delivery pipeline, and encourage both good practices and new initiatives in reducing the time it takes to deliver change.

3 – Planning for a Successful Database Lifecycle

Although it is well-known that the best efforts of a development team can be derailed by mistakes in the architecture, design, and general governance of a development project, few attempts have been made to describe what needs to be done to increase the chances of success in the development of a database application. This chapter will itemize what a delivery team needs to succeed.

Many of the problems that are faced during the development of database-driven applications are nothing to do with the delivery team at all, but are the result of poor preparatory planning work. The delivery process relies on other prior IT activities and especially the ongoing collaboration with the governance team. Some dramatic development failures have, for example, been entirely due to a failure by the IT governance team to understand the organization(s) that are destined to use the application; sometimes even if it is their own organization that has the intended users.

It is sometimes hard for the developers within a delivery team to accept that they rely on anyone else for the success of their work but, in fact, a successful application depends on close teamwork between the three key activities, **governance**, **delivery** and **operations**: the opportunities for failure are widespread, and often the source of failure is so subtle that the developers, by default, have the odium of failure unfairly heaped on them. (See OASIG 1996 and *Software Development Failures: Anatomy of Abandoned Projects* by Kweku Ewusi-Mensah). If the necessary groundwork isn't done first, no architecture can save the subsequent building.

This chapter is aimed a describing the work that needs to be done, what is often described as the *required steps*, to ensure the best possible chance of success for database projects throughout their life.

Why the need to describe the required steps?

Although there was a great deal of agonizing in the 1980s about the difficulty of developing database-driven applications, (see *Normal Accidents: Living With High-Risk Technologies*, by Charles Perrow, Basic Books, New York, 1984.) things came to a head in the 1990s, when the general skills-shortage within the industry caused damaging failures and over-runs in software projects. There was a particularly low point in the 1990s when the "software crisis" resulted in fewer than one in five software projects being successful.

> **"… every component of good engineering practice had been ignored, every guideline of software engineering disregarded, basic management principles neglected, even the dictates of common sense overlooked."**
>
> *Finkelstein & Dowell: A Comedy of Errors: the London Ambulance Service case study 1996*

The Standish Group of U.S. companies and government agencies found, in 1995, that a third of corporate IT projects were abandoned before completion, at a cost of $81 billion. The Standish Group reported similar figures for 1998. A common theme was the chaotic state of software development. Out of the subsequent heart searching, both in the UK and the US, came several initiatives, including Agile, Information Systems Lifecycle (ISLC) or the Software Development Lifecycle (SDLC) and ALM. Specifically, for databases, there was the Database System Development Lifecycle (DSDLC), and closer to ALM, there was DLM.

Because these separate initiatives aimed to fix the same general problem, there was a lot of overlap. What made ALM and DLM attractive was that they set out the development and delivery process within its wider context. This made it possible to identify many of the management failures that were responsible for making the work of delivery more difficult, and the delivery shortcomings that made a database-driven application more difficult to manage subsequently in production. It could also identify issues in operations that could cause a perfectly good development to hit problems in production.

ALM and DLM were based in turn on Product Lifecycle Management (PLM) which had revolutionized product development in manufacturing industries in the late 1980s. These techniques allowed different teams to see in far more detail what was happening in a project, and allowed better cross-team discussion and participation. It also spelled out some of the processes that required cooperation between teams to make them happen.

DLM and process improvement

DLM aims to provide the optimum environment to allow a range of development methodologies, such as Agile and XP, to thrive. It values experimentation and innovation, and aims to provide faster feedback of results. It does not prescribe a particular technique. It merely aims to make the whole lifecycle of the database repeatable, controlled, predictable, defined, measurable, visible, and optimized.

It aims to build a culture that values process improvement. It does this by means of more effective planning, automation, collaboration, cross-functional teams, an integrated toolchain, and an output that is tied closely to the aims of the organization.

Where possible, it encourages measurement to quickly give feedback about process change. DLM is rather different from ALM, because databases, by handling business data directly within a variety of constraints and conditions, tend to be nearer the core of the organization that uses it, and require more work to have been completed before delivery can begin to "cut code."

No organization can make the cultural switch to DLM rapidly. Most profoundly, because it acknowledges the extent of the contribution of governance to successful delivery, it makes the governance process more accountable for its deliverables in the same way as delivery and operations. The cultural switch aims to move the participating teams, perhaps from the chaotic, possibly the unstandardized, through to the defined and thence, via the measured, to the optimized. By homing in on a particular aspect of delivery or operation, such as issue tracking or the delivery pipeline, and assessing its current maturity, it is easier to identify the next stage to be reached with the aim of continuous process improvement. This process can be formalized with the help of a Capability Maturity Model (CMM) that provides a framework that helps teams to home in on the areas where improvements to the processes will give the most gains in effectiveness.

The deliverables of governance at the commencement of a database project

IT governance is a cross-project activity. It is responsible for the broad perspective of IT strategy, rather than the requirements of a particular application or project. This sounds rather nebulous, but it means that governance will be better placed to provide the context and history within IT as well as an understanding of the organization. When a database needs to be delivered, certain preliminaries need to be decided or determined before any effective work can be done, and governance should be well placed to tackle this. Without these preliminaries in place, the predicament of delivery is like that of a bricklayer who discovers that, not only are there no architectural plans, but there is also no idea where the services are, or how the site can be accessed.

Before delivery can commence, the governance team needs to have determined the context of the application. It should have identified the main business needs and provided an overall data architecture. It needs to have decided on how these business requirements can be met with a mix of bought-in, commissioned, and internal development work. There must be a clear idea of the business priorities for the database.

If there is to be in-house development, governance must bring their planning role to assist development teams with up-front planning for development. Governance must identify all enterprise-wide data and processes shared by applications, and plan data interfaces and feeds. It must ensure the participation of all the necessary IT expertise within the organization for security, compliance, etc. Governance must also agree a resilience plan with the operations activity in conformance with the standards defined by the organization. In a DLM-based organization, the deliverables of the governance process are often tested by the delivery test team for consistency and completeness.

Let's itemize a few of these tasks. For a small project, it is little more than a tick in a box, but within a corporate setting it may prove a major hurdle. Whatever the size of the development, though, these tasks still exist.

Identify off-the-shelf alternatives

A wise organization never develops a database-driven application if there is an adequate and cost-effective alternative available. You wouldn't want to reinvent and maintain your own payroll system for your company, would you? This may seem obvious, yet many an organization has fallen foul of this error. There may be good reasons for creating a system whose functions are duplicated by a commercial system but this is unlikely. If the reasons exist, they have to be carefully articulated and documented. Commercial organizations all share tasks that are uncannily similar and it is far better to focus IT effort on the specialized parts of the organization. At the same time, even if a database is to be created, the actual RDBMS or NoSQL product to be used needs to be selected in the light of the requirements of the application and the cost of hosting, maintenance, and support, and also whether it is likely to be on premise or Cloud-based. This will be part of the technical architecture that is developed in collaboration with the delivery team.

Data retention requirements

Due to data protection legislation, it is never the case that personal data, data about individuals, can be held forever. Depending on what is held in a database, there can be several requirements which may even conflict. I once had to deliver an application where the national government required data to be retained for five years in case it was needed as evidence in law. At the same time, European law required that it be removed after a year. Several legislative frameworks specified a two-year retention. After some head-scratching, we decided to remove data from the database after a year into a ring-fenced archive that could be accessed only with a written directive from IT management.

This activity is slightly detached from compliance, because by no means all decisions about retentions have a basis in the legislative framework, but could just as easily be business decisions or IT policy.

Data classification for access requirements

In most organizations, there are rules to ensure data integrity and security that determine which members of the organization, in which role, can access, update, or delete various classes of data. This process of classifying data was carried over from the manual systems,

but is still important, and each class of data will have its organization-wide business rules. Organizations generally adopt a legislative framework that makes sure that the way that you manage data integrity complies with the current legislation and industry practices. Data-retention practices, data access, security, auditing, and resilience all have to meet a legal standard. To make this simpler to manage, the data is classified into groups according to the restrictions put upon it. The most obvious types of data that require special access control and security regimes are personal data, financial data, and corporate data. Any data that is used by the application should be checked against this classification to ensure that it is stored and handled appropriately. Not even a startup with a simple data model is immune from this.

Master data identification – the canonical source of data

All data is "owned," and much of the data within an application will be pre-existing, owned by another agency. In other words, it comes from an external source beyond the application. Even the meanest database is likely to have static data, such as a list of languages that comes from a published source and is maintained by them. You are unlikely to want to maintain a list like this yourself: you rely on the owner. As database applications get more complex, subtle problems can emerge if you are not clear about where the master version of all the different data that is copied into a database is held. This data can only be effectively altered or deleted in the master or canonical source; (also known as the "publisher"). Any ETL process can become extraordinarily arcane if this is not clear and well documented.

Service-level requirements

Any organization will have a clear idea about how long it can afford to be without its IT services. Untoward disasters are a different topic to predictable down-time, and service level defines the amount of software and hardware effort that should go into designing systems with a minimal down-time. The architecture of an application will be quite different if it needs to be highly resilient, meaning that it is designed to avoid any possibility of down-time. Systems for the aviation industry where failure would lead to loss of life will be very different in nature to a software game, for example, and the test regime within delivery will be far more complex. The service level must therefore be determined before delivery can begin, since resilience cannot be retro-fitted into an existing system.

Compliance with legislative frameworks

Developers seldom wish to know the details or reasons for the need for regulatory compliance. The governance process must translate the appropriate frameworks into a simple "cookbook" in advance of delivery by governance. Every company should have, and adopt, a compliance framework that is detailed enough to allow the delivery team to operate a series of checklists to make sure that all the work is compliant and no redesign is ever necessary. A database that clearly defines the scope for sensitive data within an application is very different in design from one where data is not subject to legislation. Few legal requirements require explicit documentation, but compliance is a business issue that has to take account of trade practices, union agreements, national law, international law, insurance conditions, and a variety of other factors. All this has to be distilled into simple framework instructions to make compliance easy and effective.

Determining data consumers and providers

Databases are a great way of making data accessible across applications, but there is a limit to the extent to which any type of database can become a generic data server. This is a more specialized role of a data warehouse. More commonly, a database will publish data which is then used to update subscribers. ETL processes are more generally used for this purpose, although in a server environment which is a homogeneous SQL Server, replication can be used. It is easier for the delivery teams, who have to build such a system and operations who have to maintain it, if this system can be planned up front without the need for many ad hoc additions. Where data is served via OData, a Rest-API or other HTTP-based protocol, the resilience, maintenance and operational considerations need careful planning. A retrospective request close to project completion is seldom welcomed by the delivery team.

Disaster recovery planning

This is a topic where governance, delivery and operations must work closely together. Disaster recovery is related to service level, and also relates to the speed at which a service can be restored after an unexpected disaster, the so-called "Act of God." Whether the floods, earthquakes, lightning strikes, man-made disasters, or acts of war can be ascribed to an angry deity or not, it is likely that the angels are on the side of the IT people who plan and imple-

ment carefully-thought-out disaster recovery plans. Any organization must have a Business Continuity Plan (BCP), which is much broader in scope than the IT applications and data. Such plans deal with key personnel, facilities, crisis communication, and reputation management. The IT Disaster Recovery Plan is usually held by senior management as part of a BCP, and a part of this is the Data Recovery Plan (DRP). Individual applications and databases will have different requirements, depending on the nature of the data and the Service Level Agreement (SLA), and the applications must be designed in line with the DRP. Even small, apparently minor databases must have an effective DRP. A failure to do so can, for example, invalidate an insurance claim.

Data architecture

All organizations must have, and maintain, a data model that provides a birds-eye view of the data, data-flow, and processes within the organization. In the early days of commercial IT, this was often referred to as the "computer manual." The maintenance of such a model is an implicit part of any IT department's mission. Without it, survival of the organization would be difficult and, if legal problems ensue, ignorance of organizational data is no excuse in law. It has become a mistaken dogma that a business domain has its own hermetically-sealed edges, and arcane rules and language, but in reality every organization knows in broad terms the nature of its data, the location of the data, and the vocabulary of the data. It will be aware of the processes that are applied to the data. It is certainly possible that the enthusiastic stakeholders for a particular application will have little idea, but that is another matter. Governance must tackle the task of determining exactly how any new application fits into this overall data architecture. By doing so, governance can provide the delivery team with a very reasonable idea of the scope of the data, sufficient to refine the cost-estimate for the project.

Preliminary scope

It would seem obvious that a delivery team must be aware of the scope of the project that they are engaged with. It is a source of some bewilderment to the seasoned developer to discover a project being started before its scope has been specified, agreed, and signed off, but it happens. The definition of scope does not have to be precise at this stage but it must be sufficient to allow cost-estimates to be reasonably accurate and timescales to be close to reality.

Up-front planning and costing

With the input of the data architecture and the preliminary scope, together with service-level requirements and the data-retention plan, it is easier to calculate a reasonable estimate of costs and timescales. These estimates will change according to the size of the teams involved, particularly developers and testers. This planning and costing exercise is most easily presented as a standard project-management model that can come up with calculations based on the staffing levels of the project, the dependencies, and slippage in the completion date of any of the deliverables on which the development team depends, as itemized already in this chapter.

The deliverables of operations at the commencement of a database project

Although some of the more traditional of IT managers will recoil in horror at the idea of operations having a voice in project initiation, the truth is that a DevOps culture of collaboration and consideration that stems from ALM best practice has enormous value in ensuring the health of a database application throughout its active life. Databases and database-driven applications must be designed for production. Operational concerns must be factored into the design, and the experience of maintaining and supporting applications has enormous value in getting database applications right, and economical to run.

Aside from the DevOps initiative, operations must work closely with Governance to make sure that the hosted production database meets DR plans and High Availability SLAs. Beyond this, operational teams are the obvious source of detailed plans for hardware platforms, along with server and network configurations. At the same time, operations would need to work with both the delivery team and governance to prepare plans of how to host the pilot version of the database. Although governance is responsible for providing the deliverables documentation (which I've itemized) to the delivery team(s), it will be with the advice and input from operations on the requirements of maintenance, monitoring, support and training. At this stage, it will not be in great detail, but sufficient to give accurate timescales and costs.

Operations will need to work together with governance to establish the likely demands for scaling and evaluate alternatives for achieving this. They will also need to establish a CMS archive for the project, based on a VCS.

Training and user documentation requirement

For any application that requires staff or members of an organization to work with it, there will be a training requirement of some sort. Training materials and courses need to be developed, and delivered in good time. This is usually given to operations as a responsibility, and has to be planned and costed. It can, in some circumstances, turn out to be a considerable expense, on the same scale of cost as the actual development. At this stage, the plans for the delivery of training needs to be prepared in sufficient detail to understand the effort and timescales involved. Because of the level of detail required for training materials, it is unlikely that this can commence long before release, but if the tech authors work closely with user experience (UX) teams and developers, a great deal of time can be saved with this.

Support requirements

The users of a system are likely to require a measure of support, usually from a helpdesk that already provides support for a number of other systems. Most applications spend the majority of their life without being provided with any support from the delivery teams. This is done instead by operational support / helpdesk staff. Their requirements for this task are seldom met adequately, even though they are easy to provide if they are known early on in delivery. The delivery team may be able to exploit a good liaison between development and operations to choose to create a single monitoring console for both development and long-term support that can sort out all common user problems.

Support is tiered to distinguish first-level support from support that must be escalated. This escalation procedure needs to be worked out in advance of the system going live.

Maintenance requirements

A database can spend most of its operational life without the close attention of a delivery team and will rely on maintenance by operational staff. For a system that is highly available, this will mean that as little as possible should be assumed of the technical knowledge of the operational staff fixing a problem. Therefore, as many of the common maintenance tasks as possible must be automated, and all of them will need to be provided as a step-by-step guide. Maintenance teams are also often provided with a manual with all the logged warnings, errors and exceptions along with an explanation of the likely cause and amelioration. (E.g. warnings of inadequate disk space and the necessary fix!).

Escalation procedures

Both support and maintenance activities require documented escalation procedures for service requests, incidents and logged issues. These need to define what role needs to be informed of events based on severity, (e.g. production critical or production down, as compared with a usage issue with a work-around) or what role should deal with different classes of incident or support request. It also needs to define up front the chain of alerts that need to happen if nobody of a particular role (such as support DBA) has responded. It should define acceptable response times, and all service procedures.

Issue reporting and tracking

Automated issue-tracking systems are vital to production, and are extremely useful in delivery as well. If the issue-tracking mechanism is devised early on in the life of the system, with all requirements for operations, delivery, and tests accommodated by the system, it is likely to save a great deal of time. The database system should be able to feed issues into the issue-tracking software as well as to the participating teams. The sort of issues would include all database events that should result in an alert, such as overrunning scheduled processes, SQL batch compile errors, long-running queries, and deadlocks. This requires a great deal of care to prevent a duplication of issues flooding the issue-tracking system.

What the delivery team must provide at the commencement of a database project

The delivery team of any size will have, or will be able to call on, specialized expertise in UX and design, test, technical authoring, and technical architecture. Larger teams may be able to call on expertise in configuration management. This is likely to be shared across projects, and it takes skill to make sure that the right expertise is available at the right time.

Create the delivery pipeline

Though it is a pleasant thought that developers should be able to lean back in their chairs and yell: "Let's ship it!" the reality is a lot more complex. The deployment of a major application is like launching a ship, and the delivery pipeline is like the slipway.

Even a rudimentary trading website needs a deployment pipeline. At the other end of the scale, a database-driven application whose failure can cause loss of life, or financial disaster, requires many checks, release-gates, and divisions of responsibility. Whatever the scale, it is usually best to create and refine the process well before it is required. This will involve the solemn delivery of the "Hello World" application through all test regimes, sign-offs, release-gates, approvals, through staging and delivery to production. Once the process is determined and agreed with all the participants, it must be automated to make the process visible to all participants, and make it measurable. It must be flexible enough to allow improvements and amendments.

Delivery will take the lead in this but in cooperation with operations and governance. All three activities will have an essential role in providing an effective delivery pipeline.

Refine the technical architecture

Any initial technical architecture will be relatively short on detail, but should, if done properly, provide a good framework for discussion and elaboration, and have more of an accent on the requirements rather than the technical solutions. It is likely that the technical architecture will develop in many ways with the input of delivery and operations. The test team within delivery will also have an input into the technical architecture, particularly where they are closely concerned with soak testing, performance testing, limit-testing or scalability. Some test teams are extremely skilled in spotting where information is missing or there is logical inconsistency in the specification.

Select a toolset

The choice of toolset is nowadays more open and less inevitable than before. ALM as a development initiative was wrecked by the software publishers who convinced the market that you needed a single unified toolset across the whole application lifecycle from one

provider. It became a futile race for the most comprehensive IDE because developers have a cultural distaste for being boxed-in by a single supplier. The rise in the ecosystem of tools that were best-in-class but were able to communicate with each other has since changed the way that developers work. Instead of staying within a single IDE, many teams will now happily mix such diverse tools as Atom, JIRA, Bugzilla, TeamCity, Git, DocuWiki, Slack, Gitter, SQL Server Management Studio (SSMS), SQL Compare, Mantis, as well as Visual Studio, IntelliJ IDEA or Eclipse. There is much more freedom of choice of tool to assist in the database lifecycle.

Although there is less of a need to decide on a final selection of tools before development starts, it is useful to have a reasoned plan in place, and reserve time to check out the market for niche development tools in the light of experience with the project.

With DLM, the big difference is that the toolset must accommodate the requirements of all the participating teams, rather than just the developers within the delivery team. There are obvious advantages, for example, if the same issue-tracking tool can be used for both development and operations, and that scripts for maintenance jobs that will be used in production, use the same tools as the development tools. In the delivery pipeline there is a particular need for effective communication between participants and this will require special care.

Provide the roadmap for iterative development

DLM aims to be effective with any development methodology but many of its aims are more easily achieved if development proceeds step-wise, with clear objectives for each step. Delivery is achieved iteratively in a series of increments or steps, each of which delivers new functionality. This would provide a clear progress toward the system's final functionality. These iterations should be short and, if they are of regular size, will help to understand the progress of the delivery team, and hopefully successively refine the estimates of the remaining work. Each iteration should adhere to a similar pattern and should be able to demonstrate objective progress but should not be extended merely for this purpose.

At the start of development, the team needs to be clear on the processes, systems, and vocabulary that underlie an iterative development. Generally speaking, an iterative development will proceed in a regular way that allows for continuous process improvement, feedback, and monitoring of progress.

Each iteration is fleshed out at the start by means of a planning meeting with the delivery team and governance. The objectives and tasks for the iteration need to be planned by all

participants to ensure that there is agreement on what is expected in terms of delivery at the end of an iteration, and why. Governance will be particularly valuable in helping to assess the repercussions of any new work item on the current iteration. If new work cannot be delayed until the next iteration, the priority of added scope has to be negotiated carefully by all parties, so as to accommodate it against existing work items.

This meeting needs to define the objectives in terms of work items and their component tasks. The priority of individual tasks should be determined. Each work item needs to involve completion, right through to potential delivery, including all unit testing. Individual team members will sign up for tasks and estimate how long they will take. Once this has been agreed, then the iteration should be allowed to proceed with as few changes to this plan as possible.

During each iteration, the delivery team needs to share details of progress on active tasks, and address promptly any slippage. The team will need to gage progress and detect issues in order to cooperate on a solution in a way that encourages team members to actively seek advice, specialist expertise, and help. Although regular meetings are useful for this if they save time, they can be unwieldy in a DLM/ALM setting because of the requirement for governance and operations to be able to see progress and status via an iteration plan that logs progress.

At the end of the iteration, any incomplete work items are returned to the work items list, and usually included in the subsequent iteration. The team will need to assess whether the requirements that have been flagged as complete have passed test cases. Each iteration will probably include a review and retrospective, so as to check for opportunities to improve the process and flag up any issues that could affect delivery.

Iterations should result in the delivery of fully-tested software, that can be demonstrated to the users and stakeholders, and possibly released.

Support ETL, reporting systems and other services

The development work on database systems can quickly become gridlocked unless dependencies between applications and their databases are reduced to the minimum required, and contained within an interface. Although, in the popular imagination, there is an idea of a simple one-for-one relationship between a database and its application, that can be kept in step by keeping them together in a single source-control system, this model breaks down rapidly with increasing complexity. It is safer to first identify all the likely "customer" and "provider" applications; then we need to negotiate, with their delivery teams, a specified

interface that allows each system to evolve independently without unintentionally introducing dependencies. This work must be done at the start of delivery. If a single team is designing a new database to serve a user-application, then only one interface is required.

There are, however, likely to be reporting and warehousing systems downstream of the database application which will need to be served data in a way that is a matter for discussion between teams, but it should never be left to chance or an ETL system that grows organically like barnacles.

Identifying existing applications that could become customers or providers of data should be easy with a corporate data architecture, and particularly easy if the governance process has already determined the data consumers and providers, (see above).

Ensure that development is a managed process

DLM bases much of its promise on improving delivery processes by making it more reliable, repeatable and consistent. In general, development of a database application evolves along a spectrum that ranges from the chaotic through to the optimized and, to do this, delivery must become more standardized, clearly defined, and measured.

To do this in more detail, it would help in the planning if one takes a realistic view of all aspects of delivery and assesses where the team is in the spectrum, and what steps are required to refine the process to the next level. There are several techniques that assist with this, but are beyond the scope of this chapter.

Determine all database requirements to comply with the legislative framework

If all has gone well, a delivery team will be furnished with a relatively simple list of what is required in order to comply with the regulatory framework that the organization has adopted. The next step is to determine what data needs to be held in the database. For this, the data architecture is essential. Once this is done, then it should be possible to be sure of the data-retention requirements (see above) and the data classification for access requirements (see above). This will then allow the delivery team to map the data according to its access level and ensure that the database meets the organization's compliance framework.

How is this organized?

Who provides the governance to Governance?

Although project-based deliverables can be tracked by a project manager, there is often a problem in IT departments with deliverables such as business continuity plans or data classifications that are shared throughout many different activities. If, for example, there isn't an adequate framework for regulatory compliance, or the organization's data model is insufficient, it is difficult to correct the problem and a project can face over-runs as a consequence. Although it is a beautiful thought that one can have highly organized and process-conscious delivery teams within a department that is essentially chaotic, it simply doesn't work in practice.

A well-functioning test function can apply its testing skills to documentation, models, and frameworks just as easily as to software. Where test teams have been given the task of checking whether these deliverables are fit for purpose, experience has shown that it can be an effective way of ensuring adequate quality in what is provided for delivery.

What happens when information is missing?

So often, when a member of a delivery team is required to find out about the technical architecture, data model or production requirements, the person who is responsible will tap their head and say, "Don't worry, it is all there in my head. Just come to me and ask if there is anything that isn't clear." In fact, if something isn't documented, it doesn't exist, and any vague contents of the head is lost when the member of the organization leaves or gets moved to another department. A "model," whether about data, architecture, support, or any other organizational process, will consist of a written description, overviews, roadmaps, UML, ER modeling, SSADM or other standard diagrams, catalogs, and indexes written to a fairly standard format. Many will protest that such a document is never read. In fact, if it is useful, correct, and maintained, then it will be used. The objective is, after all, communication. The alternative is excessive duplication of effort, a breeding-ground for misunderstanding and endless re-engineering of code to correct business logic. In other words, chaos.

What are the practicalities of maintaining models, plans and documents?

DLM aims to improve delivery by improving processes and improving the expertise of the teams. Information is sometimes relevant to everyone, sometimes only for a particular application or database, and sometimes only for a team or activity. These all require different ways of maintaining a record.

Documentation must only be done for a purpose: Is it to allow software to be refactored or maintained? Does it define a contract model? Is it purely for audit purposes? Is it done to make support tasks easier to perform without calling on specialized expertise? Is it to define a technical solution for a series of applications, such as single-sign-on? Is it to define organizational policy? Is it a document that could be used in litigation? Each of these types of documentation require a very different approach. Documentation of software components and database schema, for example, is best held as close to the code as possible, whereas an organizational, technical or data model is likely to be an Intranet-based document and a collection of diagrams.

Any document that is maintained by governance will have a purpose, a major part of which will be to make it much easier to do any software delivery. It is essential to preserve "organizational memory" effectively, but it isn't always the best way of coordinating aspects of a delivery, or of recording a particular system. It must stick to facts rather than speculation and is best developed through evolution, with regular revisions as understanding of the subject-matter is refined with feedback. It must be trusted by those who use it. It should be as concise as possible, and should be regularly tested rather than relying on custom or flawed memory.

The task of maintaining an organizational model has to be costed so that the organization can be aware of the scale of investment. The creation and maintenance of the model must be explicitly agreed and signed off by management.

All shared documentation that is used by an organization should be in version control for the simple reason of being able to prove what was known at any particular time, in much the same way that official typed documentation was always date-stamped.

Wikis also offer a measure of audit, and specialized Wikis for documentation can provide a very effective way of maintaining documentation, particularly if they allow discussion and comments.

Documentation that is directly concerned with the code, build, and configuration of a database application should be sourced in the same version-control system as the code itself as long as it has the same level of access rights as the code.

Opportunities and problems with automation

In a DLM project where the DevOps culture prevails, there is likely to be a great deal more scripting, using shell scripts, command batches, or PowerShell. This will involve not only aspects of the application itself such as ETL, but also the configuration of networks, Cloud assets and servers. It is likely to be intensively used to automate testing and deployment.

These scripts will need the same disciplines as the source code. They will need to be in version control. However, it is not always appropriate for them to be lodged with the application source. On one end of the scale, access to the scripts must be restricted if they contain logins or credentials, or if the access to sensitive information is made easier for an unauthorized user by scanning the scripts. Also, it makes it easier to regulate access to these materials if they are stored in role-related archives such as a configuration management archive, or a support archive.

Scripts are best signed, so that only the user with the right authorization can run them. PowerShell scripts, for example have great potential power and there are several safeguards to prevent PowerShell being a security black hole. Without a restrictive access policy and other security in place, automation cannot be implemented without security risks. It is best to plan in advance to make the burden of security and access control manageable.

Summary

To provide database-driven information systems to organizations of any size and complexity, delivery teams and operations must have high-quality information when they need it, where they need it. If any delivery of a database is to be effective there must be careful planning and analysis based on process modeling, analysis of options and a good broad understanding of the nature, source, and constraints of the data being used.

Iterative development works best when the right information and assistance is provided for the designers, testers, and developers of the delivery team, and so continuous delivery of database functionality cannot happen merely from individual heroics from developers with

mutant-like special intellectual powers. The magic can only happen with close teamwork that can work in a repeatable way, that continually refines its delivery processes and organizes the work into discrete deliverables. Such a team can only work effectively if the broad sweep of the business domain is well understood, so that nothing has to be redone due to a misunderstanding of the requirements. There must be no need for heroics or pizzas in the night.

4 – Managing Data as Part of DLM

It is the data that sets DLM apart from the mainstream of application delivery. Data entities, and the way that organizations understand and deal with them, have their own lifespan. If we neglect the management of data, we risk disaster for the organizations that use it. If we take data management seriously, databases become a lot easier.

At the heart of DLM is the management of the data itself, and this is, of necessity, a continuous task in any organization. Like several DLM activities, it entails responsibilities across all the individual applications, databases, and reporting systems that use the data.

Many organizations struggle to implement and maintain proper strategies for managing data, through the various stages of its lifecycle, across all projects. Failure can lead to data that is inconsistent and hard to analyze reliably, possibly due to there being several inconsistent copies of that data in various parts of an organization. It can also lead to data being unavailable when it's required, being used illegally, or falling into unauthorized hands. In all these cases, costs can spiral dramatically. If a company hasn't got a complete grasp of the data and the way it is processed, mergers and take-overs become particularly difficult. Also, an organization will have no accurate way of knowing how much it is going to cost to implement and maintain those data systems that it will need, in order to support and expand its business.

How then, do we start to put in place some formal structures and strategies for managing business data so that an organization can ensure the availability, security, and integrity of that data, at all times, so that all relevant roles within an organization consistently understand their data responsibilities?

One useful practical strategy is to tie specific data planning and management requirements into the established phases of DLM. This chapter focuses on the planning that is required in the early phases of a new database project in order to ensure the successful management of the data, as part of the DLM activity. Although data management planning is considered to be mainly part of the governance process, it is also an essential aspect of both delivery and operations, and this chapter explains the main responsibilities of each of these roles.

The chapter will also briefly review the ongoing data responsibilities of each IT role for databases in active production use, as well as for "legacy" database systems, when the data has reached the limits of its useful life and the team need to deal with secure archiving or disposal.

What's the payback for data planning and management?

Unless an organization plans properly for data, including its capture, storage, analysis, transport, security, availability and so on, it has no way of knowing how much the management of the data is going to cost the organization, relative to the perceived business value of that data. By proper data planning, it will understand more fully the type and volume of data that it will need to manage, the type of storage systems that will be required and the sort of security measures that need to be in place. Also, it will develop a much fuller understanding of the required data-flow between applications, and the reporting and analytical requirements that will help it derive maximum business value from that data. If a disaster strikes, in the form of a security breach or application failure, the impact is minimized if prior planning and data documentation are in place.

What is the payback for operations staff? They will know what sort of backup, recovery, availability and security plans they need to have in place, for each class of data stored. They will know what data needs off-site backup, after what period data can be archived to "Tier 2" storage, and so on, and they can plan for that. Sensitive data may need to be encrypted, both at rest and when sent across the network; again this needs up-front planning. Support should be much easier if the team that gives support have a map of the flow of data through the organization. It means that they can, if necessary, ascertain immediately the consequences of a failure of any point in the data-flow network.

What is the payback for developers? The biggest one is that there will be far fewer nasty surprises at deployment time, such as having to perform massive rewrites because their chosen approach to security fails to comply with legal requirements. However, it will also mean that a lot of the work of understanding the application's business domain will be unnecessary, as much of the groundwork will already have been completed. There will be more consistent naming conventions, too, and it is quite usual to find that much of the data model has already been done. It will be easier to determine what applications will require data feeds, and the likely formats required. The same is true of the sources of some of the data which an application requires.

Managing data through the data lifecycle

Not only must there be a common, managed understanding of the business entities and the data that is associated with them, but business data itself must be managed throughout its useful life, from its initial creation to its eventual retirement. This becomes particularly important where data is used by more than one application because this is generally seen to be the responsibility of the "owning" application up to that point.

"Corporate" data usually transcends the life of the individual applications and databases that use it. This means that it is a business-level responsibility to make a clear statement about the requirements for the availability, integrity, and confidentiality of all the accumulated data and information used, as well as how it is stored and preserved throughout its life.

The management of this so-called "corporate" data is designed to encourage the using, improving, monitoring, maintaining, assessing, managing, and protecting of data throughout the organization and for the entire life of the data.

The lifecycle of the data, by which I mean the information itself, covers:

- **Creation and receipt** – the various means by which data is generated and enters the system, such as internal data entry via a CRM application, or from an external source.
- **Distribution** – who, internally or externally, needs to see the data.
- **Use** – the various business, analytical, and reporting uses of the data.
- **Maintenance** – ongoing support for the availability, security, recoverability, and integrity of the data.
- **Disposal** – strategies for archiving, and ultimately disposing of, data that has served its immediate business purpose.

There are other categorization schemes for information but the Information Lifecycle Management (ILM) scheme is the simplest.

Managing data within DLM

The maintenance of a corporate data model and the management of the information lifecycle are best subsumed under the DLM activity within any organization, purely because there is no logical point of separation. However, the governance activity of DLM is responsible almost entirely for ensuring that the key activities of the information lifecycle are initiated at the right time. Personal data, for example, must be disposed of within a set date that varies between legislative areas. It is illegal to omit to do so, and companies can be prosecuted as a consequence.

William Brewer described each of the six stages of the database lifecycle (New, Emerging, Mainstream, Contained, Sunset, Prohibited) in Chapter 1, *What is DLM?*, but here we're not going to break it down in such fine-grained fashion. Instead, we'll take a high-level view of the various data responsibilities of governance, delivery, and operations, through the following broad phases of the "life" of a database:

- **New databases** – cover the phases from initial conception, through the application/database design and development, up to the point it first moves to a production system, in pilot form. During this part of the lifecycle, there will often be rapid rounds of database development and upgrades.
- **Active databases** – cover the phases from the first production deployment, in pilot form, through to general production use, where data is being actively created, distributed, used, and maintained. These databases are subject to frequent bug fixes, upgrades, new feature deployments, etc.
- **Legacy databases** – covering databases in the final stages of the lifecycle, from databases in production only for limited or legal use, with no active development support, through to those scheduled for retirement, and on to the final nail in the database coffin, where its further use is prohibited and it needs to be retired.

A well-planned, proactive approach to data management, through each phase of DLM, will ensure that an organization has the data management processes in place to guarantee that data is always valid, available, consistent and secure.

One of the essential goals of DLM is to systematize or refine team processes such as database version control (Chapter 5), build (Chapter 6), continuous integration (Chapter 9), and automated deployments (Chapter 10), such that these tasks can be performed effectively without reliance on tribal memory or one individual's knowledge. It formalizes those processes so that they become consistent, repeatable, and reliable across teams.

Likewise, as an integral part of DLM, we need to use data management processes that allow any IT role to understand their responsibilities with regard to the data, across all projects, and for all applications and databases that capture, store, manage, use, or analyze that data, throughout its "natural lifespan."

Data responsibilities of governance, delivery, and operations

In Chapter 3, *Planning for a Successful Database Lifecycle*, William Brewer explained why a successful application "depends on close teamwork between the three key activities, governance, delivery and operations." Broadly and briefly, the responsibilities of the relevant IT team for each of these activities, through the database lifecycle, are as follows:

- **Governance** – covers planning and project management, in order to ensure that the application continues to provide what the business needs, conforms with the requirements of any regulatory frameworks, and meets organizational standards and SLAs for security, availability, recovery time, and data retention, to name just a few.
- **Delivery** – implementing all application design, development, release, and deployment processes, as well as ensuring the timely delivery, to the customer, of an application that meets all agreed business requirements.
- **Operations** – implementing systems and procedures for security architecture, access control, routine maintenance (bugs, hotfixes, or improvements), support, monitoring, resilience, and more.

Sometimes the application or database lifecycle is a close match for the data lifecycle; in other words, the data has little ongoing value or use beyond the life of the application that produces it. In such cases, it can seem as if the delivery and operations teams bear the brunt of the responsibility for the data management.

However, in other cases, data is scarily permanent. Banks, for example, will have a complete history of its customers' data, even if they have used several applications to host it. As noted in the introduction to this chapter, this is why data management is often considered more akin to a business operation than an IT function. Much of the data lifecycle of long-lasting data is maintained by governance processes; delivery teams will only have a transient effect on, or interest in, these processes.

Data responsibilities when planning new databases

Chapter 3, *Planning for a Successful Database Lifecycle*, summarizes many of the broader database planning activities that are required at the very start of a database project, so here we focus exclusively on data responsibilities of governance, delivery, and operations relating directly to the security, compliance, retention, resilience, and availability of that data.

Governance data responsibilities when planning new databases

There is a lot of planning work that needs be done at the very start of a new database project, primarily by governance, but in close collaboration with the delivery and operations teams. Some of the steps necessary are:

- Identify the main business needs and therefore set business priorities for the data.
- Provide an overall data architecture.
- Identify all enterprise-wide data and processes shared by applications, and plan data interfaces and feeds.
- Ensure the participation of all the necessary IT expertise within the organization for security, compliance, and so on.
- Agree a resilience plan that includes HA and DR in conjunction with the operations activity.

Data classification

An important responsibility of governance for a new database project is to establish a Data Classification System (DCS).

There are several relevant classifications of data, such as the type of structure, whether it is structured, tabular, hierarchical, or unstructured. It can also be about the actual metric, such as geographical, chronological, qualitative, quantitative, discrete, and continuous. However, what is most important for the data lifecycle is the type of security and access control it requires, and whether it is subject to the organization's compliance framework.

This classification affects both structured and unstructured data. The DCS will, in turn, determine who can view, or modify, the various classes of data, as well as other security questions such as which data may need to be encrypted during storage or transport, and which data may need to be stored off-site in accordance with the relevant level of security.

The main classes of data are as follows:

- **Highly sensitive company and confidential data** – where disclosure could have legal and financial consequences, such as staff and clients' personal details.
- **Sensitive company specific data** – the disclosure of which could have an adverse effect on operations, such as details of client and supplier contracts.
- **Data relating to the activities of a company** – which are not meant for public scrutiny. These could be sales figures, organizational structures, etc.
- **Public information** – which is freely available, such as price lists, contact details, or any data used for publicity and brochures.

Written guidelines and procedures for classifying data should define its classes and categories, such as financial, corporate, or personal, as well as specifying the various roles and responsibilities of employees relating to data stewardship. A classification scheme needs to be clear and easy for all employees to execute.

Having documented the DCS, the governance team need to work closely with delivery and operations, so as to formulate an appropriate plan for storing and handling each class of data. An organization will generally adopt a relevant legislative framework, describing the current standards and legal requirements in their industry for each class of data.

Data security and access requirements

The data classification work will inform subsequent security and access policies for the different classes of data, as well as decisions regarding the physical storage architecture (see below).

Data Storage Policies (DSPs) are sets of rules and procedures designed to manage and control data in an enterprise. These rules also ensure that data is used for the purpose for which it was intended, e.g. to make profits in a business, and protect the interest of customers. The policies are designed to protect sensitive data and secure it, so it is not given or released, inadvertently or otherwise, to unauthorized people, companies, or organizations. (See *Data classification*, above). In addition, the policies would cover misuse by individuals, such as malicious enhancements, or physical and digital manipulation.

DSPs can vary in how data is collected and stored, to also include a set of applications that govern and control all aspects of data. Internal as well as external data, such as web information data, should be included in the storage policies. It is becoming more important to have a DSP, because the growth in data is becoming increasingly difficult for administrators to deal with using routine maintenance tasks, e.g. backups and disaster recovery provisions.

A few selected rules for consideration in data protection while in storage are given below.

- **Authorized access and security controls** – allows only authorized people to view, change, manipulate, and update data according to their level of security clearance.
- **Audit trails and audit logs** – monitoring and checking audit trails or logs regularly to ensure access controls have not been breached.
- **Data hashing** – in order to ensure data integrity, a hash value is automatically generated, being a fixed length string of text or numbers, which represents larger volumes of data as a small number or short code. Comparing a hash value, would indicate if data has been altered in transit.
- **Data encryption** – the means of securing data, whether in transit or in the database. Encrypting sensitive or relevant data makes it less likely to fall into the wrong hands, to be misused or misappropriated.

Storage architecture

Most organizations recognize a 5-tier structure when classifying application data according to the required storage architecture. For example, a Tier 1 application relies on transactional data that is accessed frequently, every day, often by many people. The storage architecture for Tier 1 applications needs to reflect the need for that data to be constantly available and returned quickly. This structure also covers less frequently accessed or less valuable data (Tiers 2 and 3), disaster recovery architecture (Tier 4) and offline storage (Tier 5).

During data planning, it's the responsibly of governance, in collaboration with operations, to classify the application according to its storage architecture requirements, and also to define after what time period, data can be partitioned off to cheaper, lower-tier storage, what DR plan must be supported, and so on.

Physical storage location

The initial data classification work will also determine the physical storage location of the data, such as whether it can be stored locally, or in Cloud-based storage, or if it requires secure off-site storage.

- **Local storage** – The data is simply stored in a local data center or suitable holding repository. These would be local servers or other hard drive storage media, but not on individual local PCs, as these are not shared storage devices.
- **Data warehouse (DW)** – Traditionally, a data warehouse is located on the main server of the organization but recently Cloud-based storage is becoming more prevalent in some organizations.
- **Cloud storage** – This is online storage where the organization's data is located off-site, potentially on several servers and even at several locations. The Cloud, unfortunately, has yet to be widely accepted as a safe storage location, due to anxiety about the uncertainty regarding the legislative framework in the location where the data is physically held, having to rely on unknown security arrangements, and the ease with which huge quantities of data can be accidentally lost.

Data storage and transfer format

Besides policies relating to the physical storage location of data, governance needs to determine in which format the data will arrive, and the formats to be used for storage, transference between systems, and presentation.

By developing a proper understanding of the nature of the data involved, they can plan the correct type of database in which to store the data. This makes it possible to use graph databases, document databases, and other specialized database systems for unstructured or semi-structured data, relational databases for transaction processing, and OLAP/Cubes for business intelligence. In other words, they can correctly define the requirements of a heterogeneous data platform, if necessary, rather than relying on a single platform. This can make the task of planning and costing this sort of system much easier.

During data planning, governance also needs to establish the basic storage format for each type of data, in terms of how they are stored (data type, unit of measure), transferred, and represented. Some data, such as times, are only relevant for a particular time zone, money only has meaning when associated with a currency. If storing details for automobiles, it needs

to be decided if this will be stored in KPH or MPH. The planning process will need to specify the checks and reconciliation processes that are required to make sure that any data corruption is detected quickly.

The big problems of data storage arise when there is no planning or understanding of the representational version of data, the transfer version, and the machine, or binary, version. How should the system handle dates, for example? They should be stored in a format that a machine can easily understand and on which it can rapidly perform simple calculations. Essentially, the data is stored as a number representing a quantity relative to a known point in time, such as an associated time zone. However, the date would be transferred in an ISO standard format, and be represented in a way that anyone can understand.

Data quality

By ensuring data quality, an organization can trust its data, so that it can make intelligent decisions to improve its performance, based on that data.

There must be a common and consistent data language for everyone in an organization. This will contribute greatly to overall data quality. For example, how does the organization define and identify a "customer?" The answers to such questions are not necessarily straightforward, and certainly the organization needs to avoid situations, where one part of the business considers a customer to be an "account," and another an "individual person."

Data planning for a new database must establish a single, unified definition of each entity about which information must be stored and the origin, or "canonical source," of the data for each attribute of that entity; it must also stipulate a consistent way to link all entity data.

Data will originate from various internal sources, processes, machines, instruments, and sensors. Disciplines such as Master Data Management will help join together all of the company's information and data to a coherent whole. It helps prepare the data, thereby ensuring that the organization has a consistent, accurate overview of all information and, in addition, can depend on the business decisions taken, based on the analysis of that data.

During planning, governance should provide a set of high-level requirements for effective control of data quality within each IT process, and agree on objectives and design procedures to measure the data quality. This may entail use of commonly-referenced best practices and guidelines, such as COBIT, ISO/IEC 38500, the methodology of Six Sigma, and various other tools for data mapping, profiling, cleansing, and monitoring data.

Auditing and compliance

Every country has different rules, directives and laws governing the security, storage and handling of corporate, financial and personal data. Regulatory compliance outlines the aims organizations should attempt to satisfy, to ensure they are fully aware of relevant laws and regulations, and that they take the necessary actions and employ data management methods, to comply with them.

The International Organization for Standardization (ISO) produces guidelines designed to provide a common framework of how compliance and risk should operate together. In addition, a lot of regulations come from European Union (EU) legislation. Various areas are controlled by a number of different bodies, including the Financial Conduct Authority (FCA), and the Information Commissioner's Office.

The governance process must translate the appropriate compliance framework, governing storage and handling of sensitive data, into a simple set of checklists that delivery can use to ensure that all the work is compliant.

The compliance rules for handling "sensitive" data will also feed into the plans for implementing auditing requirements needed to ensure (and prove) that data has not been accessed or modified by unauthorized people or processes. An audit can range from a full-scale analysis of business systems and methods to monitoring system or administration log files which record changes and amendments.

Data retention

Complying with legal requirements of data retention is difficult and complicated. Data retention laws sometimes require data owners and service providers to keep large quantities of user data for a much longer time than their business operations require, or paradoxically sometimes demand its deletion while actively being used.

Compliance laws like the CAN-SPAM Act (http://preview.tinyurl.com/krdrl9y) and the Fair Credit Reporting Act (http://preview.tinyurl.com/guyjcxp) in the U.S. demand that organizations confer to an individual the "right to be forgotten." However, the latest laws require longer data-retention time, even though it is opposed to the individual's wishes. The result raises privacy rights which are a real legal challenge to sort out. Generally, organizations will adopt a "framework" that interprets the entire range of legislation and specifies in practical terms what is appropriate. This means that a compliance expert need only check actual practices against the framework.

Data resilience

Data resilience is the means that allows continued, uninterrupted operation, despite problems such as equipment and power failures, or any other faults or interruptions. Resiliency will ensure that the system maintains its operational ability.

Data resilience forms part of an enterprise's data architecture. It is also often incorporated into disaster-planning and disaster-recovery considerations, which are closely linked to data protection. The task of governance is to make sure that actual operational practice conforms to these documented expectations.

Delivery responsibilities when planning new database projects

In planning a database application, the delivery activity will work with governance to determine what data is entirely domain-specific, and what data is already part of the organization's existing data model. They will need to establish in broad terms what the requirements of the application will be for existing data. This will determine several requirements that will affect the development effort, especially where there are special security, logging, and reporting requirements. The delivery team plans how they will best meet these requirements.

Furthermore, in conjunction with both governance and operations, they plan a structure within the organization that supports service-oriented architecture (SOA) or requirements for specific architectural changes which support the application.

Operations responsibilities when planning new database projects

During the planning phase, the operations team will act in an advisory capacity to both governance and delivery. They will address HA, DR, security and architectural planning, as well as costing.

In planning the disaster recovery strategy, governance needs to:

- understand what is at risk
- know what data has been stored
- evaluate vulnerable points objectively
- assess all the storage and the disaster recovery environment
- perform tests to ensure that data and environments are secure
- run internal security assessments to highlight environmental vulnerabilities.

As different applications and systems require differing levels of protection, protection tiers, and availability, these areas need to be objective, analyzed and prioritized. The economic and operational advantages should be balanced with the requirement of adequately protecting data and applications.

Ongoing data responsibilities for active databases

Once a database is in production, either in pilot form or in general use, governance, operations, and delivery must cooperate to ensure that data management processes are implemented as agreed, and are then monitored, improved, and adapted where necessary, so that all databases continue to meet the requirements for data availability, quality, and security.

Governance, broadly, will need to ensure that all information management processes are implemented as per the specifications, and amended in the light of changes in the business (such as after an acquisition), or changes to the application or data (e.g. after a new deployment).

They will perform checks on data entry mechanisms, data quality, access control, auditing and logging, change-control processes, and more. They will need to work with the delivery team to ensure that all data handling, security, logging, and reporting mechanisms in production databases continue to comply with any updated compliance framework.

Operations will have data responsibilities relating to correcting reported data quality issues, implementing the agreed backup and data protection plan, managing data growth, managing the storage architecture including migrating data between tiers as required, and performing regular performance checks for compliance.

Ongoing data responsibilities for legacy databases

Toward the end of a database's lifecycle, it may be in "contained" production use, to support specific, limited business functionality. Governance will continue to monitor for changes that are required as a result of security concerns or legislative changes, but any activity of the delivery team will be limited to changes that spring from these concerns. All ongoing operations such as maintenance and monitoring will be done as a routine, relying on documentation and a central management server archive.

If there are plans to retire ("sunset") a legacy database, there is additional work to be done in ensuring that any replacement system can provide equivalent functionality, implementing plans for migrating any required data over to the replacement database, and planning for the enforcement of appropriate data archive and purge procedures for the legacy system.

Archiving involves copying or transferring the data to a secure holding environment, where it is stored in case it is required at a later date or time. This data storage is usually for a lengthy period of time, such as in legal situations, where the law requires retention of data or documents for a stipulated time period, e.g. 5 or even 10 years.

Purging or disposal is the final exit or death of the value of the data. Its use has become redundant and so, generally after the archiving stage, it is purged from the system.

Governance will need to:

- **Plan for the termination of the service**, checking that all users have plans in place for the eventual termination of the service.
- **Establish the data formats, retention time, and archival rule**s.
- **Establish how any archived data will be hosted**, as appropriate for the classification of that data, and which methods of access, encryption, and storage are permitted.

Once data has been purged, governance has a critical security management, regulatory, governing, and compliant role to play in this step. It has to prove that the purge phase has been completed to exact requirements and that there is no residual post-purge data. Appropriate system checks, controls, and user protocols should be in place, proving that total data elimination has effectively and successfully taken place.

Operations will be responsible for costing and implementing the data archiving and purging plan, as established by governance. For the duration of the data-retention period, operations must periodically check that off-site/backup storage media can still be read.

Operations will also be responsible for arranging the data-purging process, and will need to prove that the purge phase has been completed to the exact requirements set by governance and also that there is no residual post-purge data.

Summary

DLM takes a view of data right across the organization that is using it. It might be expected, that data is entirely the responsibility of the application that uses it. This has never been the case in any enterprise. Most offices rely on whole chains of applications and processes that pass data between them. This means that the creation, use, and disposal of data has to be supervised from a central point. If data needs to be changed, it must be easy to determine the original source of the data, and then amend it in such a way that it stays altered. To do this successfully requires certainty as to the original source, the ownership, and the way that data feeds into downstream systems.

If data is lost, corrupted, or stolen the losses can be enormous. The impact can be minimized in any organization that understands the nature of its data, and the processes that act on that data, especially when it keeps to a sensible set of data strategies that are shared and understood. By adopting a sensible approach to DLM and data governance, an organization can save a great deal of time and money in recovering from adversity, and will be quicker and more effective in responding to business change.

5 – Database Version Control

By placing under source control everything we need to describe any version of a database, we make it much easier to achieve consistent database builds and releases, to find out who made which changes and why, and to access all database support materials. This chapter explains how to make sure your VCS fully supports all phases of the database lifecycle, from governance, development, delivery, and through to operations.

To achieve reliable, repeatable database builds and migrations as part of DLM, we need to store the DDL code for a database in a VCS. It should be possible to reconstruct any version of a database from the scripts in the VCS, and every database build we perform, and every database change, however trivial, should start from version control.

Ideally, you will also have in the VCS a complete history of changes to individual schemas and objects, who did them, and why. Any build and version-control system must allow developers, as well as people outside the development team, to be able to see what is changing and why, without having to rely on time-wasting meetings and formal processes.

Without good version-control practices, large parts of DLM will remain difficult for your organization to achieve, so getting it right is vital. In this chapter, we explore the database versioning capabilities provided by source-control systems, considerations when choosing a one, and good practices for version control in the context of DLM.

What goes in version control?

Every application or database that we build should originate from a version in the source-control system. With most developments, there are many points in the process where a consistent working build should be available. You need to store in version control everything that is needed in order to build, test, or deploy a new database at a given version, or promote an existing database from one version to another.

The most obvious candidates for versioning are:

- individual DDL scripts for each table
- individual DDL scripts for all other schema-scoped objects such as stored procedures, functions, views, aggregates, synonyms, queues
- static data.

There are other candidates that we will mention shortly. If a VCS saves database scripts at the object level then each file corresponds to a table or routine. In this case, the task of creating a new database or upgrading an existing one from what is in source control, is primarily an exercise in creating an effective database script from the component object scripts. These will have to be executed in the correct dependency order. Subsequently, any static data must be loaded in such a way as to avoid referential constraints being triggered.

An alternative is a **migration-based** approach, which uses a series of individual **change scripts** to migrate a database progressively from one version to the next. These approaches are discussed in more detail in Chapter 7, *Database Migrations: Modifying Existing Databases*.

However they are produced, we need to also version the complete build script for the database structure and routines, as well as the data-migration scripts required to ensure preservation of existing data during table refactoring, plus associated rollback scripts. Normally, the complete build scripts would be generated automatically from the nightly integration build, after it has passed its integration tests (see Chapter 9, *Database Continuous Integration*). These integration tests will also verify that any hand-crafted migration scripts work exactly as intended, and preserve data correctly.

A common mistake that development teams make is to assume that database source code consists merely of a number of objects. In fact, the dependencies within a working database system are numerous and sometimes complex. For example, a change to one item in the database configuration, such as the database collation setting, might be enough to stop the database working as expected. Therefore, beyond the tables, routines and static/reference data, we also need consider the broader database environment, and place into version-control elements such as:

- database configuration properties
- server configuration properties
- network configuration scripts
- DDL scripts to define database users and roles, and their permissions
- database creation script
- database interface definition (stored with the application it serves)
- requirements document
- technical documentation
- ETL scripts, SSIS packages, batch files, and so on
- SQL Agent jobs.

Benefits of version control

Unless we have in the VCS the correct versions of all the scripts necessary to create our database objects, load lookup data, add security accounts, and take any other necessary actions, we have no hope of achieving reliable and repeatable database build, release, and deployment processes, nor of coordinating database upgrades with changes to the associated application. If we perform ad hoc database patching outside of a controlled VCS process, it will inevitably cause data inconsistencies and even data loss.

Provides traceability

The VCS provides a complete history of changes made to the database source. The team can see which developer is working on which particular module, which changes have been applied and when they were applied, which modules are available for release, the current state of the production system and the current state of any test systems. It also means that the team can, at any time:

- roll the database forward or back between any versions
- re-create older database versions to find when a subtle bug was introduced
- perform a code review, checking coding standards and conventions.

This traceability is crucial when diagnosing incidents in production or when responding to an internal or external audit, and is particularly powerful when using hash-based VCS tools such as Git, which trace the file contents. With the correct permissions scheme in place, a VCS provides a trail of changes to all text files used in the software system.

Provides predictability and repeatability

Keeping all text-based assets in version control means that processes are more repeatable and reliable, because they are being driven from a controlled set of traceable files rather than, for example, arbitrary files on a network share. The modern VCS is highly reliable and data loss is extremely rare, making it ideal to depend on for automation.

Protects production systems from uncontrolled change

The VCS acts as a guard against "uncontrolled" database changes, i.e. direct changes to the code, structure, or configuration of a production database. The VCS must be treated as a single source of truth for the database source, and configuration, including database schema, reference data, and server-level configuration settings.

The VCS is a mechanism to ensure that the database source that has been stored is identical to what was released. Before deploying a new database version, the team can compare the target database with the source scripts for that database version, in the VCS. If they do not describe identical databases, then the target database has drifted. In other words, there have been unplanned and untested fixes or functionality updates made directly on the live database, and the team must find out what changed, and why.

Aids communication between teams

Current version control environments offer rich, browser-based features for collaboration, communication, and sharing between teams, helping to foster interaction and engagement. Features such as pull requests, tickets or issue tracking, and commenting promote good practices such as code review and coding in the open (see: https://gds.blog.gov.uk/2012/10/12/coding-in-the-open/).

A VCS platform that makes it easy for DBAs or the operations teams to review proposed database changes, while automatically storing all the traceability information, will encourage tight feedback loops between operations and development and other stakeholders.

Choosing a VCS for DLM

A good source-control system should make it simple for people across the organization to track changes to files. Usability should be high on the wish-list for any VCS, particularly if it must be easily accessible to other teams besides developers, such as testers and database administrators, as well as governance and operations people.

Version-control tools

We need to distinguish between low-level version-control tools, usually a combination of client tool and server engine, such as Git or Subversion, and version-control platforms that provide deep integration with other tools and a rich, browser-based user experience, such as GitHub or Bitbucket Server (previously known as Stash). We cover tools in this section, and platforms in the later section, *Version-control platforms*.

Git

Git's superior workflow model together with lightweight branching, merging, and history operations make Git the best choice for most teams. Unless you have a good reason to do otherwise, you should use Git as the version-control repository part of your VCS. All modern VCS tools support Git, including Microsoft's Team Foundation Server.

Mercurial

Mercurial is similar to Git; they both use similar abstractions. Some advocate that Mercurial is more elegant and easier to use, while others claim Git is more versatile. So it comes down to a matter of personal choice. Certainly, Git is the more widely-adopted tool due to the rise of platforms like GitHub.

Subversion

Subversion is a sound choice for smaller repositories or where there is a fast network connection to the central server. Subversion can also act as a low-cost artifact repository for storing binary files.

Team Foundation Server

Team Foundation Server (TFS) 2015 and later support Git as the versioning tool. If you use an older version of TFS, you should either switch to an alternative VCS or upgrade to TFS 2015 or later, in order to take advantage of Git support. The older versions of TFS have features and quirks that significantly hamper CD and DLM; in particular, older versions of TFS require an instance of SQL Server per TFS repository, which acts as a significant driver to use only a single repository across several teams, or even an entire organization, rather than having many small repositories that are cheap and quick to create.

Centralized vs. distributed?

Some people distinguish between "centralized" and "distributed" version-control systems, but these distinctions can become quite academic, because most teams today use a definitive central repository, even when using a distributed VCS.

Distributed version-control systems (DVCS) don't require users to be connected to the central repository. Instead, they can make changes in their local copy of the repository (commit, merge, create new branches, and so on) and synchronize later. Pushing changes from the local repository and pulling other people's changes from the central repository are manual operations, dictated by the user.

In centralized systems, by contrast, the interaction is between a local working folder and the remote repository, on a central server, which manages the repository, controls versioning, and orchestrates the update and commit processes. There is no versioning on each client; the users can work offline, but there is no history saved for any of these changes, and no other user can access those changes until the user connects to the central server and commits them to the repository.

The different approaches have important implications in terms of speed of operations. In general, command execution in DVCS is considerably faster. The local repository contains all the history, up to the last synchronization point, thus searching for changes or listing an artifact's history takes little time. In a centralized system, any user with access to the central server can see the full repository history, including the last changes made by teammates or other people with access to the repository. However, it requires at least one round-trip network connection for each command.

In my experience, more teams are now opting for the speed and support for asynchronous work that DVCS offers, but it's worth remembering that with them often comes a more complex workflow model, due to their asynchronous nature. I believe that either flavor of VCS, centralized or distributed, can work well as long as it fits with the preferred workflow of the team, and provided that the right culture and accountability exists in the team.

With either system, if the team allow changes to live for too long in individual machines, it runs counter to the idea of continuous integration and will cause problems. In a DVCS, users feel safer knowing their changes are versioned locally, but won't affect others until they push them to the central repository. However, we still need to encourage good CI practices such as frequent, small commits.

Version-control platforms

Version-control platforms provide the version-control functionality but add a lot more besides to improve the user experience for devotees of both the command line and GUI, to help different teams interact, and to provide integration with other DLM tools. For example, many version control platforms offer features such as:

- first-class command-line access for experts
- helpful GUI tools for less experienced users
- browser-based code review: diffs, commenting, tracking, tickets
- an HTTP API for custom integrations and chat-bots
- powerful search via the browser
- integration with issue trackers and other tools.

We also need to consider how easy the system is to operate, particularly if we are going to run the system ourselves. An increasing number of organizations are choosing to use Cloud-hosted or SaaS providers for their VCS, due to the reduction in operational overhead and the increased richness of integration offered by SaaS tools. There is also an argument that SaaS VCS tools are more secure than self-hosted tools, since the security management in most self-hosted VCS tools is average at best.

Some popular SaaS-based VCS platforms include GitHub, Bitbucket, CodebaseHQ, and Visual Studio Online. These tools all offer Git as the version control technology, plus at least one other (Subversion, Mercurial, and so on). Other options such as Beanstalk (http://beanstalkapp.com/) may work if you have a homogeneous code environment, because they are more focused on the Linux/Mac platforms.

Self-hosted VCS solutions are generally less integrated with third-party tools, which may limit how easily they can be "wired together." Examples of self-hosted VCS platforms include Gitolite, GitLab, Bitbucket Server and Team Foundation Server.

Essential version-control practices for DLM

Version control is central to the development, testing, and release of databases, because it represents a "single source of truth" for each database. As discussed earlier, the VCS should contain everything that is needed in order to build a new database, at a given version, or update an existing database from one version to another. This may be necessary for a new deployment, for testing, or for troubleshooting (e.g. reproducing a reported bug).

In addition, several other DLM practices depend on version control. Several activities carried out by governance rely on being able to inspect the state of the database at any released version. It is sometimes necessary to determine when a change was made, by whom, and why. Also, when release-gates need sign-off, the participants can see what changes are in the release and what is affected by the change. Any audit is made much easier if the auditor can trace changes that are deployed in production all the way back to the original change in the VCS.

It's essential that the process of building and updating databases from version control is as quick and simple as possible, in order to encourage the practices of continuous integration and frequent-release cycles. This section will discuss some of the version control practices that will help.

Integrate version control with issue tracking

Version control pays dividends when it is integrated with issue tracking. This allows the developer to reference the source of defects quickly and uniquely, and thereby save a lot of debugging time. It also allows the management of the development effort to check on progress in fixing issues and identifying where in the code issues are most frequent. Developers will also appreciate being able to automate the process of reporting how bugs were fixed and when. It also allows the team to share out the issue-fixing effort more equally and to monitor progress.

Adopt a simple standard for laying out the files in version control

It is sometime useful to store database script files in version control in a format that matches or resembles a layout that will be familiar from use of GUI tools such as SSMS or TOAD.

Generally, this simply means storing each object type in a subdirectory from a base "Database" directory. For example, **Database | Tables**, **Database | Stored Procedures**, **Database | Security**, and so on. It is normal to store child objects that have no independent existence beyond the object they associated with the parent object. Constraints and indexes, for example, are best stored with the table-creation scripts.

Make frequent, non-disruptive commits

In order to maintain a high level of service of a database, we need to integrate and test all changes regularly (see Chapter 9, *Database Continuous Integration*).

To achieve this, we need to adopt practices that encourage regular commits of small units of work, in a way that is as non-disruptive as possible to the rest of the team. Any commit of a set of changes should be working and releasable. Feature toggles allow deactivating particular features but those features still need to work and pass tests before deployment. Consider a commit to be like the end of a paragraph of text: the paragraph does not simply trail off, but makes sense as a set of phrases, even if the text is not finished yet.

Adopt consistent whitespace and text layout standards

Agree and establish a workable standard for whitespace and text layout conventions across all teams that need to work with a given set of code or configuration files. Modern text editors make it simple to import whitespace settings such as the number of spaces to indent when the TAB key is pressed, or how code blocks are formatted.

This is especially important in mixed Windows/Mac/*NIX environments, where newline characters can sometimes cause problems with detecting differences. Agree on a convention and stick with it, but make it very simple for people to import the standardized settings; do not make them click through 27 different checkboxes based on information in a Wiki.

When detecting changes, a source-control system will default to simply comparing text. It will therefore see any changes of text as a change. This will include dates in headers that merely record the time of scripting, and are irrelevant. It will also include text in comments, even whitespace, which in SQL has no meaning, in contrast to an indentation-based language such as Python. If your source-control system just compares text, you need to be careful to exclude anything that can falsely indicate to a VCS that a change has been made.

Tools such as a database comparison tool will parse the text of the SQL to create a parse tree and compare these rather than the text. This not only prevents this sort of false positive but allows the user to specify exactly what is, and what isn't, a change. If this sort of tool is used to update source control, then the only changes will be the ones that the user wants. However, this needs a certain amount of care since formatting of SQL code can be lost unless it is flagged as a change.

Keep whitespace reformatting separate from meaningful changes

Even if you have a database comparison tool that will help avoid detecting "false changes," it still helps to commit whitespace and comment reformatting separately from changes to the actual code i.e. that will affect the behavior of a script.

Figure 5-1 shows the colored console output for comparing current to previous versions, after a commit that contained both whitespace reformatting, and a behavior change, making the latter (an increase in the width of the `LastName` column from 50 characters to 90) much harder to spot.

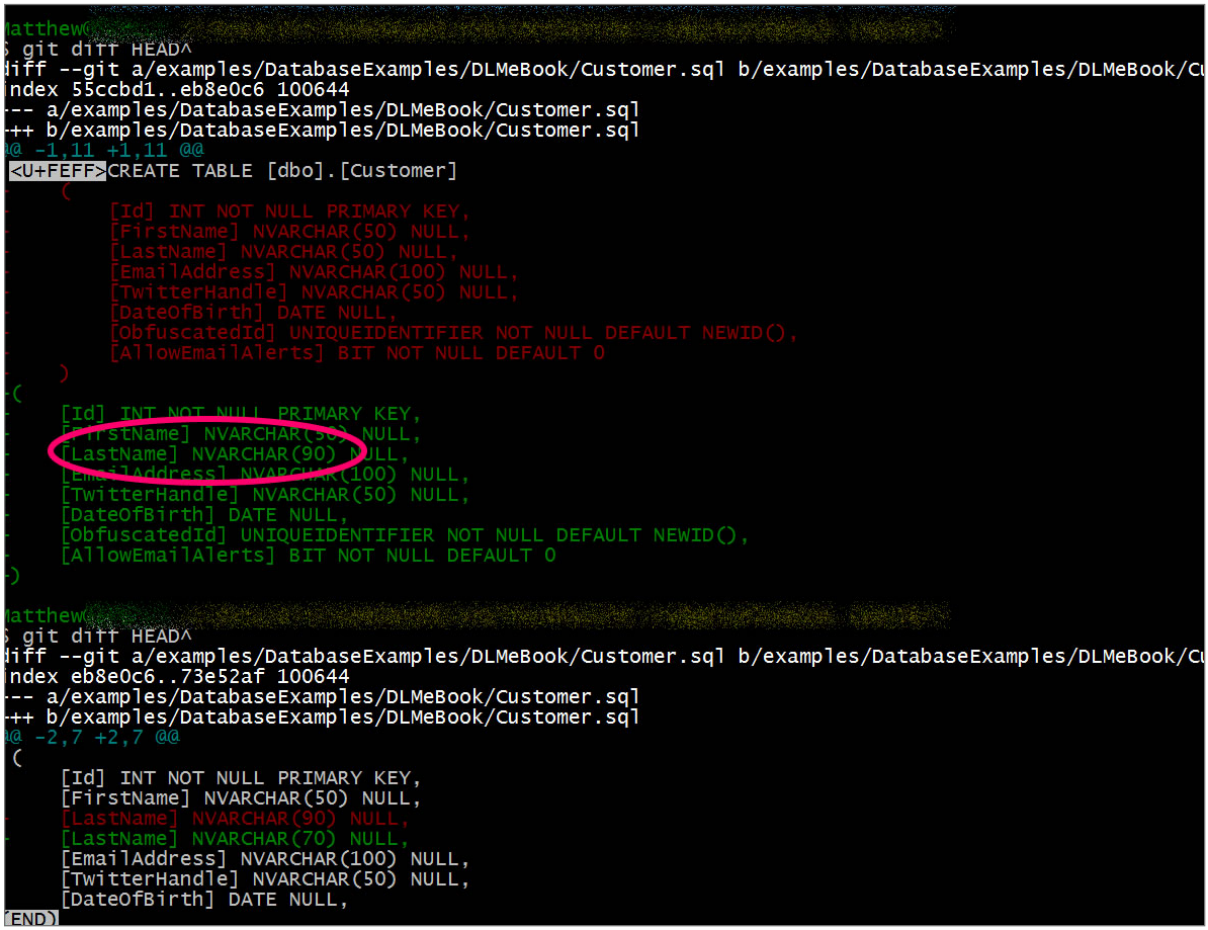

Figure 5-1: A commit that made whitespace and semantic changes.

If we separate out the whitespace and semantic changes, the colored console output, as shown in Figure 5-2, highlights the meaningful change very clearly; in this example, someone has reduced the column width from 90 to 70.

```
Matthew@
$ git diff HEAD^
diff --git a/examples/DatabaseExamples/DLMeBook/Customer.sql b/examples/DatabaseExamples/DLMeBook/Cu
index eb8e0c6..73e52af 100644
--- a/examples/DatabaseExamples/DLMeBook/Customer.sql
+++ b/examples/DatabaseExamples/DLMeBook/Customer.sql
@@ -2,7 +2,7 @@
 (
     [Id] INT NOT NULL PRIMARY KEY,
     [FirstName] NVARCHAR(50) NULL,
-    [LastName] NVARCHAR(90) NULL,
+    [LastName] NVARCHAR(70) NULL,
     [EmailAddress] NVARCHAR(100) NULL,
     [TwitterHandle] NVARCHAR(50) NULL,
     [DateOfBirth] DATE NULL,
(END)
```

Figure 5-2: Isolating a single semantic change.

Just as with any changes to files in a VCS, separating reformatting changes from semantic changes is crucial for readability and tracing problems. With a DVCS like Git, you can make multiple local commits (say two commits for whitespace and layout reformatting, followed by some semantic changes) before pushing to the central repository. This helps to encourage good practice in isolating different kinds of changes.

Plan how to coordinate application and database changes

A database can have a variety of relationships with applications, from almost total integration within the data layer of the application, to being a remote server providing data services via a protocol such as OData, the Open Data Protocol (http://preview.tinyurl.com/y9kpczrg). Commonly in enterprises, a single database will provide services to a number of applications, and provide integration and reporting services for them via abstraction layers provided for each application within the database. It means that there will be several perfectly valid approaches to managing the change process. At one extreme, the database sits in an entirely separate development regime, sometimes not even subject to the same version control, test, and build processes as the application. At the other, the database code is treated on equal terms in source control to the rest of the code and settings of the application.

Where database and application are close-coupled, we can adopt a unified approach to the development and deployment of both application and database and, therefore, a unified strategy for versioning a project, with all project resources organized side by side in the same repository.

Of course, this often means little more than creating a directory for the database alongside the application, in the VCS, evolving the structure of the database rapidly alongside the application, with the necessary branching and merging (covered later) as understanding of the problem domain evolves. For a relational database of any size, with complex object interdependencies, this can prove challenging, especially given that when the team need to upgrade an existing database with what's in the VCS, then every database refactoring needs to be described in a way that will carefully preserve all business data (see Chapter 7, *Database Migrations: Modifying Existing Databases*).

For databases that support multiple applications, if the delivery of all the changes that span the applications and database haven't been planned properly, then the subsequent "big bang" integration between application and database changes can be painful and time consuming. This, along with deploying the new versions of database and application to QA for testing, then on to the operations team for deployment to staging and then, finally to production, forms part of the infamous "last mile" that can delay releases interminably.

Databases and their interactions with other databases, applications, or services, are not immune from the general rules that apply to other interconnected systems. There are several database design and versioning strategies that can help to allow the component parts to change without causing grief to other components which are accessing it. By applying the rules that govern the interfaces between any systems, it is possible to make database changes without disrupting applications, as well as to avoid complex branching strategies, while ensuring that database code is free from dependencies on database configuration and server-level settings.

Every application needs a published application-database interface

Regardless of the exact branching strategy, a team that creates an application that makes direct access to base tables in a database will have to put a lot of energy into keeping database and application in sync. When using "feature and release" branching, this issue will only worsen, the more branches we maintain and the more versions of the database we have in the VCS.

An application version should be coupled to an application-database interface version, rather than to a database version. Each application should have a stable, well-defined interface with the database, usually one for each application, if more than one application uses a single database. A good interface implementation will never expose the inner workings of the database to the application, and will therefore provide the necessary level of abstraction.

Usually, the application development team owns the interface definition, which should be stored in the VCS. It forms a contract that determines, for example, what parameters need to be supplied, what data is expected back and in what form.

The database developers implement the interface, also stored in the VCS. Database developers and application developers must carefully track and negotiate any changes to the interface. If, for example, the database developer role wishes to refactor the schema, he or she must do so without changing the public interface, at least until the application is ready to move to a new version, at which point the team can negotiate an interface update. Changes to the interface have to be subject to change-control procedures, as they will require a chain of tests.

The database developers will, of course, maintain their own source control for the database "internals," and will be likely to maintain versions for all major releases. However, this will not need to be shared with the associated applications.

Keep database code in a separate repository from application code

Unless your database will only ever be used by one version of one application, which is unlikely with DLM, you should keep the database code in its own version-control repository. This separation of application and database, and use of a published interface as described above, helps us to deploy and evolve the database independently of the application, providing a crucial degree of flexibility that helps to keep our release process nimble and responsive.

Adopt a "minimal" branching strategy

In many cases, we will store the files related to the main development effort in a common root subfolder of the VCS, often named **trunk**, but sometimes referred to by other names, such as **main** or **mainline**.

A VCS allows the team to copy a particular set of files, such as those in the trunk, and use and amend them for a different purpose, without affecting the original files. This is referred to as creating a **branch** or **fork**. Traditionally, branching has been seen as a mechanism to "maximize concurrency" in the team's development efforts, since it allows team members to work together and in isolation on specific features. A typical example is the creation of a "release" branch to freeze code for a release while allowing development to continue in the development branch. If changes are made to the release branch, normally as a result of bug fixes, then these can be merged into the development branch.

Branches can be a valuable asset to teams working with DLM, but should be used sparingly and with the idea that any given branch will only have a transient existence of just a few days. When working in a branch, there is a strong risk of isolating changes for too long, and then causing disruption when the merge ambush eventually arrives. It also discourages other good behavior. For example, if a developer is fixing a bug in a branch and spots an opportunity to do some general refactoring to improve the efficiency of the code, the thought of the additional merge pain may dissuade them from acting.

Instead we advocate that the team avoid using branches in version control wherever possible. This may sound like strange advice, given how much focus is placed on branching and merging in VCS, but by minimizing the number of branches, we can avoid many of the associated merge problems.

Each repository should, ideally, have just one main branch plus the occasional short-lived release branch. Instead of creating a physical branch for feature development, we use "logical" branching and feature toggles. When combined with small, regular releases, many small repositories, and the use of package management for dependencies (covered later in this chapter), the lack of branches helps to ensure that code is continuously integrated and tested. The following sections outline this strategy in more detail.

Minimize "work in progress"

Teams often use branches in version control in order to make progress on different changes simultaneously (see http://preview.tinyurl.com/jsaup2t). However, teams often end up losing much of the time they gained from parallel development streams in time-consuming merge operations, when the different work streams need to be brought together.

In my experience, teams work more effectively when they focus on completing a small number of changes, rather than tackling many things in parallel; develop one feature at a time, make it releasable. This reduces the volume of work in progress, minimizes context switching, avoids the need for branching, and helps keep changesets small and frequent.

Beware of organizational choices that force the use of branching

The need for branching in version control often arises from the choices, explicit or otherwise, made by people in the commercial, business, or program divisions of an organization. Specifically, the commercial team often tries to undertake many different activities simultaneously, sometimes with a kind of competition between different product or project managers.

This upstream parallelization has the effect of encouraging downstream teams, and development teams in particular, to undertake several different streams of work at the same time; this usually leads to branching in version control.

Another organizational choice that tends to increase the use of branching is support for multiple simultaneous customized versions of a core software product, where each client or customer receives a special set of features, or a special version of the software. This deal-driven approach to software development typically requires the use of branching in version control, leading rapidly to unmaintainable code, failed deployments, and a drastic reduction in development speed.

While there are strategies we can adopt to minimize the potential for endless feature branches or "per-customer" branches, which we'll cover shortly, it's also better to instead address and fix the problems upstream with the choices made by the commercial or product teams.

Encourage the commercial teams to do regular joint-exercises of prioritizing all the features they want to see implemented, thus reducing the number of downstream simultaneous activities.

Avoid feature branches in favor of trunk-based development

A popular approach within development is to create feature branches in addition to release branches. We isolate a large-scale or complex change to a component, or a fix to a difficult bug, in its own branch, so that it doesn't disrupt the established build cycle. Other developers can then continue to work undisturbed on "mainline." One of the major drawbacks of this approach is that it runs counter to the principles of continuous integration, which require frequent, small changes that we integrate constantly with the work of others. By contrast, trunk-based development encourages good discipline and practices such as committing small, well-defined units of work to a single common trunk branch.

In SQL Server, the use of schemas and the permission system, is the obvious way of enabling trunk-based development (see Tony Davis's article at http://preview.tinyurl.com/kud4zec). Schemas group together database objects that support each logical area of the application's functionality. Ideally, the VCS structure for the database will reflect the schema structure, in the same way that C# code can be saved in namespaces.

Features within a given schema of a database will be visible only to those users who have the necessary permissions on that schema. This will make it easier to break down the development work per logical area of the database, and minimize interference when all developers are committing to trunk. It also means that the "hiding" of features is extremely easy, merely by means of changing permissions.

Use feature toggles to help avoid branching

In application development, if we wish to introduce a new feature that will take longer to develop than our established deployment cycle, then rather than push it out to a branch we hide it behind a **feature toggle** within the application code. We maintain a configuration file that determines whether a feature is on or off. We then write conditional statements around the new feature that prevents it from running until enabled by the "switch" in the configuration file. It means the team can deploy these unfinished features alongside the completed ones, and so avoid having to delay deployment till the new feature is complete.

In database development, it is relatively straightforward to adopt a similar strategy. As long as we have a published database interface, as described earlier, we can decouple database and application deployments to some extent. The views, stored procedures and functions that typically comprise such an interface allow us to abstract the base tables. As long as the interface "contract" doesn't change, then we can make substantial schema changes without affecting the application. Instead of isolating the development of a new version of a database feature in a branch, the new and the old version exist side by side in trunk, behind the abstraction layer. (For more detail, see Martin Fowler's article at http://preview.tinyurl.com/m8gvdsp.)

It means that we can test and "dark launch" database changes, such as adding a new table or column, ahead of time, and then adapt the application to benefit from the change at a later point. Tony Davis's article, *Database Branching and Merging Strategies*, referenced above, suggests one example of how this might be done using a "proxy stored procedure."

By a similar strategy, we can avoid creating custom branches per customer requirements. Instead, we can use "feature toggles" or plugin modules to produce the effect of customization without the need for branching and per-client customization of the source code. Of course, systems with long lived customizations (for example a payroll system that must support different countries' or regions' regulatory frameworks) will probably also require architectural decisions promoting those requirements.

Make any non-trunk branches short-lived

Sometimes it is useful to create additional branches, particularly for fixing problems in the live environment, or for trying out new ideas. Branches that are short-lived can be a valuable asset to teams working with DLM. If a branch lasts only a few days, the person who created the branch knows exactly why it exists and knows what needs to be merged into the main branch; the drift from the main branch is small.

Branches that last for many days, weeks, or months represent a merge nightmare, where each branch is in effect a separate piece of software because the changes have not been not merged for a long time, possibly not until after the author of the changes has moved to a different team or even a different organization.

Account for differences in database configuration across environments

As discussed earlier, database source code does not consist merely of a number of tables and programmable objects. A database system is dependent on a range of different database and server-level configuration settings and properties. Furthermore, differences in some of these settings, between environments, such as differences in collation settings, can cause differences in behavior. As a result, we need to place into version control the scripts and files that define these properties for each environment.

However, we still want to use the same set of database scripts for a given database version to deploy that database version to any environment. In other words, databases' schema code and stored procedures/functions should be identical across all environments (development, test, staging, and production). We do not want to use different versions of database code for different environments, as this leads to untested changes and a lack of traceability. This means that we must not include in the scripts any configuration properties, such as data and log file locations, nor any permission assignments, because at that point we need one set of database scripts per version per environment.

Use database configuration files and version them

A common source of problems with software deployments in general is that the configuration of the software is different between environments. Where software configuration is driven from text files, we can make significant gains in the success of deployment and the stability of the server environment by putting configuration files into version control. Where security is an issue then this should be within a configuration management archive separate to the development archive. Whichever way it is done, it is best to think of it as being logically separate from database source because it deals with settings that are dependent on the server environment, such as mapping tables to drives, or mapping database roles to users and server logins.

Examples of configuration settings for databases include:

- **Database properties such as file layouts** – data file and log file
- **SQL Server instance-level configuration** – such as Fill factor, max server memory
- **SQL Server database-level configuration** – such as Auto Shrink, Auto Update Statistics, forced parameterization
- **Server properties** – such as collation setting
- **Security accounts** – users and logins
- **Roles and permissions** – database roles and their membership.

Scripts or property files that define and control these configuration settings should be stored in version control in a separate configuration management source-control archive. A general practice with SQL Server databases is to use SQLCMD, which allows us to use variables in our database scripts for properties like database file locations, and then reads the correct value for a given environment from a separate file. SQL Server Data Tools (SSDT) also exports a SQLCMD file to allow the same script to be used in several environments.

This approach is particularly useful for DLM because it opens up a dialog between software developers and IT operations people, including DBAs; both groups need to collaborate on the location of the configuration files and the way in which the files are updated, and both groups have a strong interest in the environment configuration being correct: developers so that their code works first time, and operations because they will have fewer problems to diagnose.

We recommend one of two simple approaches to working with configuration files in version control, as part of configuration management. The first approach uses one repository with branch-level security (or multiple repositories if branch-level security is not available).

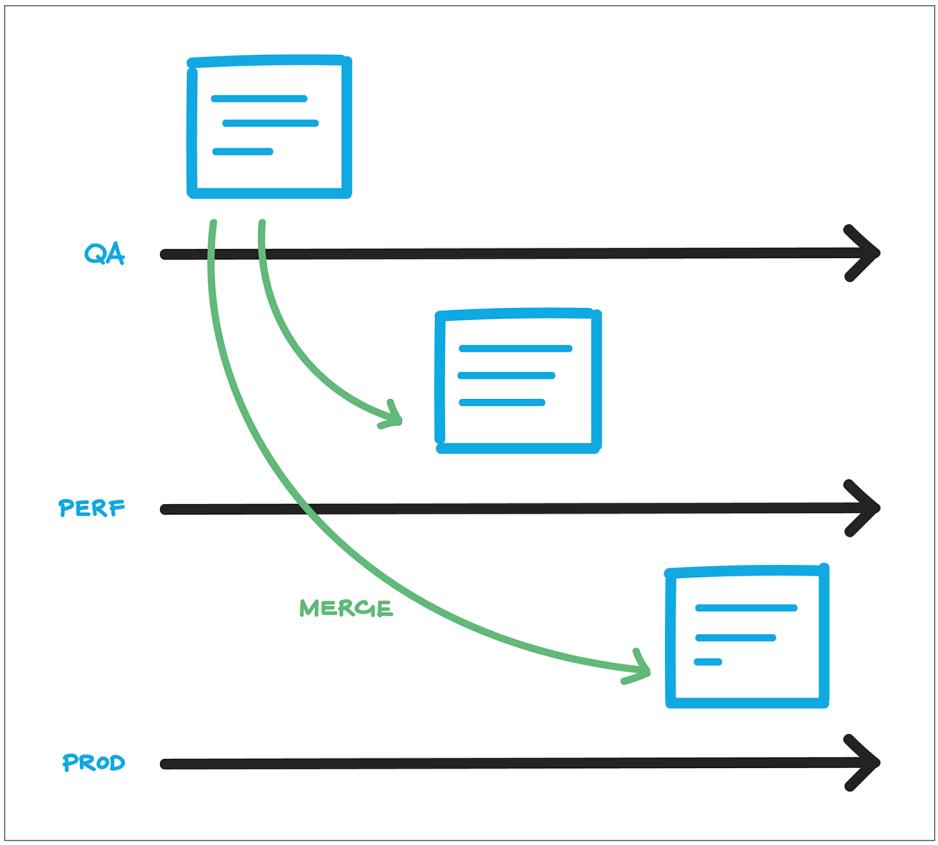

Figure 5-3: Configuration Management – single repository, branch-level security.

In this model, each environment has its own branch in version control, and settings are merged from one branch to another. This makes security boundaries simple to enforce, but changes are more difficult to track compared to the second model.

The second approach uses a single branch with multiple side-by-side versions of a particular configuration file, one per environment. In this model, per-environment security is tricky to set up, but merging and tracking changes is easier than in the first approach.

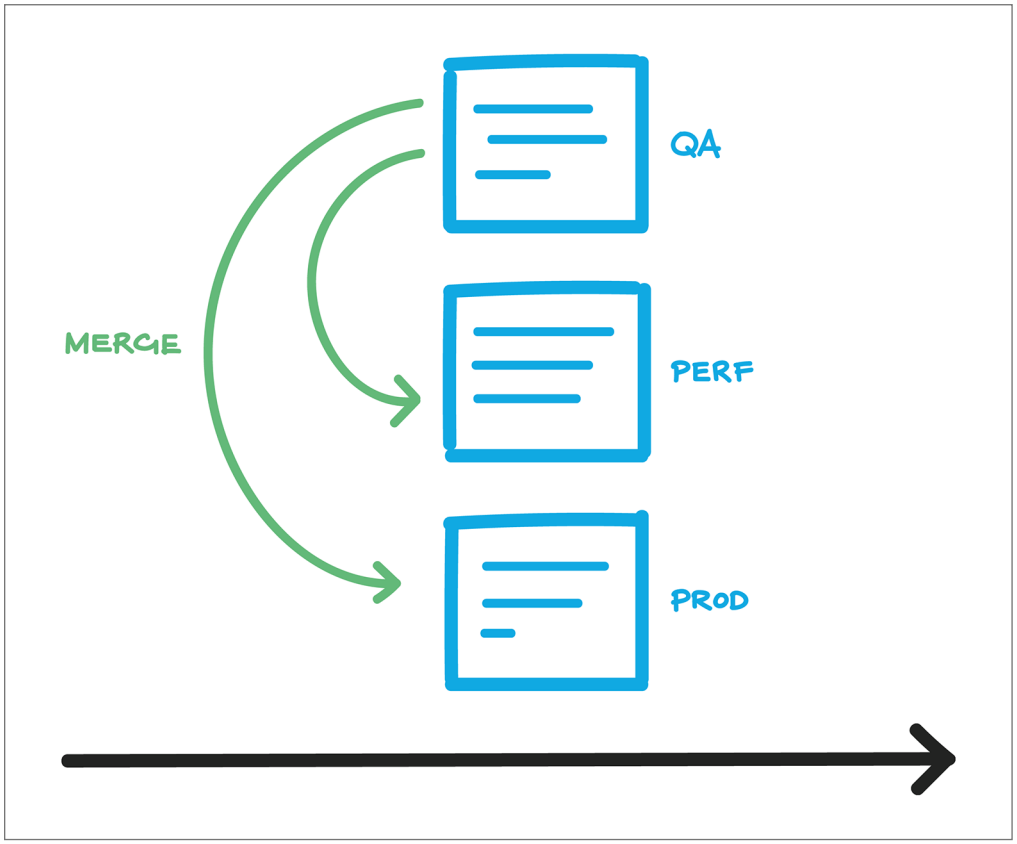

Figure 5-4: Configuration Management – single repository, single branch.

Either of these approaches will most likely lead to increased trust in environment configuration and more rapid diagnosis of environment-related problems.

Do not use per-environment repositories

The configuration information for all server environments should live in one version-control CM archive. Do not use a repository per environment, because this prevents tracing of changes between environments. The only exception to this will be for compliance reasons where the regulatory framework requires that configuration settings for the production environment are kept in a separate repository to all other configuration values.

Use packaging to deal with external dependencies

Many database applications have dependencies on third-party modules. Rather than import into our VCS every module required by any of our applications, we can integrate these libraries and dependencies through packaging, for example using NuGet packages for .NET libraries, or Chocolatey for Windows runtime packages. We can specify which versions of which libraries each application repository depends on, and store that configuration in version control. At build time the packages containing the library versions we need are fetched, thus decoupling our dependencies on external (to our repo) libraries from the repositories organization.

We can also use package management to bring together configuration and application or database packages by having the configuration package depend on the generic application or database package which, in turn, might depend on other packages.

Summary

By putting database code into source control, we provide a way of making it easier to coordinate the work of the different teams who share responsibility for the database. At different points in the life of a database, it is the focus of very different types of activities, ranging from creating an initial data model to decommissioning.

Version control acts as a communication channel between teams, because the changes captured in the version-control system are treated as the single definitive "source of truth" for people to collaborate on, bringing together delivery, governance and operations around a common DLM approach.

A working database will have a variety of materials that are required by the delivery team, governance or operations. With source-controlled archives, it is always obvious where the source of any material is, who made which alteration and update, when, and why. Any materials, whether code, instructions, training materials, support information, configuration items, step-by-step disaster recover procedures, sign-offs, release documents, ER diagrams, or whatever, can be accessed easily.

Version control isn't the core of DLM but it makes it achievable by providing a source of information that is accessible, traceable, reliable, repeatable, and auditable.

6 – Better Ways to Build a Database

The purpose of a database build is simple: prove that what you have in version control can successfully create a working database. And yet many teams struggle with unreliable and untested database build processes that slow down deployments and prevent the delivery of new functionality. This chapter will show you how to achieve an automated and reliable database build that is only as complex as the database system it needs to create.

Whatever method you use to develop a database, it is important to build it regularly from the DDL scripts and other materials in version control. This ensures that any version can actually be built successfully, highlights potential difficulties before they become problems, and prevents work being lost.

The need to perform regular builds means that it needs to be made as easy to as possible, so that it becomes a routine part of the workflow. For a simple database, this build process is trivial; but it gets trickier to get right as the database gets more complex.

In this chapter, we'll explain what we mean by a database "build," and then suggest how to apply DLM techniques to make it automated, predictable and measurable. In later chapters, we will delve into the details of continuous integration and testing, and explain how to release and deploy the artifact that is the end-product of the build process.

What is a database build?

In application development, we build a working application from its constituent parts, compiling it in the correct order then linking and packaging it. The "build process" will do everything required to create or update the workspace for the application to run in, and manage all baselining and reporting about the build.

Similarly, the purpose of a **database build** is to prove that what we have in the version-control system – the canonical source – can successfully build a database from scratch. The build process aims to create, from its constituent DDL creation scripts and other components in the version-control repository, a working database to a particular version. It will create a new database, create its objects, and load any static, reference, or lookup data.

Since we're creating an entirely new database, a database build will not attempt to retain the existing database or its data. In fact, many build processes will fail, by design, if a database of the same name already exists in the target environment, rather than risk an `IF EXISTS...DROP DATABASE` command running on the wrong server!

Builds versus migrations

In environments such as production, staging or user acceptance testing (UAT), teams sometimes use the term "the build" more generically, to mean "*the artifact that will establish the required version of the database in that environment*." Unless the database doesn't already exist, the build in these environments is in fact a migration, and will modify the existing database to the required version, while preserve existing data, rather than tearing down and starting again. We focus only on building a database in this chapter. We cover modification of existing databases in Chapter 7, *Database Migrations: Modifying Existing Databases*.

A database is likely to be built many times during the development and testing process. Once it has been tested fully, in development, we should be able to use the same build process on all environments, assuming we've separated environmental dependencies from the build script itself (more on this shortly).

A significant build is the one that produces the release candidate, and a subsequent deployment involves the same build process, but with extra stages, for example to account for the production server settings, and to integrate the production security, and incorporate production architecture that isn't present in development, such as SQL Server replication. In fact, Microsoft refers to a build as "*the mechanism that creates the necessary objects for a database deployment*." A build, per their definition in the SQL Server Data Tools Team Blog at http://preview.tinyurl.com/zju2gvr is just the act of creating a script or a DACPAC file. They then refer to the act of applying that script to the database as "publishing the build."

The rapid and efficient build process is essential for some databases that are released in several variants for individual customers or installations for any particular version. It is in these development environments that a build process that takes its source from the VCS and maintains strict versioning comes into its own.

The purpose of the build

From the database developer's perspective, a build provides a regular health check for the database. During development, the build process should be automated and performed regularly, because it tells the team whether it is possible to do this with the latest version of the committed code, and keeps all the team up to date with the latest development version. The canonical database that results from a successful build can also be checked against other development versions to make sure that they are up to date, and it can be used for a variety of routine integration and performance tests.

If the build is also sufficiently instrumented (see later) then if a build breaks, or runs abnormally long, then the team need access to detailed diagnostic information and error descriptions that will allow them to identify the exact breaking change and fix it, quickly.

In a shared-development environment, it is possible to avoid building the database. However, a build process that is manual and poorly tested will inevitably lead to delayed and unreliable database deployments. The database build process becomes a bottleneck for all downline deployment processes, and deploying the application must wait while the manual steps required for the database build are navigated. In addition, because these processes are likely to be manual, they are much more error prone. This means that the application deployment is waiting on a database deployment that may not even be correct when it eventually gets finished.

A database build process that breaks frequently and causes subsequent downstream problems with a database deployment will knock the confidence of the other IT teams, perhaps to the point that they request that the development team perform *less* frequent releases, until all the database build issues are resolved. Unreliable database build processes (as well as unreliable migration processes) can prevent the applications that use the database from delivering new functionality quickly enough.

DLM cannot be performed effectively unless you get your database builds under control within the development environment. Conversely, if automated builds run regularly and smoothly then, assuming all necessary database and server-level objects have been accounted for in the build, the team will be confident that little can go awry when it is shepherded through the release process to deployment.

The mechanics of a database build

The database build mechanism will need to pull from the version-control system the correct version of all necessary components. This will most often include the scripts to build all schema-scoped objects such as tables, stored procedures, functions, views, aggregates, synonyms, and queues. It will also need to load any necessary static data.

A common way to represent the database in version control is as a set of individual object scripts. It is usually a convenient way of versioning a database system because you get a useful and easily-accessed history of changes to each object, and it is the most reliable way of ensuring that two people do not unintentionally work on the same set of objects. It is, however, not the only legitimate way of storing the source of a database in a version-control system.

In fact, when building many databases, you are unlikely to want to create every table separately, via **`CREATE`** statements. A small-scale database can be built from a single script file. If you prefer, you can script out each schema separately. For larger, more complex databases, where it becomes more important for the team to be able to see the current state of any given object, at any time, you'll probably be versioning object-level DDL scripts.

Scripting the database objects

Many databases can be built solely from scripts that create the schemas, database roles, tables, views, procedures, functions, and triggers. However, there are over 50 potential types of database objects in SQL Server that can form part of a build. The vast majority are scriptable.

Tables are by far the most complex of scriptable components of a database and they provide the most grief for the developer. In SQL Server, tables are represented as objects but the schema that it resides in isn't. A table's constituent parts are child objects or properties in a slightly confusing hierarchy, as shown in Figure 6-1.

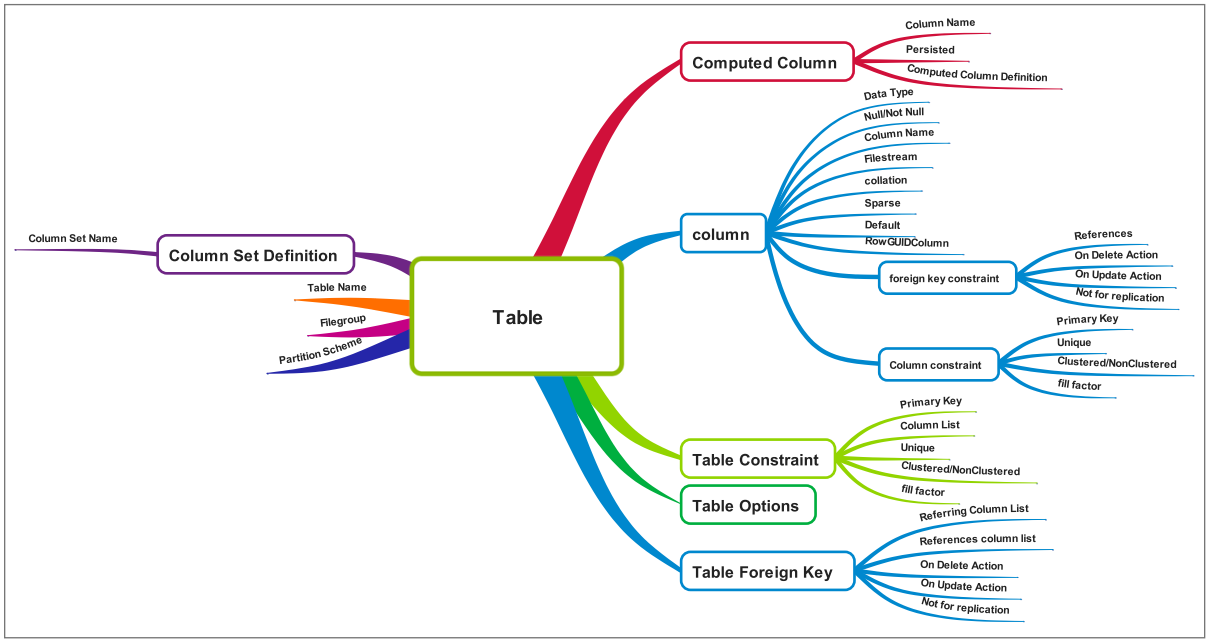

Figure 6-1: Properties of a database table.

Tables can be built in one step with a **`CREATE TABLE`** statement or can be assembled from the initial **`CREATE TABLE`** statement and decorated subsequently with columns, constraints, and **`FOREIGN KEY`** constraints, by using **`ALTER TABLE`** statements. Without a doubt, table scripts are best written in the former table-level way, rather than the latter method of treating the child objects and properties in their own right. It is clearer to read, and shows in one view what has changed between versions of the table. It gives a much more concise history of the changes.

Scripting server-based objects

It is a mistake to think that, by storing the database objects in source control and building those, you have the entire database. Some components aren't actually stored in the database. If they are used, they may be at server level, and or in SQL Agent. If these server objects, such as database mail profiles, logins, SQL Agent jobs and alerts, schedules, server triggers, proxy accounts, and linked servers, are required for the functioning of the database, then they must be held as SQL scripts in the development VCS and built with the database.

Some server-based objects are associated with the database application but need entirely different scripts for each server environment. Operators, the contact details of the person who is alerted when a database goes wrong, are a simple example of this type of script. As we prepare to deploy the database to production, the same build process must be able to define the appropriate operator in each environment, whether it be development, test, integration, staging, or production.

SQL Server Management Objects (SMO) exposes server-level objects through the **`Server`** and **`JobServer`** classes, and we can create a PowerShell script, for example, to iterate over all of the various collections of this class, containing the jobs and alerts and other server objects, along with associated operators, schedules, and so on. The example shown in this article at http://preview.tinyurl.com/zscsu4n scripts out these objects into a directory named according to the server name.

In many organizations, scripts for these server objects are likely to be stored in a separate version-control system under the control of the operational or production DBA staff, on the central management server. In many cases, they are not typically put into any development scripts for production. As such, the specifics of dealing with these server objects during a subsequent deployment will be saved for Chapter 10, *Database Deployment and Release*.

However, a DLM approach to database builds requires that the development environment mimic production as closely as possible (more on this shortly), and demands early and frequent collaboration between development, governance, and operations teams, in order to avoid as far as possible build difficulties arising from differences between server environments.

The build scripts

A database build process should be about as complex as the database system it needs to create. For a small database, a single build script can encompass the entire database. As a database increases in size, the scope can be decreased first to the schema level, and finally to the object level.

To be productive, you will probably use a variety of tools and techniques. If you use an ER diagramming tool, such as the table-design tool in SSMS, or one of a plethora of other table designing tools on the market, to design the database, then you can make the "first cut" by generating the table-creation scripts directly from the tool.

However, remember that it is then the developer's responsibility to subsequently edit that "generated" script to ensure that the script conforms to the house style and standards, and is commented adequately to ensure the whole team understand the purpose of each object. It is a bad idea to make a habit of reverse-engineering the source code of an object from your development server to store in the VCS. For example, a table may be altered slightly by a diagramming tool as part of a wider re-engineering task. Would you overwrite the easily-understood, carefully-documented and hand-crafted table script with a generated script to update it, and lose all that extra information? The answer is probably "never."

In addition to using diagramming tools and other visual tools, you will probably "hand-cut" tables from raw **`CREATE`** statements, and will probably over time develop and reuse boiler-plate designs for handling common functionality such as Names and Addresses. You have plenty of freedom and more options than you'd wish for.

Single script builds

If your database comprises just a handful of tables and a few views and stored procedures, there is no need for a complicated build process using object-level DDL scripts. You can just save in the VSC a single build script that creates the database, and then all tables and code objects, and loads any necessary data. This script will incorporate referential integrity constraints so there are no concerns over dependency order.

To create the first cut of a database, in version control, you may consider generating the build script from a live development database, using a database diagramming tool, or SSMS, or via an SMO script. You can then use a tool such as SQLCMD or PowerShell to run the script on all required environments, assuming you've separated out environmental dependencies, such as database configuration properties, security settings such as database roles, and so on. Using a tool like SQLCMD, we can use variables for properties like database file locations, and the correct value for a given environment will be read in from a separate file.

Schema-level builds

SQL Server works naturally at the level of the schema. Developers that use a login that is restricted to the schema that they are working on will see only those schema objects and their scripts will contain only those objects. To make this work properly, they will only access scripts on other schemas via an interface, consisting of a view, procedure or function. These

scripts can be hand-cut, and this is customary at the later stages in development. In the earlier stages, where developers are using GUI tools such as ER diagramming utilities, the build script can usually be created via an SMO script or via the SSMS GUI.

Object-level builds

Eventually, a database will grow to the size where only object-level scripting is feasible. Normally, this is only required for the schema-bound objects such as tables, functions, views, and procedures. Other database objects such as XML Schema collections, schemas, and service broker objects can be placed together in one script per object type. There are many PowerShell-based scripts that can be used to create object-level scripts for the first time if the database was formerly saved to the VCS at database or schema level, or if the database was initially cut from an ER diagramming tool. You can also script each object using any good database comparison tool that compares the live development database as source against the **`model`** database on the same server.

The build process

Whatever level of build script is most suitable for the stage of database development, it is always best to store the source in a VCS archive. The first advantage is that it makes it easy to audit the change history through standard VCS mechanisms, and identify quickly who added or removed code.

The second advantage is intelligibility, because the VCS will always hold the latest build script for each object, ideally well formatted and liberally commented. The **`CREATE`** script for a table, for example, can be complex. There are many subordinate objects that have no existence without the table, such as constraints, indexes, triggers, and keys. A good table script is liberally sprinkled with comments, and the script defines all the subordinate objects at the same time. The table is formatted to make it more intelligible. Column constraints are created with the columns rather than trailed afterwards as **`ALTER TABLE`** afterthoughts. Table constraints are placed likewise at the table level.

Having in the VCS a well-documented build script for each object makes it very easy for the team to see the exact state of an object at any given time, and to understand how changes to that object are likely to affect other objects.

Manual builds from database object scripts

If we store individual object scripts in version control to then build a functioning database from scratch, we need to execute these scripts on the target server, in the right order.

Firstly, we run a script to create the empty database itself, and then we use that database as the context for creating the individual schemas and the objects that will reside in them, by object type and in the correct dependency order. Finally, we load any lookup (or static) data used by the application. We may even load some operational data, if required. There are other scripts to consider too, such as those to create database roles, which will be mapped against users separately for each environment, as well as those to set up server-level objects and properties, as discussed earlier.

To build all the required database objects in the right order, you can specify these individual files in a "manifest" file, for example as a SQLCMD manifest file, using the " `:r` " command, and this can then be run to perform the build. A SQLCMD file can also specify the pre-build and post-build script (see later) and is kept in the VCS, and can be updated whenever a new database object is created.

The manifest is necessary because the objects must be built in the correct dependency order. SQL Server, for example, checks that any table referenced by a **`FOREIGN KEY`** actually exists. If it isn't there at the point that you execute the **`CREATE`** statement, then the build breaks. The safe order of building tables isn't easy to determine and it is even possible to create circular dependencies that will defeat any object-level build. There are ways around this difficulty such as:

- put the **`FOREIGN KEY`** constraints in the VCS separately and then execute these after the tables are created
- put **`FOREIGN KEY`** constraints in the build script but disable them, and add a post-build script to enable them.

However, the build process can get complicated and error prone, very quickly if you're putting the build script together by hand. The manifest file has to be altered whenever necessary and can become a cause of errors when the database grows in size and complexity. At that point, you will need to automate the build more comprehensively to dispense with the need for the manifest.

Automated builds from object scripts

There are tools that can automate the generation of the build script, from object-level scripts. Essentially, these tools consist of a comparison engine to determine the difference between the target database and the source files and packages in the VCS, or a source database. SQL Compare (Oracle, SQL Server, and MySQL) can read a number of scripts and combine them together in the right order to create a synchronization script that can be executed to publish a new database. In effect, a synchronization script that publishes against an entirely blank database, such as **`MODEL`**, is a build script.

The tool compares a live target database at version *x* with source scripts that describe the database at version *y*, and generates a change script to turn database version *x* into database version *y*. If your comparison tool can't compare to the source scripts then you would still require a build of the database from the VCS at version *y*.

The DACPAC (SQL Server) works rather differently and requires the developers to create a logical "model" of the database from scripts. It does not fix the problem of getting object-level scripts in the right order, since the object model has to be built in the correct order.

Modifying existing databases

We can use essentially the same technique to create a synchronization script that that **`ALTER`**s a target database at version *x* and changes it to version *y*, while preserving existing data, i.e. to perform a migration. We'll discuss this in Chapter 7, *Database Migrations: Modifying Existing Databases*.

The comparison tools generally have various options for excluding or including different kinds of database objects (procedures, functions, tables, views, and so on) and some have support for data comparisons, allowing new reference data (such as store locations or product codes) to be updated in the target database.

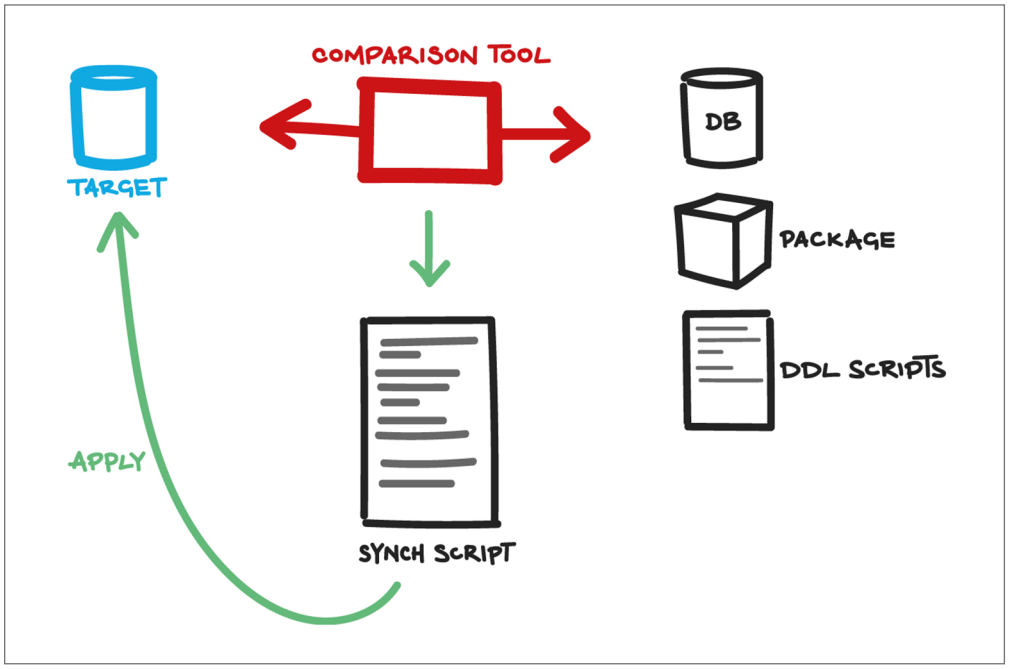

Figure 6-2: Auto-generating a database build script from object scripts.

While this build process relies on the object-level DDL scripts in the VCS as the source of truth, the team needs to test and archive every synchronization script that has been used to build a database in this way, because the synchronization script is one step away from that truth.

Achieving repeatable builds

Each build script should contain any necessary logic to ensure that it is idempotent, meaning that you can run it over and over again and it will achieve the same result. You are likely to hear of different types of build script that differ mainly by the use of **guard clauses** that prevent mistakes from harming data. The main types are the **brute force**, **kill and fill** and **idempotent** scripts. Some build systems offer **transactional** build scripts as an optional extra, so we'll cover this type of script, too.

The brute force script

The simplest build script to create the database objects will assume an empty database. It creates each object in a dependency order that prevents references to objects that haven't yet been built. If you accidentally run it twice, then you will get a ghastly list of errors, but no harm is done.

The kill and fill script

A slightly more complicated one is the kill and fill. Before executing the DDL script to create an object such as a table, function, procedure, view, or whatever, it will check whether it already exists, and if it does, it removes it. This allows you to rebuild a database regularly on the same server without the need to re-create the empty database. It has two snags. Firstly, a careless or distracted DBA can accidentally run it on a live production version of the database and thereby destroy several tables of data. Secondly, it allows you to miss out an important part of the build, which is to create the empty database with the correct configuration options.

The idempotent script

With an idempotent script, you will be unable to do any harm from running it more than once. If it finds a version of the database already there, it will check the version of the database, and refuse to run if it isn't the same version that the script builds. If it is the same version, it will merely build any missing objects. It does not attempt anything resembling a change to an object.

The transactional script

The transactional script will attempt the process of building part or all of a database and, if it hits an error, it rolls all the successful operations back to leave the database in the state it was in before the script was executed. The build script is different from a migration in that, if any part of a build fails, the whole operation has to be completely torn down. The rollback of one component script isn't necessary because the whole database will be dropped anyway.

However, this does not apply to server-based components or scheduled tasks which are always better done by a transactional script. Migrations (covered in another chapter) should always be transactional, otherwise the database is left in an indeterminate, and possibly non-working, state.

Handling data

Often, the build process will need to load any necessary data required for dependent applications to function, and possibly to support database testing. There are various types of mainly immutable data that we may need to load because it is required for the database to function, including:

- **reference data** – data that refers to relatively unchanging data such as information about countries, currencies, or measurements
- **static data** – usually a synonym for reference data
- **enumerations** – short narrow tables that represent sets of unchanging entities
- **seed data** – data that is required to initialize a hierarchical table with the root of the hierarchy
- **domain data** – data that defines the business domain and which will not change in the lifetime of this version of the database
- **error explanations for business or domain errors** – such as for bad data in ETL jobs or failed business processes.

Such data is the only data that should be held in version control. All other data is loaded in a post-build script as dictated by the manifest, unless the build takes place against a version that already has the required data.

Test data should be generated to the same distribution, characteristics and datatype as the potential or actual production data, and each type of test is likely to require different test data sets. Performance and scalability testing will require a range of large data sets, whereas integration test is likely to require standard before-and-after sets of data that includes all the likely outliers. Test data is best loaded from native BCP format using bulk load.

Pre- and post-build processes

A build process will generally include pre-build and post-build processes. Various automated or semi-automated processes are often included in the build but are not logically part of the build, and so are slotted into one or other of two phases of the build called the "pre-build process" and the "post-build process." A build system will have slots before and after the build to accommodate these. These are most likely to be either SQL or PowerShell scripts.

An important pre-build process, for example, is preparing the workspace, which means ensuring that all the necessary requirements for the build are in place. What this entails is very dependent on the nature of the database being built, but it might include preparing a Virtual Machine with SQL Server installed on it at the right version, or checking to ensure that the platform is configured properly, and that there is sufficient disk space.

Typical post-build processes will include those designed to manage team-based workflow or reporting. For example, we need to log the time of the build, the version being built and the success of the build, along with warning messages. It might involve email alerts and could even list what has changed in the build since the last successful build.

Also, of course, there will need to be a post-build step to validate the build. Specially-devised tests will not only check that the build was successful but that no major part of the system is entirely broken.

Promoting builds through the environments

Ideally, you should not have a different build process for development, test, integration, release and deployment. Our goal is that the build is fully automated, and that the build package that we produce, and subsequently test, will be used to build the database, at that version, for all environments, from development through QA and staging up to production.

In order to achieve automated, consistent builds across all environments, it is best if the development environment mimics the production environment as closely as possible in terms of the architecture of the database and objects, and in the surrounding database technologies.

This isn't always possible. Enterprise databases will likely use features such as partitioned tables, ColumnStore indexes, and many other Enterprise features, which cannot easily be imitated in the development environment. In addition, there may be disaster recovery, high availability, or auditing features, such as replication or change data capture, that aren't appropriate in the development environment.

DLM encourages teams to collaborate closely, making it easier to provide "hooks" in the build to allow aspects of the build to be different in each server environment, for example to associate the appropriate logins with each server and database role, in each environment, or to provide the correct auditing regime.

Builds in the development environment

In order to achieve build consistency across environments, all DLM solutions to the database build process need to offer a high degree of freedom to what is included, or not, in the build process.

A build process within development will generally use only local resources. A DLM build is designed to be flexible enough that external dependencies such as ETL processes and downstream BI can be "mocked," by creating interface-compatible stubs, to enhance the range of testing that can be done and allow development to run independently. This means that upstream ETL processes will be mocked sufficiently for them to be tested. Allied databases, middleware services, and message queues will either be skeletal or mocked. Likewise, downstream processes that are "customers" of the data from the database are generally also mocked.

A development build process will also have a security and access-control system (e.g. `GRANT`s and user accounts) that allows the developers complete access. If it is role-based, then the production access control can easily be used for a production build during the deployment process. Typically, operations will want to assign users via active directory, so the production build will have a different set of Windows groups created as database users, and assigned database roles as befits the security model for the database.

One step toward the goal of consistent builds is to ensure that any necessary modifications to the build process for the different environments should be in the manifest rather than in the scripted process itself. All these manifests can be saved either in development source control or on a central management server.

Essentially, we need a clear separation between the data and the database configuration properties. We can use templates to cope with such challenges as references to external systems, linked databases, file paths, log file paths, placement on a partition scheme, and the path to full-text index files, filegroup placements, or file sizes, which would produce different value for each environment. We can supply separate configuration files, tested by your CI processes (see Chapter 9, *Database Continuous Integration*), as part of the database release package.

For example, we can:

- **specify environment-specific parameters** in environmental configuration files – such as the location of the database files, their size, growth rates and so on
- **remove all environment-specific statements from the build and migration scripts** – no change of the configuration should involve any code or script change

- **make the security configuration easily maintainable** – security (logins, roles, database object permissions) settings belong in configuration files, not in T-SQL scripts.

Builds in QA, staging and production

For a database to work properly, when it comes time to deploy to production, or to production-like environments, the canonical source, in the VCS, will also need to include components that are executed on the server, rather than within the database. These **Server objects** include scheduled jobs, alerts, and server settings. You'll also need scripts to define the interface linking the access-control system of the database (database users and roles) to the server-defined logins, groups, and users, so that the appropriate users can access the roles defined within the database.

Therefore, if a build succeeds and is validated, it means that all the DDL source code, assemblies, linked servers, interfaces, ETL tasks, scheduled jobs, and other components that are involved in a database have been identified and used in the right order. This makes it more difficult for a subsequent deployment to fail due to a missing component, and all of the people involved in the database lifecycle can see, at any stage, what components are being used.

The characteristics of a DLM database build

If the database development team does not rigorously enforce the discipline of always starting from a known version in source control, and regularly testing to prove that they can build the required version of the database, then you are likely to suffer with an unreliable and time-sapping database build process.

A fairly typical non-DLM approach goes something like the following. The development team make changes to a shared database, either directly or with the help of a DBA or a database developer. When it comes time to release a new version of the application, the development team works with the DBA to come up with a list of just the database changes that are needed to support the new version. They generate a set of scripts, run them against the database in the test environment (although, in the worst cases, this is done directly against a production environment), and then build the application.

Testing reveals that some of the necessary database changes aren't there, so additional scripts are generated and run. Next, they discover that some scripts were run that implement database changes that the application is not yet ready to support. They generate another set of scripts to roll back the unwanted changes.

This chaotic process continues until, more by perseverance than by planning, the application works. The team now have a whole new set of scripts that have been created manually, and that will have to be run in the correct order to deploy to the next environment.

This causes all sorts of problems. Development and testing are slowed down because of all the additional time needed to identify the changes needed. The manual process of generating scripts is time consuming and very error prone. It leads to delayed functionality, and often to bugs and performance issues. In the worst cases, it can lead to errors being introduced into the data, or even loss of data.

To avoid this sort of pain, database builds (and migrations) should be automated, controlled, repeatable, measurable, and visible.

Automated

As described previously, to allow a database to be built that has scripts with a number of different objects in them, you either need to create, and subsequently maintain, a manifest file that lists the files in the order of execution, or you should use a tool such as SQL Compare which can read a directory of scripts as if it were a database.

- If using a manifest, keep that in source control with the scripts.
- Label and branch the manifest with the scripts.
- If using a differential tool, such as SQL Compare, save the generated build script in source control with the appropriate label or branch.

A big part of automation is having the ability to both replicate what you've done and go back and review it. If you keep the manifest or generated scripts in source control, you'll always be able to do this.

Although a build tool or proprietary build server is often used to automate database builds, it is not necessary. You merely need a script that executes each script serially in order, and SQLCMD.exe can be used easily to execute a SQL file against a target server.

If you are using a shared-development server with the development database in it, and it is kept entirely up to date with version control (all current work is checked in) then you can calculate the build order with a SQL routine that does a topological sort of the dependencies between objects to list first the objects that have no dependencies and then, successively, all the objects that are depending only on already listed objects, until all objects are accounted for. You can, of course, create a PowerShell routine that calls SMO to do the task for you. If all else fails, you can use SSMS to generate an entire build script for you from the development database only to use the order in which the objects were created to determine the order for your manifest.

Frequent

During development, the build process should be done regularly, because it tells you whether it is possible to do this with the latest version of the committed code, and allows you to keep all the team up to date. The canonical database that results from a successful build can also be checked against other development versions to make sure that they are up to date, and it can be used for a variety of routine integration and performance tests. Frequent or overnight builds can catch problems early on. A build can be scheduled daily, once all developers' changes have been committed and if necessary merged, or after every significant commit, when any component parts change. Testing is performed on the most recent build to validate it. Usually, builds are done to the latest version, but with version control, ad hoc builds can occasionally be done when necessary on previous versions of the code to track down errors, or to revive work that was subsequently deleted.

Repeatable

As described earlier, we should be able to run that same script over and over on any system without it ever breaking because it had been run previously. A DLM build process should never result in a partial build. If it fails at any point, it should roll back all the changes to the starting version. This requires wrapping the build into a transaction that rolls back on error. If a build is successfully automated, it is much more likely to be repeatable in exactly the same way, with minimal operator intervention. The governance process will be more confident in the time that must be allowed for a build process and validation, and will be much more confident of a successful outcome. This makes the entire end-to-end delivery process more predictable.

Tested

Of course, we need to validate not just that the build or migration succeeded, but that it built the database exactly as intended. Are your database and static data intact? Do all parts of your application and important code still function as expected? Are all the database objects there and correct?

By incorporating automated testing into your build process you add additional protections that ensure that, ultimately, you'll deploy higher-quality code that contains far fewer errors. We address testing in detail in Chapter 8, *A DLM Approach to Database Testing*.

Instrumented

Once you have the build process automated, the next step is to set up measurements. How long does a build take? How many builds fail? What is the primary cause of build failure?

You need to track all this information in order to continuously improve the process. You want to deliver more code, faster, in support of the business. To do this, you need to get it right. If you're experiencing the same problems over and over, it's best to be able to quantify and measure those so you know where to spend your time fixing things.

As the databases increase in number, size, and functionality within an organization, the complexity and fragility of the build process will often increase, unless serious thought has been given to structuring and modularizing the system via, for example, the use of schemas and interfaces.

A common blight of the database build is the existence of cross-database and cross-server dependencies. At some point, the build chokes on a reference from one object to a dependent object in another database, which it can't resolve for one reason or another. This might typically be because the database containing the object is not available in the context in which the current database is being built. It's often good practice in terms of data architecture to remove the dependency to a higher layer (integrating in a service for example), but if that's not possible, you might want to examine "chaining" database builds to best satisfy dependencies.

At this point, it's very important that the build process is well instrumented, providing all the details the developer needs, including the full stack trace if necessary, to pinpoint with surgical precision the exact cause of the problem. In poorly instrumented builds, it's more likely the developer will see a vague "*Can't find this column*" error, and will be left to sift

through the remaining few thousand objects that didn't load to find the one that caused the problem. No wonder, in such circumstances, database builds and releases are infrequent and often delayed, and therefore potential problems not spotted until much later in the cycle.

Measurable and visible

An automated build can be more wide-ranging, building not only the database, but also any automatically-generated build notes and other documentation such as Help pages.

It is easier to report on an automated build and, as well as simple facts such as the time it took and whether it succeeded, the number of warnings, and even an overall code-quality metric, or code policy check, if necessary. This allows the governance process to do its audit and accountability tasks, and firm up on planning estimates.

Any build and version-control system must allow developers, as well as people outside the development team, to be able to see what is changing and why, without needing time-wasting meetings and formal processes.

Advanced database build considerations

There seem to be an infinite variety of databases and this is reflected in the number of options that have to be considered in the build and subsequent deployment process. There are many choices to be made as to how the build processed in case of various eventualities. Here are just some of them (we cover others in Chapter 7, *Database Migrations: Modifying Existing Databases*).

Is the server version (platform) compatible with the database code?

Different versions and editions of the RDBMS will have different capabilities. It is easy to produce working code for one version of SQL Server that, on a different version, will fail entirely, or work in a subtly different way or with degraded performance. A database is always coded for a particular version of the RDBMS. If the database specification includes

the compatibility-level of the database, or the range of versions it must run on, then the build script should contain a guard clause to check the compatibility and version of the host SQL Server that the database is being installed on, and fail the build if it is incorrect.

Is the default collation of the database the correct one?

A collation will affect the way that a database behaves. Sometimes, the effect is remarkable, as when you change from a case-sensitive to a case-insensitive collation or the reverse. It will also affect the way that data is sorted or the way that wildcard expressions are evaluated. This will produce subtle bugs that could take a long time to notice and correct. To be safe, the build script should check the database collation and cause an error if it is incorrect.

Can database-level properties be changed in the build?

Code that works under one database configuration can easily be stopped by changing a database-level configuration item. When a server creates a database, it uses the database-level configuration that is set in the **`model`** database. This may be quite different from the database configuration on the developers' servers and, if these differences are allowed, the result will be havoc. The build script should check to ensure that the database has the correct configuration for such things as the default isolation level, date format, language, and ANSI nulls.

Should the database being built then be registered as a particular version on the server?

SQL Server allows databases to be registered when they are built or modified, meaning that an XML snapshot of a database's metadata is stored on the server and can be used to check that no unauthorized changes have been made subsequently. It is generally a good precaution to register a database if you are using DacFx, but very large databases could possibly require significant space.

Should all constraints be checked after all the data is inserted into the build?

This only applies if some sort of static data is imported into the database as part of the build. In order to guarantee a fast data import, constraints are disabled for the course of the import and then re-enabled after the task is finished. By re-enabling the constraints, all the data is retrospectively checked at once. It is the best time to do it but it could be that more data has to be inserted later into a related table, and so this should, perhaps, be delayed.

Should the build allow for differences in file paths, column collation, and other settings?

Many paths and database or server settings may vary between server environments. For example, there will likely be different file paths for data and log files, perhaps different file-group placements, a different path to full-text index files, or for a cryptographic provider.

Some settings, such as file paths, log file paths, placement on a partition scheme, the path to full-text index files, filegroup placements, or file sizes have nothing to do with development and a lot to do with operations and server administration. They depend on the way that the server is configured. It can be argued that the same is true of FillFactor. There are dangers in overwriting such things as the seed, or increment to identity columns, or collation.

These settings should be in a separate build script for each server environment or dealt with by SQLCMD variables that are set at execution time.

How should the build handle different security options between server environments?

How should the build handle differences in the security settings, security identifiers, permissions, user settings, or the role memberships of logins of the build, for the different server environments? This very much depends on the context of the build, and the way that the database handles access control. If the database is designed to hold live personal, financial, or healthcare data, it is likely to be illegal to allow the same team of people to both develop

the database and to determine the access rights or security settings for database objects. Best practice is for these to be held separately in a central CMS archive by operations, and determined according to whether the database holds real data or spoofed test data.

In practical terms, it makes sense to base the development access-control system on database roles and assign these to users or Windows groups by a build script that is based on the one provided in the release, but with the appropriate assignments made for the server environment, whether QA, staging, or production, and held in a CMS archive.

Within development, the problem is greatly simplified as long as live data is never allowed. The development and test builds can run unhindered. As soon as live data is used, as in staging, forensic analysis, or production, then all security and access control settings need to comply with whatever legislative framework is in force.

Summary

When developing a database application, it is a mistake to believe that database build is a simple task. It isn't. It has to be planned, scripted and automated. The build must always start from a known state, in the version-control system. You can then use various means to generate the scripts necessary for a build or migration. You need testing in order to validate the build, and to ensure that you're building the correct items in a safe fashion. You need instrumentation and measurements that will log the time of the build, the version being built and the success of the build, along with warning messages. Warnings and errors need to provide detailed diagnostics, if necessary, to quickly pinpoint the exact cause of the failure.

As a result, your database builds and releases will be frequent and punctual; you'll spot and fix potential problems early in the cycle; and you'll have the measurements that will prove just how much your processes are improving and how much higher the quality and speed of your development processes have become.

7 – Database Migrations: Modifying Existing Databases

It should be simple to upgrade a database to a new version. It certainly can be, but if you need to preserve the existing data and you have made changes to the design of the tables, then it can get complicated. If you are deploying changes to a heavily-used Online Transaction Processing (OLTP) system on which an organization depends, then you need to understand, and be familiar with, the issues that can affect a database migration. In this chapter, we look at the basic approaches.

Until a database is first released into production, there is no necessity for change scripts. It is generally simpler and more convenient to build a database from the DDL source-code components, and bulk load whatever data is required. From the source DDL scripts, plus the basic versioning functionality of the VCS, you can see at a glance the history of each table's evolution and who did what, why, and when. You can build the database from any release that is in version control and then use that same build for whatever purposes you require, such as developing, testing, staging, and deployment.

Database migrations, rather than simple builds, suddenly become necessary when a production database system must be upgraded in situ. You can't just build a new version of the database: you have to preserve the data that is there, and you need to do the upgrade in a way that doesn't disrupt the service. Migrations are useful in other cases too, for example if you need to repeatedly update development databases where the volume of test data is large enough to preclude the possibility of running frequent builds. In the deployment pipeline particularly, the released version will need to be tested with large data sets, and at this stage migrations often make more sense.

Where databases are reasonably simple, this process of modifying existing databases is very simple too; in fact, the process can be automated. Where databases get more complex, and become the hub of corporate information systems, then there are more details to check and precautions to take. Teams that delay tackling these issues until late in the development cycle find that changes build up to the point that deployments become daunting, and releases become infrequent.

This chapter gives an overview of some of the ways of making the migration process as painless and stress-free as possible. It discusses the mechanics of a database change process and describes the two basic approaches to managing changes (state-based versus migration-based). It offers some steps, starting with relatively straightforward improvements, which will lead toward a reliable, automated, and tested database migration process, and minimize the chance of any data integrity issues in the production database.

What is a database migration?

A database migration involves changing the database from one defined version to another. A **migration** script alters the metadata of a database, as defined by its constituent DDL creation scripts, from one database version to another, while preserving the data held within it. Migration scripts can be forward, or "up," to upgrade a database to a newer version, or backward, or "down," to downgrade to a previous version of a database. In order to promote an existing database from its current version to any other version in the VCS, we simply run, in the correct order, every "up" migration script in the set.

The migration process will use, as a starting point, the build scripts for the individual database objects held in version control. In fact, as described in Chapter 6, *Better Ways to Build a Database*, a "build script" is then really just a special case of a migration script that migrates the database from an empty database (as defined by `model)` to the required version.

The primary challenge when migrating an existing database is how to handle changes to the database tables. The migration script must alter any table in the database that has changed, while preserving data. The required change may be as simple as adding or removing a table column, or a complex refactoring task, such as splitting tables or changing a column in a way that affects the data it stores. Most changes are managed by SQL Server in such a way as to preserve the data; most, but not all. For example, let's say we need to "split" the zip code portion of an address column into its own column, to facilitate searches. This would require a three-stage migration script, first creating the new column, then moving new data into column, then cleaning the data in the original column.

Two approaches to database migrations

There are two basic approaches to migrating databases, one based on versioning object **`CREATE`** scripts, commonly referred to as the **state-based** approach, and one based on storing in the VCS only the initial **`CREATE`** scripts for each object followed by a string of object change (**`ALTER`**) scripts, often referred to as the **migration-based** approach.

There is no fundamental difference between the two approaches. Both use migration scripts to do the work. In essence, they are distinguished by whether we consider the source of any version of a database object to be that version's creation script, or the original creation script combined with whatever alterations to it took place to produce the current version.

The two approaches are not mutually exclusive. Tools used to migrate a database using the state-based approach occasionally need help defining the correct route for migrations that affect existing data; i.e. it may involve hand-crafting the change scripts required for a tricky migration. Tools that use the migrations approach often need a way to "back-fill" the state of each object, at each version, so that we can see easily how the object has changed over time.

State-based approach

When using the state-based technique, we store in the VCS the source DDL scripts to **`CREATE`** each database object. Each time we modify a database object, we commit to the VCS the latest creation script for that object. In other words, we are versioning the current state of each object in the database.

We then derive a migration script based on a comparison between the static definition of the database in version control (the object DDL scripts) and the target database. We will need to automate the generation of this migration script, often referred to as an **automated synchronization script**. Generally, we will generate this script dynamically using a schema-comparison engine (discussed in more detail later) and then a developer or DBA will need to check it, and perhaps edit it to deal with any tricky data migrations.

Sometimes it is impossible for a tool to create a synchronization script that will preserve the data correctly. Although it knows what the initial and final versions of the database look like, there is sometimes ambiguity regarding the correct transition path. For example, if we rename a table, the tool is likely to miss the fact and simply drop the existing table and create the one with the new name, thereby failing to preserve the data under the newly-named table.

Of course, there are other changes, such as splitting or merging columns or changing the data type or size of a column where the tool may not have enough information to be certain of where the data should go.

These situations require human intervention. We may need to manually alter the synchronization script to cope with difficult migration problems, and define manually additional change scripts that describe how existing data should be migrated between versions of the database. However, some automation tools now make handling this much easier than in the past.

With each code-based object in the database, we can simply store in the VCS a script that deletes the current object if it exists, and re-creates the changed object. However, with this approach we need to reset permissions on the object each time. A better approach is to store in the VCS a script that creates the object only if it doesn't exist, and if it does exist, alters it so that existing permissions on that object are retained (although, under the covers, the object is still dropped and re-created).

Migration-based approach

This technique migrates a database by means of one or more **change** scripts. Any DDL script that alters one or more existing database objects is a change script. For each migration between consecutive database versions, we need to store in version control the initial build scripts, plus change scripts that describe precisely the steps required to perform the necessary changes to migrate to the required database version. For table changes that involve data migrations, the object change scripts include extra logic that defines how existing data is migrated between changes in the organization of tables. We adopt the same change-script-based strategy for tables and code objects, which is useful; but arguably this approach is redundant for code objects that don't hold data and are entirely overwritten on every change.

SQL-based versus code-based migrations

This discussion assumes we store SQL change scripts in the VCS. However, some tools use code-based migrations, which define the required database changes in the application code language, such as Java, C#, or Ruby. We'll cover a few of these tools briefly, a little later.

To move a database from one version to another or to build a database, you can use a tool that builds the migration script from a series of change scripts. It does this merely by applying this set of immutable change scripts to the target database in a predefined order.

Often the order of execution of the SQL scripts will be determined by the file naming, for example using sequential numbers at the start of the name (`00000_2015-11-14_CreateDatabase.sql`, `00001_2015-11-14_CreateTableOrders.sql` and so on) so that it is simple to determine the order in which they need to be run. An alternative scheme uses the date and time at which the script was generated (www.phpassionate.com/2011/08/22/changing-dbdeploy/) to create a script sequence that still has ordering but does not need sequential integers, for example `20151114100523_CreateDatabase.sql`. Here, the ordering is defined by a concatenation of year, month, day, hour, minute, and second in 24-hour format. The latter scheme works better with source-code branching because the ID number is not needed prior to the script being created. However, the ordering is more difficult for humans to read, and there is a chance that migrations may merge fine technically but not achieve the desired intent (just as with any code merge).

A change-script-based approach generally relies on a metadata table held in the database itself that captures the database version and keeps track of which scripts have been run for that version. At the least, it stores the date of the migration, and the script version/name, but some frameworks or tools store additional information too, such as a description of the migration, the author of the script, the length of time taken to run the script, or a hash of the script file contents, so that any accidental changes to a script can be detected.

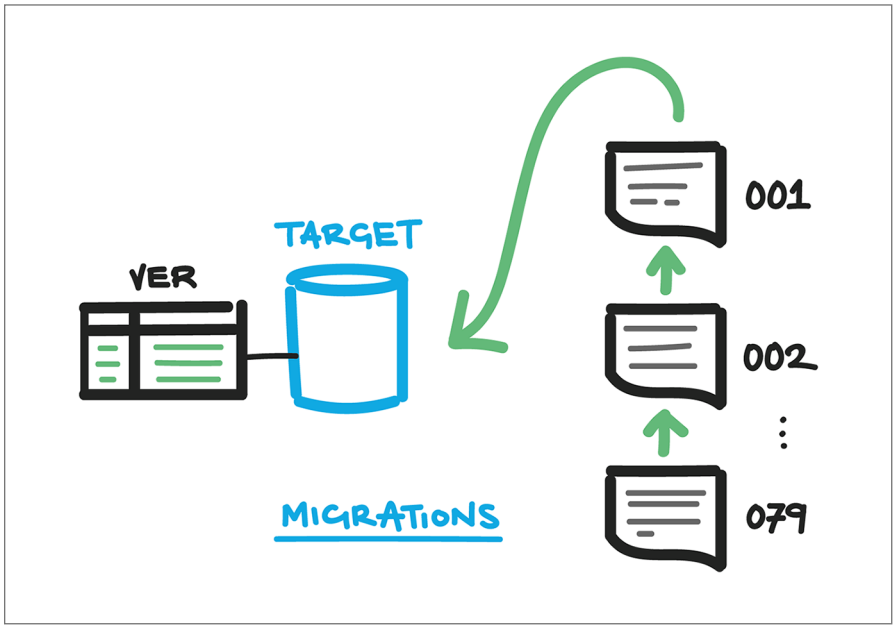

Figure 7-1

When running a migration, the framework or tool checks to see which script was last run by accessing the metadata version table, and then runs only those scripts that are numbered sequentially after that version. The versioning metadata also provides traceability for database changes, which can be essential for regulated industries such as banking, finance, or healthcare.

Choosing a DLM change automation approach

Which approach you choose will depend on a number of factors, such as how well either approach integrates with your existing approach to deploying application changes, and how frequently the team make significant changes to the initial design of the base tables, as well as other factors relating to the team's capabilities, and overall approach to database development.

As discussed earlier, the approaches are neither fundamentally different nor mutually exclusive. While a single, unified approach is usually preferable, a team or organization may need more than one approach for different parts of the system or at different times in development. With the right tools, either is viable.

We'll discuss just a few of the technical factors that may influence your choice of approach, but there are other factors to consider too, such as team discipline and collaboration. For example, a change-script-based approach that works well with a single team or two closely-related, highly-collaborative teams might prove unworkable with three geographically dispersed teams that belong to different organizations that cannot collaborate as effectively.

Frequency of significant database changes

The state-based approach works very well if the team starts development with a robust ER design, with relatively minor tweaks during development. In projects with frequent-release cycles, but where the team's understanding of the data domain continues to evolve even *after* the first deployment to production, it's likely that significant database changes will frequently form part of database migrations. In such cases, the state-based approach can begin to feel like a disjointed approach in development projects where many of the required database changes require custom change scripts.

If the team makes frequent significant changes to tables as they gain an understanding of the data, then the change-script-based approach is likely to be a better fit. This is because it allows the team to exert granular control over all database modifications, providing they have tools to support and automate the process of constructing, tracking, and executing correctly the individual object change scripts.

If the migration scripts are tested frequently, the development team gets closer to the goal of having "push-button" database deployments to production, or at least directly to staging prior to final checks, from a bunch of change scripts.

Bear in mind, though, that if every change must be reflected in a new script, since existing scripts are immutable, then if you make a mistake, you must write a script to undo it, then write a script to do it again, the right way. As a result, your database change process is doomed to repeat every wrong turn.

For large, frequently-changed databases you may end up with a vast number of individual object change scripts that may need to be run. Some change-script-based tools, such as ReadyRoll (covered later), recognize that some teams, at certain points, would like to "rebase" and simply have the tool auto-generate a change script that effectively makes redundant all the separate object change scripts.

Visibility into database state

Having a well-documented build script in the VCS for every object makes it very easy for the team to see the exact state of an object at any given time, and to understand how changes to that object are likely to affect other objects. This is arguably one of the biggest advantages of a state-based approach to database migrations. Having the latest state of the database schema in version control, in the form of the latest set of object-level scripts, is also a natural fit for the standard version-control processes that provide the audited history of changes

By contrast, the change-script approach is not such a natural fit for standard source control processes. If every table change is a new immutable script then there is effectively nothing to version. However, it does allow very granular control over every change.

Some teams attempt a best-of-both approach, maintaining both the current state of each object, and the required sets of change scripts to move between database versions. The obvious danger with this approach is that there are now two points of truth, and the very real possibility that they will become out of sync. What if creating an object according to the latest **`CREATE`** script doesn't give the same result as running the specified set of **`ALTER`**

scripts on the existing build? The obvious way of getting around this is to do the initial build by the sequence of change scripts and then script out the objects individually into source control. SMO provides an easy way to do this via scripts but it is tricky to get it completely accurate. Some migration tools will do this automatically for you.

We can check accuracy of this back-fill process by upgrading a target database by the migrations approach, using a set of change scripts, and then using a schema-comparison engine, such as Redgate SQL Compare, to compare the resulting database to the set of object-level scripts in version control. If they are the same, then the build or migration is valid and there is no drift.

Support for branching and merging

As discussed in Chapter 5, *Database Version Control*, the DLM approach advocates that the development team avoid using branches in version control wherever possible, and try to keep short-lived those branches that are required.

Nevertheless, many teams rely on branching strategies to streamline their development and release activities and, generally, the state-based approach tends to be a better fit. When we are versioning the object DDL scripts, and developers make conflicting changes to the same object, in different branches, then the subsequent merge attempt will raise the conflict and the team can deal with it.

With the change-script approach, and every change is a new script, there is a very real chance that one developer will simply overwrite another's changes. If you are maintaining multiple branches, the team faces the prospect of painstakingly going through every script, line by line, to work out the order that they need to run in or whether there are conflicts. With a migrations-based strategy, team discipline is exceptionally important and managing the order of upgrade scripts is a very important task.

From chaotic to optimized database migrations

In organizations where the approach to database change is essentially "chaotic," the database release process will often consist of the development team depositing a bunch of scripts in a folder shortly before they wish to deploy. This is the throw-it-over-the-wall approach to database releases.

The scripts will likely be metadata scripts, describing only the required changes to the database tables and programmable objects. As discussed previously, the migration script may take the form of a list on individual object change scripts, or an auto-created transactional synchronization script to update the existing database from the source object scripts.

In either case, there are likely to be places where radical table modifications require hand-crafted migration scripts. It can take the operations team many days to craft migration scripts that they are confident will preserve existing production data exactly as intended, won't cause instability in their production systems, and will fail gracefully and roll back if a problem occurs. It all has to be written and tested, it all takes time, and it can delay delivery of functionality to the business.

If the development team has a justifiable need to put changes into production (or at least staging) frequently and rapidly, then this system is no longer appropriate.

The answer is either to deploy less frequently, or to push responsibility for data preservation in the migration scripts down into the development team. It might be a daunting prospect for a DBA to cede control of database modifications, as part of a new deployment, to the development team. After all, the DBA is responsible for the preservation of the existing production data, and for data integrity issues or service disruption caused if things go wrong. Indeed, it is only likely to be successful as part of an automated, DLM approach to database change, undertaken with close collaboration between development and operations teams, according to the DevOps model.

This has several implications. It means that the development team must grasp the nettle of database continuous integration (see Chapter 9), in order to test these migrations sufficiently before they get anywhere near release. It also means that they would have to perform tests that prove that the migration scripts work, while the system was working with a realistic data and load. It means that many governance checks will need to be done in parallel with development, rather than by sticking with the familiar chaotic procession down the deployment pipeline. The whole change process will need to be instrumented, as discussed for the build process in Chapter 6, so that the team tracks all the information required in order to spot problems quickly and continuously improve the process.

The following sections suggest some staged improvements you can make to your database change-management processes.

Ensure scripts are idempotent

Every migration or change script should be idempotent; it should have no additional effect if run more than once on the same database. A migration script takes you from one version of a database to another, however many times it is run.

A standard database scripting practice is to include guard clauses (see Jonathan Hickford's article at http://preview.tinyurl.com/y75wrhw7) that prevent a change script making changes if it doesn't need to be or can't be run, or to abort a script if it determines that it has already been run, or to perform different logic depending on the database context. Alternatively, some automation tools will ensure that a migration script is not run more than once on a given database version.

Ensure scripts are immutable

A migration (or change) script is immutable. Once it has been written, tested, and any bugs fixed, we should very rarely change it. Instead, we add extra scripts to perform schema changes or data updates.

Often a migration script needs to resolve tricky data-migration issues. If a script is never changed once it has been run successfully and has been proven to work, then we only need to solve each data-migration problem once, and we can then run exactly the same set of migration scripts on every server environment that requires the same changes.

This helps to ensure that we can upgrade any database "in the wild" without needing to inspect it first. As long as the database objects have been changed only via the migration tools, it is safe to run the migration scripts against it.

Guard against database drift

A migration script is valid only for the database versions for which it was intended. Before we run any database migration, we need to be sure, both that the source represents the correct database version, and that the database after the migration is at the expected version. In other words, before it does any modification, the database migration process must check that the database is really at the correct version and that no uncontrolled changes have caused the version to drift.

If the target is at a known version, but not the version we anticipated, then we need to use a different change script, or set of change scripts. If the target is in an unknown state, then it has been altered outside of the final change process; this is known as "database drift." If we run the migration regardless, it may delete the out-of-process change so that the work may be irretrievably lost. If that is a risk, then this is the time for the anomaly to be investigated; otherwise the migration should go ahead and obliterate the drift.

Many teams, regardless of whether they adopt a state- or change-script-based approach to database changes, will often use a state-based comparison tool for database drift detection, giving the team early warning of unrecognized changes to databases in different environments. The comparison tool keeps track of a set of known database states (or versions, assuming every known state is given a version), and periodically checks that the target databases match one of the known states.

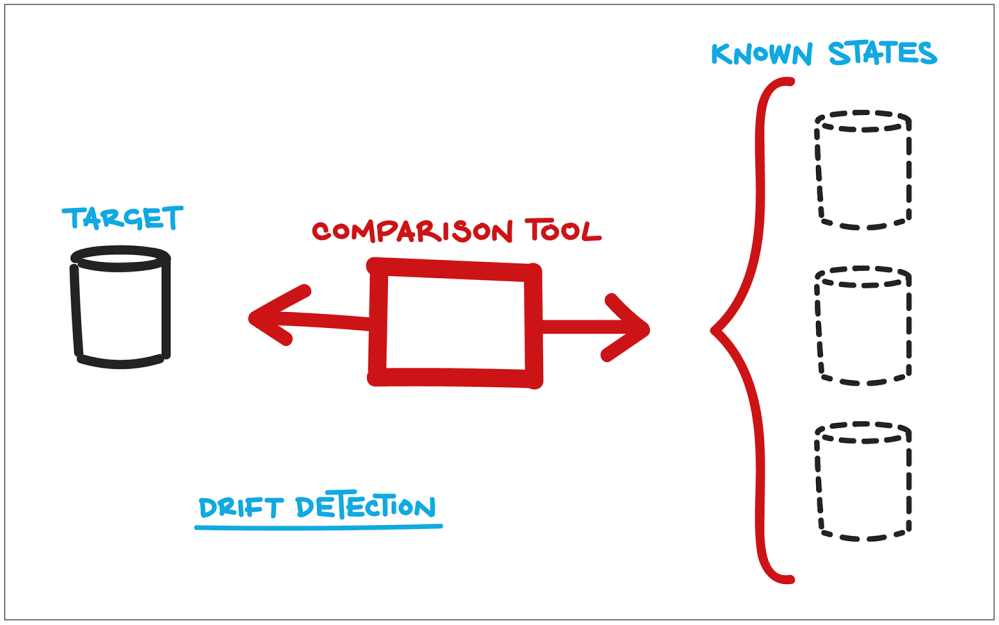

Figure 7-2

State-based tools typically check the database state in every environment before applying changes. For situations where database changes are applied in several different or even haphazard ways, state-based tools provide a high degree of confidence that database changes will be executed correctly, even when the nature and the source of the changes are unknown or unpredictable.

Perform early and frequent testing

We need to write integration tests during development in order to check that the migration scripts worked exactly as intended, and the final database state, including the data, is exactly as we expected. The development team, ideally, need access to a "production-parallel" environment, often Cloud-based, where they can perform these tests with realistic, production-like data, and under realistic data access loads. See Chapter 9, *Database Continuous Integration* and Chapter 8, *A DLM Approach to Database Testing* for further discussion.

Fail gracefully

A DLM change process must never result in a partial build or migration if something goes wrong. We need to be able to revert to the starting version if an error occurs during the migration process, or to revert to a previous version if the error becomes apparent after the migration. Any DDL change can be run in a transaction. This means that, with correct handling of errors within the transaction, the script can be rolled back, leaving no trace. Some tools are able to generate a migration script that handles this properly.

There is really no satisfactory alternative to this technique. It is possible to create a script that checks for the existence of any of the alterations and undoes them, but this is difficult to achieve, and is generally only appropriate in an emergency once the transaction within which the migration script has run has been committed. Generally, DBAs prefer to "roll forward" a failed migration using a hand-cut script.

Introduce governance checks early in the cycle

Databases are migrated from one version to another for a wide variety of reasons, so governance will have a wide range of interest in the process for many different reasons.

A change-script-based approach is generally a concern for governance unless the state of each database object can be recorded for each release. There needs to be a record of when each change was made, and why. In a change-script-based approach, each change usually has its own script, but the context of the object is difficult to work out from this. A change of a datatype will only record the new type in the migration, not the old one. An auditor needs to be able to see quickly the extent of the differences between two consecutive versions, in

order to check that the changes have no implications for audit. A data expert may need to check the context of a database under development to determine how it fits into the company's master data file or model. A project manager will need to determine from a release whether the progress of a development is going according to plan.

Assuming we've enforced immutability of a migration script, it will be used for all migrations between the two versions for which it was designed. This means that, whatever method of reporting it uses to record the date of migration, the name of the server, and whether it was successful, should be applicable to all contexts in which the script is used.

A brief review of automation tools for database migrations

Regardless of whether you prefer change-script-based or state-based database migrations, making database changes can at times be risky and error prone, unless the processes are well-controlled and automated. Thankfully, there is a growing range of software tools we can use to help automate the deployment and testing of database changes, regardless of whether we opt for a state-based or change-script-based approach.

Automating state-based database migrations

It is a hard task to construct a migration script manually, based on a comparison between the object DDL scripts in source control and the target database. We would need to tackle all the issues of building new objects in the correct dependency order, as described in Chapter 6, *Better Ways to Build a Database*, plus the need to preserve data. Your only real option if using a manual approach based on object-level scripts, may be to take the database offline, read out all the data, via for example the bulk copy program utility (bcp), build the new database version, and read all the data back in. However, this is only viable for small databases, and manual migration techniques quickly become laborious as the size and complexity of the database increases, and as more people become involved in the task.

A DLM approach requires automation of this process and most teams use a schema-comparison tool to do this job for them. The comparison engine will inspect the target database and the source DDL scripts and packages in the VCS, compare the two, and

auto-generate a synchronization script that will **`ALTER`** a target database at version *x* and change it to version *y*, while updating the living database definition and preserving existing data. The comparison tool can run this change script against the target database to make the required changes. The comparison tools generally have various options for excluding or including different kinds of database objects (procedures, functions, tables, views, and so on) and some have support for data comparisons, allowing new reference data (such as store locations or product codes) to be updated in the target database.

While this migration process still relies on the object-level DDL scripts in the VCS as the "source of truth," the team needs to test and archive every auto-generated migration script that has been used to build a database in this way.

There is now a range of tools and frameworks that use the state-based approach to DLM changes. We're only going to cover a few of these automation tools in any detail, but we also provide a section that lists out other candidates.

Redgate Software

Redgate (www.red-gate.com) provides a variety of tools for SQL Server, Oracle, and MySQL, aimed at database change and deployment automation, including SQL Source Control, DLM Automation (automated build, testing, and release control via Octopus-Deploy), and DLM Dashboard (for tracking database changes through environments).

SQL Compare, the comparison engine behind these tools, sniffs out your directory of object scripts, including all subdirectories, parses what it finds and creates an internal model (network graph). It takes the scripts in any order. It compares this internal representation with the database, backup, or script directory. What emerges from all this work is a change script that is not only in the correct dependency order, but is also transactional, so it doesn't leave bits of a failed migration in place; and it then builds the objects. To avoid the possibility of creating circular dependencies, it delays enabling constraints when necessary.

The SQL Source Control tool can handle the use of migration scripts where the correct transition can't be determined. It allows custom change scripts to be substituted for the auto-generated migration script, at the object level. The system checks whether there is a custom change script for relevant database object(s) to convert from one version to another, within the range of the migration being undertaken, and if so, this overrides the automatic script at that point. This facility has been in the product for some time but has recently been thoroughly re-engineered to make it robust enough to handle all known requirements.

Microsoft SQL Server Data Tools (SSDT)

SSDT (http://preview.tinyurl.com/pj9st9e) is Microsoft's declarative approach to database change automation. It is aimed at developers using Visual Studio, and has tight integration with other Microsoft tooling, particularly tools for BI. Some teams successfully use SSDT in combination with other tools (Redgate, tSQLt, etc.) where SSDT lacks features (such as reference data management).

SSDT uses Microsoft's DacFx technology, and DacPacs, (http://preview.tinyurl.com/pbdqs9l) to auto-generate database change scripts. Unfortunately DacFx and DacPacs can't help much with the problem of migrating a database automatically from object-level scripts, because you have to append change scripts to the DacPacs in the correct dependency order.

DacPacs produce a SQLCMD change script, which accepts parameters to provide values for such properties as the location of the database files on the target server. DacPacs can't compare a database to the scripts in the VCS.

Altova Database Spy

Altova Database Spy (http://preview.tinyurl.com/zu33rka) provides a database schema-comparison tool that can produce SQL **`ALTER`** scripts to synchronize one database from another and save database comparisons for later use. There is also a tool for data comparison and merging.

The tools support a wide range of databases, including SQL Server 2000 to 2014, PostgreSQL 8 to 9.4, Oracle 9i to 12c, DB2, and Sybase.

DBMaestro

DBMaestro (http://preview.tinyurl.com/jlboze4) provides tools for SQL Server and Oracle, with a strong focus on policy enforcement.

Devart dbForge

dbForge from Devart (http://preview.tinyurl.com/jepmaf6) supports SQL Server, MySQL, Oracle, and PostgreSQL databases. It provides the ability to compare both schema differences and data differences and has a command-line interface for automation.

Others

AdeptSQL tools: http://www.adeptsql.com/

ApexSQL: http://www.apexsql.com/sql_tools_diff.aspx

Datanamic: http://www.datanamic.com/datadiff-crossdb/index.html

DBArtisan: http://www.embarcadero.com/products/dbartisan

DBComparer: http://dbcomparer.com/

OpenDBDiff: https://opendbiff.codeplex.com/

SQLDelta: https://www.sqldelta.com/

Automating change-script-based database migrations

For small or simple databases, a "hand-rolled" solution can work well enough: the very first script sets up the metadata versioning table in the blank database, and early subsequent scripts build out the schema and add static/reference data. With a hand-rolled approach you have the complete confidence that you are in full control of the SQL scripts and therefore the database.

However, as the database grows in complexity or other colleagues or teams need to change the database, it can be useful to adopt a tool to help with some of the heavy lifting. In particular, tools will tend to have solved most of the tricky migration activities and the logic associated with the migrations mechanism itself (that is, the checking of the metadata version table and working out which migration scripts to run).

The following sections describe briefly some common database migration tools and frameworks, all of which use SQL change scripts.

Dbdeploy

Dbdeploy (http://dbdeploy.com/) is a veteran, well-tested database migration tool used in much commercial and open-source software. Dbdeploy has versions for Java, PHP, and .NET.

Flyway

Flyway (https://flywaydb.org/) is an open-source database migration tool that strongly favors simplicity and convention over configuration. Flyway has APIs for both Java and Android and several command-line clients (for Windows, Mac OSX, and Linux). Flyway uses "plain" SQL files for migrations, and also has an API for hooking in Java-based and custom migrations. Notably, Flyway has support for a wide range of databases, including SQL Server, SQL Azure, MySQL, MariaDB, PostgreSQL, Oracle, DB2, and others.

Liquibase

Liquibase (http://www.liquibase.org/) is a database migration tool that has strong support for multi-author changes and branching, and targets a wide range of databases including MySQL, Oracle, SQL Server, DB2, and H2. Liquibase migrations can be defined in SQL, XML, YAML, or JSON, and it has tools that generate SQL scripts in order to address some of the "people and process" complexities (http://preview.tinyurl.com/h3yvfdv) of DLM. Liquibase also has explicit support for SOX-compliant database changes (http://preview.tinyurl.com/j5v6z4s) via automatically- or manually-generated rollback statements.

MyBatis

The MyBatis migrations framework (http://www.mybatis.org/migrations/) draws on the techniques of both Rails Migrations and dbdeploy.

ReadyRoll

ReadyRoll (http://preview.tinyurl.com/hro28u3) offers a pragmatic workflow for making database changes that combines online database changes (making changes directly to a database) with sequential SQL scripts. This makes it particularly suitable for working with existing databases that may not have been created programmatically and also for working in mixed teams of DBAs and developers, where some people may make changes directly in databases, and others may write SQL scripts.

ReadyRoll inspects online databases and generates sequential scripts via a sync tool, using version control diff and merge, and the experience of DBAs and developers, to determine which changes to merge, and how.

It also supports a feature called Offline Schema Model (http://preview.tinyurl.com/jpuxuta) which will "back-fill" the object scripts for a database version, to enable you to audit changes to your database objects.

Automating code-based migrations

Whereas SQL-based migrations use native SQL code to define and control database changes, code-based migrations define the required database migrations in the application code language, such as Java, C#, or Ruby. Tools that automate code-based migrations will often generate the SQL code automatically from the annotated Java/C#/Ruby code. This makes code-based migration more natural for many software developers although not so accessible to DBAs who are less familiar with these languages.

Example automation tools

ActiveRecord Migrations (http://preview.tinyurl.com/lbpgxd4) – a Ruby migrations framework, works with the Ruby implementation of Martin Fowler Active Record data access pattern (http://preview.tinyurl.com/z5fpyjs). Note that the Active Record way claims that intelligence belongs in your models, not in the database, so migration frameworks based on Active Record are likely to be unsuitable for situations where referential integrity needs to be maintained by the database.

Entity Framework (http://preview.tinyurl.com/hkwxrwf) – Migrations are written in C# and can be exported as SQL for review by a DBA. Starting with EF6, the EF Code-first Migrations generate idempotent scripts (http://preview.tinyurl.com/h3j67yq), making them safer to run in different environments.

FluentMigrator (http://preview.tinyurl.com/z8oyly6) – an open-source .NET database migrations framework with support for many different databases. Installation is via a NuGet package, and migrations are written in C#. Each migration is explicitly annotated with an integer indicating the order in which the migrations are to be run. There are several ways to run the migration scripts (http://preview.tinyurl.com/z2rwxvy), including command line, NAnt, Rake, and MSBuild, which makes for useful flexibility in testing and deploying using FluentMigrator.

Phinx (https://phinx.org/) – a database migrations framework for PHP with support for MySQL, PostgreSQL, SQL Server and SQLite.

Advanced database migration considerations

As for database builds, there are many choices to be made as to how database migrations proceed in case of various eventualities. Below are just some of them.

Should the database be backed up before the update is made?

If a migration is done as part of a deployment process, then a number of precautions need to be taken. A database backup is an obvious backstop-precaution, though if a deployment goes wrong, the restore-time can be significant. However, many people have been grateful that the precaution was taken when a deployment has gone badly wrong.

Can we stop users accessing the database during the upgrade?

We may require non-breaking online deployments, but these will risk locking and blocking on large tables (for example, a table column **`ALTER`** is likely to lock the entire table). Some migrations need the application(s) that use the database to be quiesced, or need the database to be in single-user mode, and so the script will need to check whether this has happened before they proceed.

Do we alter change-data-capture objects when we update the database?

Change-data-capture involves capturing any changes to the data in a table by using an asynchronous process that reads the transaction log and has a low impact on the system. Any object that has change data capture requires the sysadmin role to alter it via a DDL operation.

Do we drop existing constraints, DDL triggers, indexes, permissions, roles, or extended properties in the target database if they aren't defined in the build?

Where operations teams are maintaining a production system, they may make changes that are not put into development source control. This can happen legitimately. Normally, changes to indexes or extended properties should filter back to development version control, but in a financial, government, or healthcare system, or a trading system with personal information, there must be separation of duties, meaning that database permissions, for example, have to be administered by a separate IT role, and the source stored separately in a CMS. To apply a migration to a target database that has separation of duties, permissions and role membership have to be ignored. DDL triggers that are used for audit purposes are sometimes used to monitor unauthorized alterations of DDL, and so obviously cannot be part of development source control. In this case, the migration scripts need to ignore certain types of objects if they are outside the scope of development.

How do we deal with NOT NULL columns for which no default value is supplied?

If the migration involves updating a table that contains data with a column that does not allow **`NULL`** values, and you specify no column default, should the build provide a default to make existing null data valid?

Developers sometimes forget that, if they are changing a **`NULL`**-able column to a **`NOT NULL`**-able constraint, all existing rows with a **`NULL`** in that column will be invalid and will cause the build to fail. Sometimes a build that succeeds with development data then fails when the build is used to deploy the version to production. Of course, the code needs to be altered to add a **`DEFAULT`** to a non-null value to accommodate any rogue rows. However, some developers want the build to do this for them!

Summary

Databases come in all shapes, sizes, and requirements, and so, too, should the way we deploy these changes. Database builds and migrations aren't necessarily complicated, but merely reflect the complexity of the database.

Whether you choose a state-based or change-script-based approach to managing database upgrades, you will end up using a migration script. A development team using a change-script-based approach, sees the change scripts that make up the migration as the canonical source of the database, whereas the team using the state-based approach sees the **`CREATE`** scripts in version control as the canonical source, and uses them to derive the migration script.

8 – A DLM Approach to Database Testing

When deploying database changes, automated testing is critical to a successful DLM approach. You need to know that a database change does what you want it to do and, more importantly, that you have not introduced any "bugbears" or other unwanted side-effects.

Database tests need to prove that the database always meets the requirements defined by the tests. We need database unit tests, to prove that the individual units of code always function as predicted, and return the expected data. We need integration tests to make sure that all the required units work together properly, to implement business processes. We need tests to ensure that the database application performs and scales to requirements, and conforms to all security requirements. As much as possible of all this needs to be automated, and ideally "left-shifted" so that such testing is incorporated in your database continuous integration process (see Chapter 9, *Database Continuous Integration*).

The point of database testing

The DLM approach to database testing requires that the development team establish a working version of the database very early in the development cycle, and then build a suite of database tests that will verify that it remains in a working state, as they continue to build and refactor the database schema and code. In addition, they need tests to confirm that the application always meets its requirements in terms of performance, scalability, and security, and that no change they make causes "regressions" in any other part of the application, or in any other connected systems.

Some of this is involves classic "black box" testing of the database interface. For example, does a stored procedure or function always returns the expected data, in the expected structure (correct column names and types), for the whole range of data or parameter values that a user can supply? When inserting or modifying data, do the correct data values always get persisted?

We also need "internal" database tests to ensure that no change made during development, and no action the user can perform, will violate data integrity. The tests will need to confirm that the **`PRIMARY KEY`** and **`FOREIGN KEY`** constraints enforce referential integrity correctly,

and avoid orphaned data values, and that **`CHECK`** constraints and other rules exist to ensure that no part of the application can insert or modify data values such that they fall outside the valid domain of values defined by the business. Using automated testing, we need to prove, not only that a database persists the correct data, but also that it rejects incorrect data from being stored.

In addition, database load and performance tests must demonstrate how the application will behave under real, concurrent access conditions when other transactions may be updating data in a table at the same time as your transaction wants to read it.

Finally, the testing regime must, as far as possible, account for complex interdependencies that exist in real, enterprise database systems, where the same table may be accessed by several OLTP applications, as well as a reporting application, ETL processes, replication processes, change capture processes, and so on. Often, one seemingly small change to a data structure during development can have myriad potential consequences in the enterprise database system. Tests must be in place to catch such "breaking changes" as early as possible in the development cycle.

Special challenges of database testing

This section briefly summarizes just a few of the key challenges of testing databases effectively, compared with "normal" application testing.

Tests need to verify database migrations

A typical application consists of pre-compiled binaries. Replacing an old version of code with a new version means shutting down the application, making the switch, then restarting. This is not generally possible with a database. Instead, you need to migrate the state of the existing database to match the desired state, without compromising or losing any of the existing data.

Whether you write database migration scripts manually, or use a tool, we must test them thoroughly to make sure that the final database state, including the data, is exactly as we expected. The only way to make rapid changes safely is to have automated tests verify that the post-change database state is exactly as expected, and therefore that the change is safe.

Tests need to account for complex database interdependencies

When working on data and code in a database, the developer is effectively working on a set of public objects, much like working with a public API. While, hopefully, permissions are in place to restrict access to base tables, the fact remains that, in many cases, database objects including tables can be accessed by several applications, including a front-end website, an ETL process, ad hoc reporting by users, and so on.

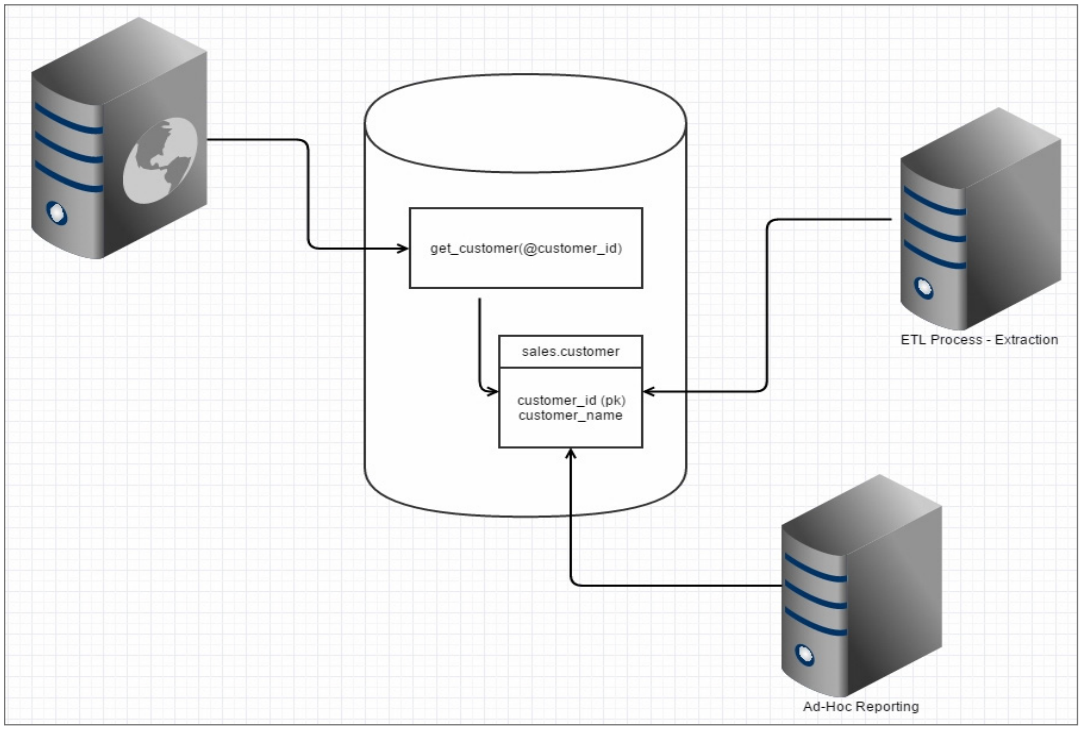

Figure 8-1

In the example shown in Figure 8-1, any change to the `sales.customer` table could potentially break one of the dependent processes or applications. Each "unit" in the database, each table, stored procedure, function, and so on, needs to have a set of "covering" tests, such that if any change to the structure of a table, or the code in procedure, breaks some other part of the system, then one or more of the tests will fail, and the team will be made aware of the problem immediately.

For example, let's say we have a unit test that verifies that when we pass in a **`customer_id`** to the **`get_customer`** procedure, we get back the one customer we expect. Effectively, this test puts in place a contract that says:

- When we call **`get_customer`**
  - We pass in a **`customer_id`**
  - We want back the one **`customer_name`** that **`matches_that customer_id`**

If any change, made anywhere in the system, breaks this contract, then the unit test will fail and the developer knows, there and then, that they will need to fix that before moving on. Ideally, the build system for the database would also have tests that cover each of the interfaces used by any dependent systems. If an ETL process reads 10 columns from the **`Customers`** table, there should be a test that reads those 10 columns.

By building up a suite of such tests, any developer who uses the database can be sure that their changes will not affect the rest of the system, without having to have an in-depth knowledge of the entire system. This has the added advantage that developers can work on areas of the system about which they do not necessarily have an in-depth knowledge; this reduces code ownership and the need to defer all work to a subject-matter expert.

For enterprise databases, the dependencies get more complex, and each change to a table has potential impact on many dependent objects and processes, both at the database level (replication and CDC processes, indexes, triggers, and so on), as well as at the server level (jobs, linked servers, database mirroring processes, and more). While larger databases are going to require a significant investment in the test code, it will always pay dividends over manual testing, in terms of the time it takes to test the risk associated with each release.

Effective database testing requires real data

When we write a function or method in application code, we know exactly how this code will be called, exactly how it will be executed when called, and therefore how it will behave when called concurrently from multiple end-user processes. It also means we can reason more easily about the likely impact on other processes running concurrently on the machine.

Database programming is declarative. The SQL programmer does not decide how the relational database engine executes the submitted code, only what data set we would like that code to return. A component of that engine, called the optimizer, devises an **execution plan**

that sets out exactly how the execution engine should execute the code. The optimizer examines the text of the SQL we submit, and the available database structures, and uses statistical knowledge of the volume and distribution of data in the database to determine how many rows are likely to be returned, and to make decisions such as:

- which indexes will be used to access the data, and how
- where data needs to be joined together, the method of joining data
- how data is filtered and ordered and aggregated.

The execution engine executes the submitted query, per the optimizer's plan, takes care of reading and writing data to memory and disk, and controls what locks need to be taken, and so on. If our queries take a long time to complete, monopolize processing time, and/or cause locks to be held longer than necessary, then it means, not only that our code will perform and scale poorly, but also that it will "block" and disrupt other processes attempting to access the same data.

All this means that how our SQL code executes depends not only on the logic of our data access methods, but also on the optimizer's choices, which in turn depend on its knowledge of the data, and on the volume and profile of the data.

Traditionally, the database is "mocked" during data access testing, though there is an increasing understanding of the benefits of testing with realistic volumes of "real" data, as early as possible in the development cycle so that the team can properly understand the execution characteristics of their code. This also allows developers to gain a proper understanding of the data their applications generate and use, and its inter-relations, and to build tests that accurately reflect the true execution characteristics for their code, for whatever unexpected or "outlier" data values their applications may provide.

The need for repeatable testing

Without repeatable testing, we would need to ensure that the entire system still worked correctly after every change or set of changes. So, the time it takes to test database changes is a combination of:

- The complexity of the system.
- The complexity of the database.
- The complexity of the change.

- The knowledge of the system, (in that if the developer and tester are SMEs on the database then they will have a higher confidence in knowing what to test).
- The knowledge of the data. Will they easily be able to tell if data is incorrect or corrupt?

Considering the potential complexity of an enterprise database, with all its upstream and downstream interdependencies, it's clear that manually testing each change would be a time-consuming process. Similarly, if the team make a whole bunch of changes then manually test everything to ensure no regressions, then testing becomes laborious, and it's harder to track down what change caused the problem.

There will always be the need for some manual database testing, but developers will generally want to automate their tests as far as possible, and integrate them into the build and deployment process. There ought to be automated unit tests for every routine, tests for data quality and integrity, integration tests for processes, which include performance tests, scalability tests, security and access-control tests and many others.

With automated tests in place, during testing the developer and tester simply need to work together to decide:

- Can the change be unit-tested?
- Can the change be integration tested?
- Can the change be acceptance tested?

If none of these, or the change is high risk, then manual testing may be required.

The payoff for developers

It takes sustained effort to build, automate, and maintain the unit tests, integration tests, performance tests, and so on, required to test a database effectively. Even more effort is required to be sure that, for any database change, these tests capture all the effects on other components and processes in the database system. So, what are the rewards?

Immediate feedback

One of the most obvious is that an effective and automated testing gives the developers immediate feedback on any change they make. They will know what change caused what problem, and they can fix it quickly. Developers first write the tests which will prove that a small new piece of functionality works. They then implement the code to make the tests pass. When the tests pass, they commit the code to the shared version-control system, confident that it can be merged without causing bugs that will disrupt the work of other developers.

All of this reduces the time it takes to develop and test changes, meaning that changes can be deployed more rapidly, which is ultimately the aim of taking the DLM approach to database changes. It gives the team more confidence to make changes, to refactor or improve individual components. Over time, it gives the organization more confidence that they can allow database changes without fear of disruption.

Faster bug-fixing in production

Automated database testing also reduces the time to fix any bugs that are discovered once the application is in production use, especially if the automated testing is integrated into the DLM pipeline. In response to a reported bug, the team run an automated build of the latest version of the database, provision it with data, reproduce the bug conditions, write failing tests, and then implement and document a fix. The new tests ensure that the bug cannot recur without immediate detection.

A better understanding of how to write maintainable code

The additional payoffs, in my experience, come from adoption of a "test-driven" approach to database testing. When a developer writes database tests, especially when they write tests before the code, they must think, not only about the logic of the function or procedure they need to implement, but also:

- How each type of object in the database is used and how they interact with each other.
- How to specify any required parameters, and what data should be returned, and in what form.
- How to set up the data that the test uses to prove that the code is correct.

Writing tests forces developers to think beyond writing the code, to how the code will be used, tested, and maintained. They must understand how the code they write will interact with the data in the database.

Simpler database code

The need to think about how to write simple tests that will prove their code works generally leads developers to write simpler database code, each piece with a singular, specific purpose. For example, let's say a developer needs to devise code that will:

- retrieve a set of customer records showing their daily spending
- run a calculation on those records to get the average daily spend
- add an entry to an audit trail containing the result
- return the result.

The temptation might be to simply write a single stored procedure to do all the work and return the results. Consequently, you'd then need to write a test that also does a lot of work, to prove the right records are returned, that the calculation is correct, that we always get an entry to the audit trail. If the failing test does lots of things, it is often hard to see what exactly caused the failure.

However, if we think about how to write simple tests that verify just one thing, we tend to break the task down into manageable units, for example writing a procedure to return the result, another to add an audit log entry, an inline function to perform the calculation, and so on. Each piece of code has a single purpose and so, therefore, does each corresponding test, simply needing to verify that each piece of code does exactly what it is supposed to do. Being able to test each piece of code in isolation means that we also often get much more maintainable code as, instead of having a single procedure to do everything, we can separate our code and reuse it in other places.

Types of developer tests

When writing automated database tests, it is important to understand the different types, and where some are more useful than others. The best database testing strategy will consist of a mixture of unit tests, integration tests, and acceptance tests, which together cover all the functionality of your application, and verify its behavior under all conditions. Integration tests can be subdivided into **functionality**, **performance and scalability**, and **security** tests.

Type	Description	Example
Unit tests	Verify that one object does one thing.	Does the `create_user` stored procedure add a new row to the `users` table?
Integration tests	Verify that a set of objects work together.	When a new user is created, does a user object get saved to the database along with the users default settings and an email is sent to the user welcoming them to the project?
Acceptance tests	Verify the behavior of the application, usually in a format that non-technical users can understand.	Is discount rate given to orders over 30 GBP on the customer's first order of the week?

Unit tests and test-driven development

When people refer to using **test-driven development**, or TDD, what they often mean in practice is that they write unit tests. In other words, they write unit tests first, then write or modify the code, and then use the tests to verify that the code works. When writing unit tests, this "test-first" approach is useful, but you do not have to be practicing TDD to write unit tests and, conversely, if you are writing unit tests you are not necessarily practicing TDD.

Unit tests are pieces of code that verify that another piece of code functions correctly, independently of any other code. Unit tests should just test one property or behavior per test, and should be fast enough that they can be run by any developer on their own machines, every time they make a change. For unit tests that require access to data, developers should use a local copy of the database, loaded with minimal data.

Let's return to our earlier example, where we need to calculate a customer's average daily spend, and write an audit trail entry. Let's say we break this task down into a combination of stored procedures and inline table functions, as shown in Listing 8-1.

```
CREATE PROCEDURE get_customer_summary (@customer_name VARCHAR(25))
AS
    DECLARE @result MONEY = (
                                SELECT ds.average_daily_amount
                                        AS average_daily_spend
                                FROM Customers c
                                    CROSS APPLY
                                     dbo.daily_spend(c.customer_id) ds
                                WHERE c.customer_name = @customer_name
                            );
    EXEC add_audit_entry 'get_customer_summary', @@username, @result;
    SELECT @result;
```

Listing 8-1

We have a series of unit tests that each do one thing only:

- Does **`get_customer_summary`** return the correct result?
- Does **`get_customer_summary`** call **`add_audit_entry`**?
- Does **`dbo.daily_spend`** return the correct result?
- Does the audit log get written to by **`add_audit_entry`**?

What we are trying to do is create enough tests so that we cover all the likely outcomes, but at the same time quickly identify the precise cause of an issue. If we mock out the call to **`add_audit_entry`** in **`get_customer_summary`**, so that if a bug in **`add_audit_entry`** causes only the tests for **`add_audit_entry`** to fail, we know where to start looking. If the tests for **`add_audit_entry`** and **`get_customer_summary`** all fail, then it points to a broader issue beyond the scope of the tests, such as a database running out of disk space.

In this manner, what we would like to end up with is a set of unit tests that, firstly, prove that every part of our code is correct and returns the right result, but also stop other changes breaking our code, and offer examples of how the code is used and so provide a level of documentation for the code.

Proving code correctness

This is often the first thing people think of, and it is true that this is very important. When you write some code, you need to demonstrate that it does what you want it to do. A typical unit test will prove that, given some known inputs, which could either be parameters or data you have set up in a table, you always get the correct outputs.

Stop other changes breaking our code

Our unit tests on a given piece of code will help us detect breaking changes to the objects this code calls, to the objects that use this code as a dependency, or indeed to this piece of code.

Let's say someone was to change the name of a column in a table, or even drop a table on which our code, such as a stored procedure, relies. SQL Server has a feature where you can deploy code that references objects which are yet to be created, or lets you drop or rename objects, without anything more than a warning. If such changes get deployed without running any tests, then you're only going to find out about the problem at the point someone tries to execute the stored procedure on the database, and gets an error message. Instead, if you ran the code via tests, you would see immediately that the tests failed and that there was a problem that needed to be addressed.

Of course, unit tests help detect problems with accidental changes, as well as deliberate ones. For example, sometimes a developer will unwittingly check in the wrong code files, or delete the wrong line of code. A good set of unit tests means that such mistakes are spotted immediately, and we can reduce the chance of causing problems later.

Unit tests as documentation

The third main benefit of unit tests is a controversial one and there aren't many people who advocate relying only on tests as documentation. Nevertheless, having a good set of tests, with actual examples of how code is used, as well as tests which describe how and when they are used, is excellent living documentation for code.

Unit testing frameworks

Unit testing frameworks fall broadly into two categories:

- **Test written in application code** – for example, in Microsoft Visual Studio we can use MSTest to write tests in C# or VB.NET, and DBUnit allows developers to write tests in Java.
- **Tests written in the language of the database** – for example, for Microsoft SQL Server, we can write tests in T-SQL using the tSQLt framework.

Most developers prefer to write unit tests in their native application language, but it is worth bearing in mind that tSQLt has some additional features that make database testing simpler.

Using tSQLt as a test runner

With tSQLt, you define a list of tests in your code and tSQLt will run all your tests and report the status of each one. It also provides various helper procedures such as **`tSQLt.AssertEquals`**, which takes two variables and compares them to make sure that they are the same. You can optionally pass in a parameter to display a specific message, for example about which specific assertion test failed. This is useful, especially when tests are included in automated builds as you get details.

tSQLt will return the result of any tests that either pass or fail. If a test fails, then an exception is raised, so that you can easily run the tests as part of a manual or automated process. If you can run a command against SQL Server, then you can run tSQLt tests.

Mocking tables in tSQLt

tSQLt makes it very easy to mock a table, which makes setting up test data significantly easier for a normalized database. Let's say that we have a database schema that looks like Figure 8-2.

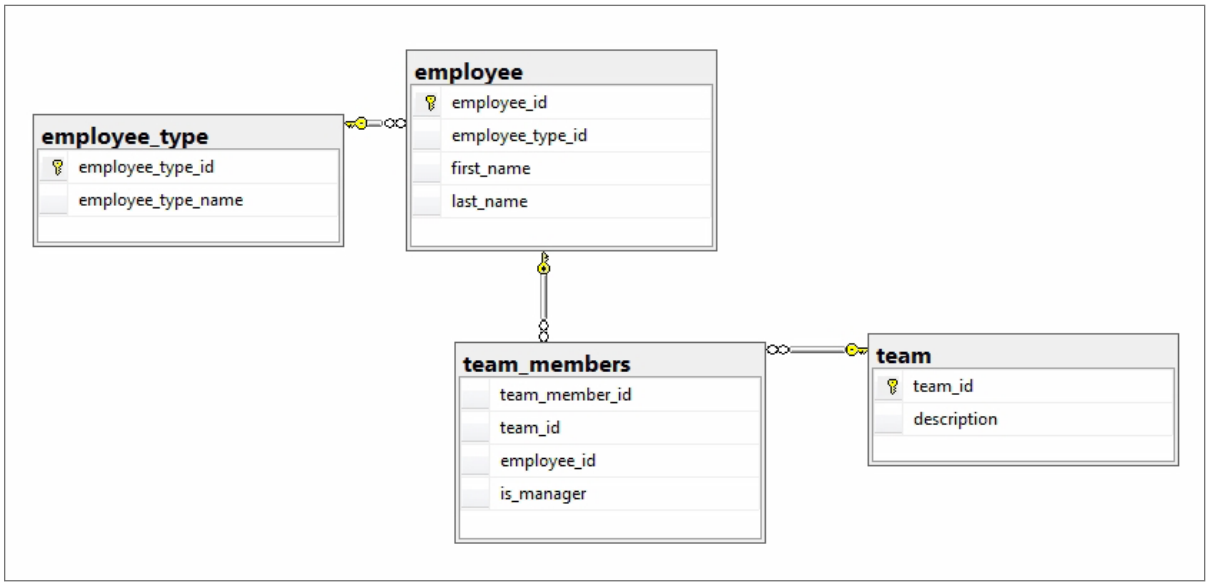

Figure 8-2

We write a **`get_employee`** stored procedure that looks as shown in Listing 8-2.

```
CREATE PROCEDURE get_employee (@employee_id INT)
AS
    SELECT first_name,
        last_name
    FROM employee
    WHERE employee_id = @employee_id;
```

Listing 8-2

We want to write a unit test to check that the **`get_employee`** stored procedure always returns the correct employee. For this test, we are interested only in the **`employee`** table and only in three columns in that table, **`employee_id`**, **`first_name`** and **`last_name`**.

Since we're using tSQLt, we write the test itself as a stored procedure (called **get_employee_tests**). The first thing it needs to do is pre-load some test data so that we have a set of rows that we know exist in the table. The problem with this is that, depending on the relationships and the constraints that exist on the tables, setting up test data can become quite a burden. Our test stored procedure would need to insert, into the three other tables, row values that complied with the existing keys and constraints. If another developer subsequently adds a new, non-**NULL** column to any of these tables, but doesn't define a **DEFAULT** constraint, then our tests will start failing at the point it tries to insert data, even though the change is potentially unrelated to the object we're testing.

To help avoid these sorts of problems, tSQLt can specify that a table is going to be used as part of a test and, for the lifetime of that test, the constraints, including any **FOREIGN KEY**s are dropped. When the tests finish, the constraints are re-enabled. The process looks like this:

1. In the test code call: **exec tSQLt.FakeTable 'TableName', 'SchemaName'**
2. Inside a transaction, tSQLt renames the existing table and creates a new table with the same definition but without triggers or defaults.
3. If you want to re-enable a specific constraint or trigger, then you can call **tSQLt.ApplyTrigger** or **tSQLt.ApplyConstraint**. This gives you the ability to test only specific parts of the application.
4. In your test code, insert the test data you need. It is advisable to add a row that you do want returned, and one that you do not want returned, so you can be sure that, not only are you getting the values you do want, but you are not getting the values you do not want.
5. In your test code, call the stored procedure you want to test and validate the results.

Listing 8-3 shows an example of a tSQLt test for the **get_employee** stored procedure.

```
CREATE PROCEDURE [get_employee_tests].[test correct employee is
returned]
AS
    EXEC tSQLt.FakeTable 'employee', 'dbo';

    INSERT INTO employee
    (
        employee_id,
        first_name,
```

```
        last_name
    )
    VALUES
    (1, 'ignore', 'me'),
    (2, 'find', 'them'),
    (3, 'no one', 'here');

    CREATE TABLE #results
    (
        first_name VARCHAR(MAX),
        last_name VARCHAR(MAX)
    );

    INSERT INTO #results
    EXEC get_employee 2;

    DECLARE @record_count INT = (
                                    SELECT COUNT(*) FROM #results
                                );
    EXEC tSQLt.AssertEquals 1, @record_count;
    SELECT @record_count =
    (
        SELECT COUNT(*)
        FROM #results
        WHERE first_name = 'find'
              AND last_name = 'them'
    );
    EXEC tSQLt.AssertEquals 1, @record_count;
```

Listing 8-3

In Listing 8-3, we can see that all we need to do is to call **`FakeTable`** for any table that we are going to use as part of the test, and ensure it only has the data that it requires for the test.

This approach makes it simple to set up test data, which makes it easier to write and maintain tests. It also makes the developer think about which tables and columns and rows are required for the test.

Mocking stored procedures with tSQLt

When unit testing, we want to be able to test an object without testing its dependencies. If one object uses another, then we want to show that each piece works individually using unit tests, and then use integration tests to show that they work together.

As well as mocking, or faking, tables, tSQLt allows us to mock procedures. When testing code that calls a stored procedure, we can instead call a "spy" procedure that simply returns some known values. Interestingly, this means that we can be sure that the code we are testing gets a valid value, even if the stored procedure it is calling is broken. This means that the unit tests on the broken stored procedure should fail, but all other unit tests will continue to work.

Listing 8-4 shows a new version of our **`get_employee`** stored procedure.

```
CREATE PROCEDURE get_employee (@employee_id INT)
AS
    DECLARE @access_allowed INT;
    EXEC @access_allowed = security.verify_access @employee_id;
    IF @access_allowed = 1
    BEGIN
        SELECT first_name,
            last_name
        FROM employee
        WHERE employee_id = @employee_id;
        EXEC auditing.add_audit_log 'get_employee', @employee_id;
    END;
GO
```

Listing 8-4

We can set up spy procedures for **`verify_access`** and **`auditing.add_audit_log`**. For example, we can replace the **`verify_access`** procedure with a spy procedure that simply returns a value appropriate for that test. This allows us to perform "happy" and "not-happy" path testing without having to set up a lot of test data. The idea is that we should verify, both that the procedure works as expected (returns the employee when access is granted), and that it does not work, or do anything unexpected, when we expect it to not work.

Listing 8-5 shows a sample test for the **`get_employee`** stored procedure.

```
CREATE PROCEDURE [get_employee_tests].[test results returned when
access granted]
AS
BEGIN

    EXEC tSQLt.FakeTable 'employee', 'dbo';
    EXEC tSQLt.SpyProcedure N'security.verify_access', N'return 1';

    INSERT INTO employee
    (
        employee_id,
        first_name,
        last_name
    )
    VALUES
    (1, 'ignore', 'me'),
    (2, 'find', 'them'),
    (3, 'no one', 'here');

    CREATE TABLE #results
    (
        first_name VARCHAR(MAX),
        last_name VARCHAR(MAX)
    );

    INSERT INTO #results
    EXEC get_employee 2;

    DECLARE @record_count INT = (
                                    SELECT COUNT(*) FROM #results
                                );
    EXEC tSQLt.AssertEquals 1, @record_count;
    SELECT @record_count =
    (
        SELECT COUNT(*)
        FROM #results
        WHERE first_name = 'find'
              AND last_name = 'them'
    );
    EXEC tSQLt.AssertEquals 1, @record_count;

    SELECT @record_count =
    (
```

```
        SELECT COUNT(*)
        FROM security.verify_access_SpyProcedureLog
        WHERE employee_id = 2
    );

    EXEC tSQlt.AssertEquals 1, @record_count;

END;
```

Listing 8-5

When a procedure is mocked using spy procedure, a table is created that is available inside the test that is the name of the procedure, with **_SpyProcedureLog** appended to the end. This table contains the calls to the procedure and the parameters passed in, so we can query it, as demonstrated toward the end of Listing 8-5, to verify that the **verify_access** procedure was called exactly once and the **employee_id** passed in was "2".

Using MSTest

MSTest is the testing utility that comes with Visual Studio. The unit tests are implemented using C# code that uses the Visual Studio testing framework, so running tests and including tests in a build process becomes simple. It also means that the tests can run alongside any application unit tests that you might have in the same Visual Studio solution.

If you also have SQL Server Data Tools installed then you get a set of templates to **add SQL Server Unit Tests**, which allow you to set up test data, run some tests and then verify the results. The tests themselves have a basic UI to help develop code and they integrate with the Visual Studio test runner so you can run your tests from within Visual Studio. You do not have to have your database in SSDT to use MSTest to create your tests, and there is no dependency on SSDT database projects.

When you create the SSDT unit tests, Visual Studio will generate C# code as well as T-SQL scripts; the C# code calls the T-SQL scripts to run:

- before your test
- during your test
- after your test.

The first step is to ensure you have the correct data set up. So, for example, before running our previous test for the **get_employee** stored procedure, we'd run a pre-test script that looks like Listing 8-6.

```
TRUNCATE TABLE employee;

INSERT INTO employee
(
    employee_id,
    first_name,
    last_name
)
VALUES
(1, 'ignore', 'me'),
(2, 'find', 'them'),
(3, 'no one', 'here');
```

Listing 8-6

Then the test code is simply exec **get_employee 2;**. To validate the results, there are many test conditions that you can configure, such as checking that there is a non-empty result set, or that the test returns a specific scalar value. You can use a wizard to generate a checksum for a result set (see Figure 8-3), and include this in your test, so you can be certain that the result is exactly as you expect.

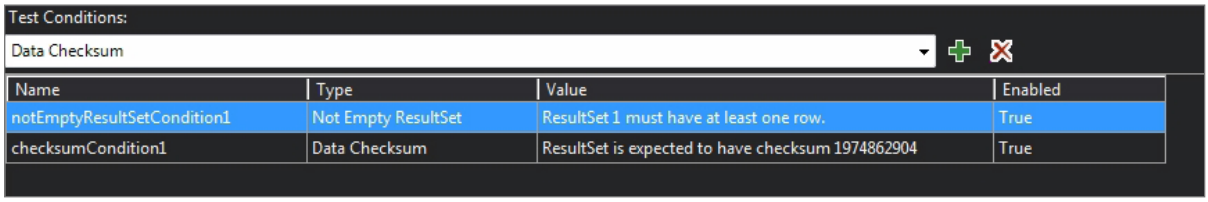

Figure 8-3

Other unit testing frameworks

A full description of every unit testing framework on the market is outside the scope of this chapter but below is an overview of some of the ones available.

Name	URL	Overview
SQLUnit	http://sqlunit.sourceforge.net/	Tests are defined using XML and run from Java. These work with any RDBMS that has a JDBC client.
DBUnit	http://dbunit.sourceforge.net/	Cross-platform framework for deploying test data and running tests.
tSQLt	http://tsqlt.org/	Microsoft SQL Server specific framework. Includes mocking support.
Visual Studio SQL Server Unit Test	https://msdn.microsoft.com/en-us/library/jj851212.aspx	Microsoft Visual Studio templates for creating unit tests that are a mixture of C# and T-SQL.

Integration testing

Where unit tests aim to verify that each distinct unit of code works in isolation, integration tests validate that all components work together, and the interfaces between them are correctly configured and deployed. Integration tests validate that entire processes or process-flows work as expected. For example, you may have an integration test that validates that when a user attempts to log on, if the correct username and password are supplied then access is granted, and an audit log is updated.

When writing integration tests, it is important to take a view of the whole application, and decide what is important and what should be tested. Each application will have its own set of unique tests but there are some general categories of tests that we can start to identify:

- **Functionality test** – Does the process work as expected?
- **Performance and scalability tests** – If performance is important, what are the desired metrics? Does the application process meet those?
- **Security tests** – Can we prove that the application process is secure?

Having identified the key areas of the application to test, often the hardest part of writing integration tests is creating the infrastructure and framework to run the actual tests.

For example, we need a quick way to generate enough test data in the correct state. What I typically do is have a generic C# class, shown in Listing 8-7, to represent a table. It takes the table name as a parameter and exposes a "clear" method that deletes all the data in the table, and in any tables that reference the table, by way of a **FOREIGN KEY**. Each table that inherits from this class implements its own columns as properties that can be used by an ORM such as dapper.net to quickly and easily translate the table object into SQL.

```
using System;
using System.Collections.Generic;
using System.Data.SqlClient;
using System.Linq;
using Dapper;

namespace STIntTest
{
    [TestFixture]   //nunit tests
    public class Orders_Table_Tests
    {
        [Test]
        public void Order_Is_Added_For_Customer()
        {
            var ordersTable = new OrdersTable("server=.;integrated
security=sspi;initial catalog=VSTS");
            ordersTable.DeleteAllData();
            ordersTable.AddRows(1000);
            ordersTable.Write();

            //execute code....


            var rows = ordersTable.GetRows();
            Assert.True(rows.Any(p => p.CustomerID == "a_unique_
id"));
        }
    }

    public class Table
    {
        private readonly string _connectionString;
        private readonly string _tableName;
        protected readonly List<object> Rows = new List<object>();
        protected Table(string connectionString, string tableName)
```

```
        {
            _connectionString = connectionString;
            _tableName = tableName;
        }

        public virtual void AddRows(int count)
        {

        }

        public virtual void Write()
        {

        }

        protected void Write(string columns, string values)
        {
            foreach (var row in Rows)
            {
                using (var connection = new SqlConnection(_
connectionString))
                {
                    connection.Execute($"insert into {_tableName}
({columns}) select {values}", row);
                }
            }
        }

        public void DeleteAllData()
        {
            var tables = GetReferencingTables();
            foreach (var table in tables)
            {
                using (var connection = new SqlConnection(_
connectionString))
                {
                    connection.Open();
                    connection.Execute($"DELETE [{table}]");
                }
            }

            using (var connection = new SqlConnection(_
connectionString))
            {
```

```
                connection.Open();
                connection.Execute($"DELETE [{_tableName}]");
            }
        }

        private IEnumerable<string> GetReferencingTables()
        {   //you will likely need to make this a recursive call...
            using (var connection = new SqlConnection(_
connectionString))
            {
                connection.Open();
                return connection.Query<string>("SELECT OBJECT_
NAME(object_id) ReferencingTable FROM sys.foreign_keys WHERE
OBJECT_NAME(referenced_object_id) = @TableName",new {TableName =
_tableName});
            }
        }

        public IEnumerable<dynamic> GetRows()
        {
            using (var connection = new SqlConnection(_
connectionString))
            {
                connection.Open();
                return connection.Query($"SELECT * FROM [{_
tableName}]");
            }
        }
    }

    public class OrdersTable : Table
    {
        public OrdersTable(string connectionString) :
base(connectionString, "Orders")
        {
        }

        public override void AddRows(int count)
        {
            for (var i = 0; i < count; i++)
            {
                Rows.Add(new { CustomerID = "0000ABC" + i,
OrderTotal = 102.11 + i });
            }
```

```
        }

        public void AddRow(string customerID, double orderTotal)
        {
            Rows.Add(new {CustomerID = customerID, OrderTotal =
orderTotal});
        }

        public override void Write()
        {
            this.Write("CustomerID, OrderTotal", "@CustomerID, @
OrderTotal");
        }
    }

    public static class Assert
    {
        public static bool True(bool result)
        {
            return result;
        }
    }

}
```

Listing 8-7

This seems like a lot of work but, once you have a table class for your main tables, you can reuse it, and spend more time writing the actual test code.

Typically, our testing framework, such as NUnit, MSTest, JUnit, or similar, will call the application code and perform the following steps:

- Start/Refresh any required infrastructure.
- Deploy the code to be tested.
- Run the code.
- Run the test.
- Gather the results.

As an example, let's consider an ETL process that downloads a **.csv** file from an FTP site, then runs an SSIS package to load the data.

What we might end up with is an integration test that:

- creates a **data.csv** file, ideally with a unique value that the test knows about
- starts a local FTP server
- copies the csv to the local FTP server
- clears out the target database
- starts the SSIS package with a custom configuration (possible using **dtexec.exe**)
- reads the database and checks that the unique known test values are present.

The code needed to implement this test is likely more than the code required to complete the ETL process but, if each step is made available as a separate component that can be shared among other tests and projects, then having these tests can provide massive cost-savings in terms of time between writing code and detecting failures and diagnosing bugs.

This method of having an integration test that deploys and runs the code with test data is a great way to diagnose and fix bugs. The interesting part of having integration tests like this is that, as well as testing the code itself, you also start testing your deployment systems. If you have a PowerShell script that deploys your database and reference data, why not use it as the step that sets up your test database? In that way, some of the best-tested components of your application are the deployment components!

Performance tests

Many teams don't find out about performance problems until they get so bad that users complain, because they are unable to do their work. The goal of performance tests is to prove that any variation in performance, because of changes to the code or to the infrastructure, can be measured. If we build automated performance testing into the DLM process, then not only can we be sure that the code performs to expectations, and that subsequent changes do not have a negative impact, but we can also performance-test potential fixes to production issues before the code is deployed to production.

As for all tests, we need to know when it makes sense to add more performance tests, or more complexity to existing tests. If the performance of a business process is not critical then, typically, we will not need explicit performance tests. If performance is critical but volumes of data and number of concurrent users is low, then performance testing is relatively lightweight and easy to set up.

Performance testing becomes very important when the performance of a business process is critical, and when the production database has a large data volume and/or a large number of concurrent users. In such cases, many organizations attempt to guarantee performance by "throwing money at the problem" and over-spec'ing the hardware, but the most reliable way to ensure acceptable performance in the long term is to be able to mimic those conditions in the integration test environment, and perform performance assessments that prove that the process performs and scales to requirements.

Performance tests are often just an extension of an integration test. For example, we might have an integration test that works like the previous ETL example, but with additional requirements:

- we need to understand the difference between the test and production hardware
- we need reliable performance baselines for comparison
- we need to be able to load data with a volume and profile that matches the production data
- we need a way to simulate production-like concurrent loads
- we need a mechanism to record execution time and system metrics, such as CPU usage.

Hardware and physical specs

When using test data, it is important to understand the difference between the test and production hardware. Ideally you should have a test rig that has the same hardware specs as production and the same configuration. If you have a three-tiered application in production, mimic that in your performance tests. If your database is in a different data center to your application tier with a large latency, then you must mimic that latency in test.

If, in test, you have a set of hardware that differs massively from production, then what typically happens is that performance issues in the test environments are ignored and expected to be fixed when moving to production, but this often masks issues that could be fixed early on. If you test with a similar set of hardware, then any performance issues must be fixed in a test environment and not left until production.

Baselines for comparison

Let's say that, as part of your integration and performance tests, you deploy the latest database build, fill it with data, and run the tests. You compare the results to the results from the last set of performance tests, but how can you be sure that nothing else changed in the test environment that would skew the results? Ideally, you should be able to deploy the current version and the previous version, and run the tests side by side, so you can rule out environmental changes.

The best example of this I saw was a team who nightly ran the previous ten releases and charted the performance of the latest set of changes. The main benefit was that, if anything was changed on the test server, such as adding additional RAM, the performance across the ten releases was still comparable. If you only test the latest release, you won't know if any performance increase is attributable to code improvements, or just more of the test database being cached.

Generating or loading production-like data

We need both a realistic volume and a realistic profile of data to ensure that we get as close to production results as possible. As discussed earlier, when our application attempts to query a database during the performance test, the RDBMS will examine the text of the query and generate a query execution plan based on its statistical knowledge of the data, or it will reuse an existing plan, if it has seen the exact same text before and the plan for that text is still in the plan cache. The plan it chooses will depend on any parameter values submitted with the text, the distribution and volume of data in the tables, available indexes, and so on.

If we're running tests against small sample data sets, for example, then the optimizer is likely to choose **Nested Loop** join operators for joining data. If the same query, on production, needs to join two large data sets, the optimizer might instead choose to use a **Hash Match** join which uses temporary storage, which could spill to disk. In other words, the execution profile of your test will not match what's seen in production. Likewise, if you run a performance test with a table with mostly **`NULL`**s, you will likely get different behavior to production, where the data is mostly non-**`NULL`**.

Ideally, test data should be generated to the same volume, distribution, characteristics, and datatype as the production data, and each type of test may require different test data sets.

There are several ways to get production-like data in your test systems such as:

- restore the production database
- restore the production database, and overwrite any personally identifiable information (PII) data
- build the latest database version, then generate test data using a tool, such as the Redgate SQL Data Generator
- build the latest database version, then manually load realistic test data, e.g. from "native" BCP format using bulk load.

Choosing a way to get realistic test data depends on the environment; there are many database applications where the security requirements mean that copying the production data is unacceptable.

Stats-only database

Microsoft SQL Server allows you to copy a database schema plus a copy the statistics using DBCC CLONEDATABASE (http://preview.tinyurl.com/n4rbbr8) so the optimizer generates the same plan as it would when the query was run against the production data. For pre-SQL Server 2014 databases, you can create a statistics-only database by scripting out the statistics (see Matteo Lorini's article at http://preview.tinyurl.com/jvhstro). This is great for looking at and checking execution plans, but it should not be used for tests where you time the results as, although the plan has the same cost, there is a big difference between processing a few pages and processing a few million pages.

Simulating production-like loads

The performance of an RDBMS is also affected by other requests currently executing, as those other requests may hold locks on data your query needs, or may monopolize CPU resources, cause disk bottlenecks, hog memory resources, and so on. Depending on the transaction isolation levels in use, it's possible for a query to start returning unexpected results when it's running concurrently with other queries that are modifying that data.

Ideally, you'll be able to simulate a background workload reminiscent of what you expect to see in production to catch such "race condition" bugs. There are various tools available to simulate load from multiple concurrent users such as SQLQueryStress or, from Microsoft, the RML utilities and OStress (http://preview.tinyurl.com/multbvv). SQL Server can also record a profiler trace and use the RMS utilities distributed replay client to replay requests.

Record execution time

Our performance tests need to demonstrate clearly how a change has affected performance, either positively or negatively. If we measure, store, and chart how long it takes to perform a test, then we can easily visualize the performance impact of a change, and so predict how that change will behave in production.

One obvious metric to track is simply how long a test takes to run. In a C# test, we can use the **`System.Diagnostics.Stopwatch`** class. This is useful to give us an overall "the test took x long" which is a quick metric that shows overall performance. For T-SQL tests, we can capture I/O and timing metrics using either the **`SET STATISTICS IO`**/**`TIME`** T-SQL commands, or using Extended Events. Phil Factor's article on Simple Talk (http://preview.tinyurl.com/lcsnfeu) shows one example of setting up a test harness that captures these metrics plus the query execution plan.

Record server metrics such as CPU usage

Alongside raw timings, we also need to collect SQL Server performance metrics, so that we can start to understand, not only what sorts of changes affect performance, and by how much, but also *why*. By charting these performance metrics, it will, over time, make troubleshooting future performance issues much easier.

Another one we can use is Performance Monitor, or **Perfmon**, which measures various performance statistics on a regular interval, and allows us to save those metrics to a file or to a SQL Server table. We can collect broad server-level metrics relating to disk, processor, and memory usage, as well as specific metrics relating to SQL Server's consumption of I/O, CPU and memory (disk transfer rates, amount of CPU time consumed, and so on).

When a code change causes a significant change in performance, these metrics will highlight the "queues," i.e. possible areas of resource contention, for SQL Server. Typical PerfMon counters I collect include the following:

- **`PhysicalDisk – Avg. Disk sec/Read`**
- **`PhysicalDisk – Avg. Disk sec/Write`**
- **`Processor - % Processor Usage`**
- **`Process – sqlservr.exe - % Processor Usage`**
- **`SQLServer:Buffer Manager – Buffer cache hit ratio`**

- `SQLServer:Buffer Manager – Page Life Expectancy`
- `SQLServer:Buffer Manager – Target Pages`
- `SQLServer:Buffer Manager – Total Pages`
- `SQLServer:Locks – Average Wait Time (ms)`
- `SQLServer:Locks – Lock Requests/sec`
- `SQLServer:Locks – Number of Deadlocks/sec`

Sometimes, we'll use these metrics directly within a test. For example, if the **`Number of Deadlocks/sec`** metric is non-zero, the test should fail. Others, as discussed, are useful for understanding *why* a test took too long to run.

We can automate collection of these metrics, using the Performance Analysis of Logs tool, or by running a script that collects them from the Dynamic Management Views, as demonstrated by Jonathan Kehayias in his article, *A Performance Troubleshooting Methodology for SQL Server* on Simple Talk at http://preview.tinyurl.com/lny7tde.

As Jonathan discusses in his article, having located areas of recourse contention, you may need to collect further data to pinpoint the cause of the performance problem. A classic analysis is to correlate the resource metrics (the queues) with wait statistics (see SQLSkills.com, http://preview.tinyurl.com/mrsd7c2) describing which SQL Server processes are being forced to wait a significant amount of time before proceeding with their work, and what is causing them to wait.

As you discover common causes of performance problems with your application, you can add new tests that collect data from the system catalogs, DMVs, and elsewhere, and then your tests can fail. In this way, they can provide information that will help you identify quickly what caused the issue. In my experience, this approach can drastically reduce troubleshooting time.

Security tests

Security tests are like performance tests in that each application tends to have different security requirements, and you need to understand your requirements before you start. The tests required for a database with no external access points and containing no sensitive data are very different from those required for a database that contains users' personal details and credit card information. In the latter case, you have a legal responsibility to ensure the data is protected correctly, in all environments and during all activities, including testing.

Typically, however, you will need some basic security tests to ensure that:

- Users do not have extra privileges to data they ought not to see.
- Inputs are verified and not executed if they are not exactly as expected, so someone called "`jim o'--\r\n drop database db_name()`" does not actually cause any harm.
- Users are only able to view or modify data to which they have been granted access or modification rights. You need to be sure that a user has the minimum necessary permissions to the data they need, and is not a member of, for example, powerful server roles, and can't gain access to other data by simply changing a query string.

Most of the work involved in writing security tests is defining what the requirements are and classifying the data. For example, if you have credit card information or personally identifiable data then you need to make sure that the data is encrypted and secured appropriately. There are various tools to help you keep your data secure and the technology and approach you use will guide what tests you need to write.

Once you have classified your data, you can ensure that it is secure and that unauthorized users are unable to get to it. This approach typically requires two tests, one that checks that a user with the appropriate permissions can access the data, and one that verifies that a user without the required permissions is unable to access the data.

We also need tests to check that users are unable to send specially-crafted inputs that can access, corrupt, or steal data using SQL injection attacks. It is often wise to protect against this at different layers in your application, and write tests to prove that SQL injection attacks are not possible but, again, it depends on the type of application.

It should be obvious, but using production data with sensitive information for your test environments is a bad idea. Instead, you should either generate test data or sanitize the data before deploying to your test environment. Sanitizing the data needs to be done with care and in a secure environment.

Test coverage

When writing tests, it is not always obvious how many tests, and of what type, are required. If we must implement a login process for a web application, then we should aim to make sure that we have enough tests to show that the individual parts all work in isolation and that the end-to-end process works, so we might end up with:

- Acceptance / UI tests that show that the web page accepts the correct input and logs the user in.
- Integration tests that check that the `LogonService` call works with the right username and password.
- Unit tests for each of the components, such as the stored procedure that validates the username and password.

In other words, we'll usually have one or two acceptance tests, a few more integration tests, and a lot more unit tests. Overall, we should have "covered" with tests pretty much every line of code and every statement in the application.

When starting to write tests for an existing application, it is often easier to start with a few integration tests that validate that critical processes work, and later add unit tests as you modify code. If you do this, you will gain the benefit of having as much of the application covered as quickly as possible, with the benefit of being able to add unit and integration tests as you go along.

Summary

A DLM approach to database testing will not only prove that an application works as expected in the development environment, but will also give the team confidence that all requirements will continue to be met, and that all processes will continue to behave reliably and correctly under production conditions.

This requires unit tests to verify that each component works correctly in isolation, integration tests to prove that these components work together correctly in broader business processes, and that they perform to requirements under production conditions, and that the data is only accessible by those who have permission to see it.

A good database test suite will help the team uncover and fix problems as early as possible in the development cycle, and therefore deploy changes faster, in the knowledge that the changes will work as expected, and will not have any negative side-effects on other application components or system resources.

Ultimately, I've found that good database testing helps a team enjoy developing more, and worry less. When issues do occur, which are missed by the test suite, then you should see it as an opportunity to test your disaster recovery plans and to improve your test suite still further.

9 – Database Continuous Integration

This chapter explains some of the secrets of using CI for databases to relieve some of the pain points of the database delivery process, making it more visible, predictable, and measurable. In addition, quality can be improved, change speeded up, and cost reduced.

DLM recommends that you integrate database changes as frequently as possible because it makes change easier and identifies mistakes faster. By integrating changes to database code and components as often as possible, and testing the results, problems become visible while they are still easily managed. CI remains the ideal, though it is not always achievable.

CI doesn't get rid of bugs, but it does make them dramatically easier to find and remove provided the team can be sure that the cause is one of the changes that were made since the last successful integration. If a developer's code has introduced an error, or has caused another developer's previously-working piece of code to fail, then the team knows immediately. No further integrations occur until the issue is fixed.

Many of the benefits of database CI stem from the automated tests that prove that the database meets the requirements defined by the tests, at all times. The unit tests are important: they ensure that the individual units of code in isolation always function, and the integration tests make sure that they work together to implement processes. We'll cover the specifics of what database tests we need to run in the next chapter. Here, we will discuss how to adopt and implement database CI as a practice, its importance in DLM, and how to overcome common challenges when making the database a full partner in your existing application CI processes.

Why is integration so important?

All applications are made up of components. Even an application hand-crafted from procedural code will depend on third-party components, frameworks, libraries, operating-system interfaces, data sources, and databases. At the other extreme, the custom application could be a mash-up of standard off-the-shelf applications.

All components are liable to upgrades and changes, especially during active development. Each time any component part of an application changes to a new version, we must perform a complete build of all parts of the application that rely on that component, followed by

integration, with its integration tests. With a rapidly changing application, integration must be continuous so that we can prove, continuously, that nothing has been broken as a result of any change.

Any change to the application code requires integration, as does any change to a third-party component. Where does the database fit in? From the application-perspective, the database is just another component, and a typical corporate application can be dependent on a handful of databases. A database change merely triggers the same work as any other component. The application integration tests must prove that all database access methods continue to perform correctly, according to the database interface specification.

From the database perspective, it is, in effect, an application, with its own hierarchy of components on which it depends. Just as for an application, any change to the database code or components should prompt a fresh build, followed by integration and its tests.

The purpose of database CI, then, is exactly the same as for application CI. The development team establish a working version of the database very early in the development cycle, and then continue to verify regularly that it remains in a working state as they expand and refactor the schema and database code objects. Developers integrate new and changed code into a shared version-control repository several times a day. Development proceeds in small steps. Developers first write the tests that, if passed, will prove that a small new piece of functionality works. They then implement the code to make the tests pass. When the tests pass, they commit the code to "trunk" in the shared VCS, and their "commit tests" are added to the broader suite of tests for the application. Each commit, or check-in, is then verified by an automated database build or migration, and subsequent testing, allowing teams to detect problems early.

Why database CI?

CI within database development remains relatively rare compared to application development. Inevitably, "late integration" of database changes causes headaches and delay toward the end of a project. There are often unpleasant surprises that lead to manual scripting, and a lot of tweaking of those scripts, to get the database working as expected. Database CI minimizes the element of surprise.

Database testing is different in nature, and often far more complex, than application testing. The integration testing of a database has to go well beyond answering the question of

whether code passes or fails. It must go much more deeply into whether the system performs well under load, and if it scales properly. It should check that access rights are done correctly. Integration test should, in short, ensure that it is providing a service as expected.

However, ad hoc integration of database changes doesn't help. In consequence, a database release can cause a lot of subsequent problems in the QA and production environments. The problems can stem from unreliable stored procedures, functions, and triggers, to schema changes. Even an apparently simple database change that requires no complicated data migrations can cause problems if not integrated early and tested thoroughly. For example, a change such as adding a new column can cause unexpected failures if certain applications don't qualify column names, or expect only a certain number of columns, or columns in a certain order. Even more potentially damaging to the business are problems affecting data integrity, which stem largely from untested or missing data quality and referential integrity constraints.

Instead, database changes must be integrated and tested regularly and, if possible, continuously. Although shared-development databases are, in a sense, self-integrating, you still ought to integrate them because we are still obliged to test the overall behavior of the database system, and prove that a working database can be built from the component list, which includes not only the database code and schema changes, but also SSIS jobs, CLR libraries, Agent tasks, alerts, messaging, and so on.

Database CI leads naturally to earlier and more frequent releases, which means that all the scripts and files required to migrate a database from one version to the next will be thoroughly tested in development, and then again when deploying the release to QA and staging. As a result, deployments to production will be rehearsed and far more reliable. Also, the more rapidly new functionality and improvements can be delivered, the easier it is to improve collaboration with the DBA and systems team. If we use the same tools, both to repeatedly build the latest database version in the CI environment, and to deploy the release to QA, staging and ultimately production, then we can have more confidence that they will work as intended.

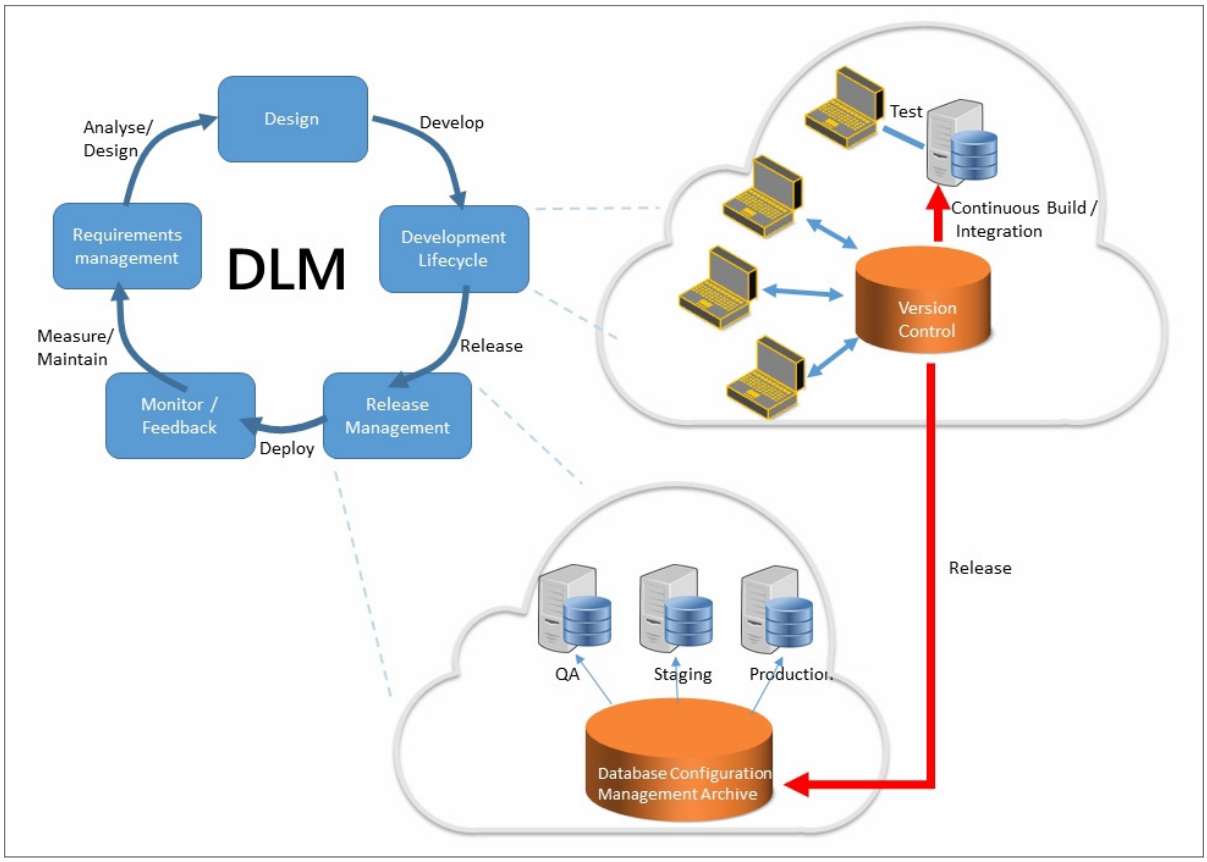

Figure 9-1

Of course, the subsequent deployment pipeline requires more than this to work. The testing in these more production-like environments is rather different in nature. It will encompass acceptance testing, and verify that security, capacity, availability, and performance characteristics of the production system are acceptable to the business or customer, and in line with the signed-off specification of the system. This, in turn, requires a reliable process for safe delivery of production data, with the necessary obfuscations, into those environments. We'll discuss these issues in Chapter 10, *Database Deployment and Release*.

Additional benefits of a well-drilled database CI process will emerge as the team gain experience with DLM practices, and introduces more automation and workflow techniques into their database delivery pipeline. For example, if the team have integrated the issue-tracking

system into the VCS, as described in Chapter 11, *Issue Tracking for Databases*, they can respond quickly to any critical issues. Through their CI processes, they will hopefully be able to quickly spin up an environment and establish the correct version of the database, as well as data in a state it existed soon before the error occurred. This often requires a "clean" version of production data (for example, minus any sensitive data elements). They can then reproduce the error, and write a failing test that will validate the fix and prevent regressions, meaning introducing a new problem while solving an existing one.

Application-specific versus enterprise databases

As a general principle, it is advantageous to develop applications and their databases together. It makes sense to integrate them as closely as possible to their point of creation, in order to help improve the quality of both code bases and the interactions between the two. It is much better to settle, as early as possible, issues about where any particular functionality is best placed. However, this integration between the application and the database CI processes should not interfere with the individual CI processes for either the application or the database. Instead, it must actively support both processes in order to enhance them. When the database serves only a specific application, and the two are being developed at the same time, then you may be able to do the application CI and the database CI together in one integration process, treating it as if the database code were part of the application code.

Some enterprise databases support numerous applications. If you're working with a database through which several applications coordinate, or an external database, then it is unlikely that you'll be able to develop your application and the database as a tight unit. Database CI will still bring benefits but, in this case, the database is treated by the application as a separate component. For integration testing of the application we may need to "mock" the database interface based on an "interface contract" that is negotiated with, and supplied by, the database owners. This interface will comprise a set of routines (views, functions, and procedures) you call from the application to access the required data. In such projects, there is inevitably special emphasis on the testing that takes place in the production-like environments, such as staging, which will host versions of all of the applications that use that database.

The decoupling of application and database changes will be enabled by sensible version-control mechanisms, as described in Chapter 5, *Database Version Control*. Essentially, scripts must exist in version control to roll forward, or back, between any two database versions. Assuming the live database stores somewhere its version number, and the application stores

the database version with which it is designed to work, then the CI or deployment tool will always be able to run the appropriate set of scripts to create the version of the database required by the application.

Prerequisites for database CI

Once the team have agreed to implement database CI, perhaps the two major prerequisites are database version control, and automated database builds. However, there are a couple of other practices and tools that will help ensure your CI processes proceed smoothly.

Maintain the database in version control

The first prerequisite for database CI is that the source of truth for the database CI process and all subsequent database deployments must be the build and migration scripts in a version-control system, such as Git or Subversion, as described in Chapter 5. In other words, the database CI process must always be triggered from the VCS. There must be no ad hoc builds or database modifications that bypass this formal process.

There are several CI management software services available (covered briefly a little later). Every one I've worked with has hooks into various version-control systems. With these hooks you can set up mechanisms for automating your database CI processes.

You must also incorporate into your version-control system a mechanism of labeling or versioning successful integrations. If the latest code failed testing, it's not ready for further integrations until fixes are implemented and it passes the CI process. This is especially true when running overnight database CI testing, with a realistic data load. Never schedule this process to run on a database version that has associated failing tests. Conversely, if the overnight integration tests pass, the database should be assigned a significant new label or version. One approach is to advance the first significant number after the major version number only if the overnight database CI tests, typically run nightly, pass (2.20, 2.30, and so on).

The next day, each developer can run the scripts to create this new version in their dedicated development database, or the team can upgrade the shared database, and proceed with development, with fast integration tests running on each commit, which will progress the least significant digit (2.2.1, 2.2.2, and so on).

Automated database builds

Before you attempt database CI, you should already to be able to automate a build of your database from the command line (i.e. separately from your CI system), using a set of DDL scripts stored in the VCS. As explained in Chapter 5, *Database Version Control,* a well-documented build script for every object makes it very easy for the team to see the exact state of an object at any given time, and to understand how changes to that object are likely to affect other objects. I recommend getting a complete build of an empty database set up as soon as you have your database moved into version control. There are a number of reasons for this:

- It provides you with information on the effectiveness of your database version-control implementation.
- You can immediately identify gaps, such as missing objects, in your version-control management.
- You get feedback on your ability to build your database out of source control.

Once you've established independently that you can successfully build a database from scratch, from its component scripts, then triggering an automated database build through your CI process, in response to database modifications, is a great first step on the road to database CI.

Early and frequent commits

The idea of database CI is to have a fast and continuous set of validations of your database changes, so my general advice is to integrate like you're voting in Chicago: early and often (an old joke; you're only supposed to vote once per election). In other words, rather than hold onto functionality for days or even weeks while you're working on it, segment the work into discrete sets of changes, such as just one stored procedure, or just the changes necessary to modify a single table, and commit those changes into version control as often as possible. Ideally, you'll have a "commit-based database CI" process set up (see later) that builds a new database, and runs basic tests, on each commit of a database change. Even if this isn't achievable, you'll still want to perform as many successful integrations of your database changes as possible. You'll then have the benefit of integrating each small change as you work on it, rather than having a single massive set of code to integrate, which is likely to result in multiple failures.

One proviso, of course, is that you should only ever commit a database change to version control after you've validated that it's working on your development machine, via unit testing. If you don't know if the code is working, don't commit it into version control. You're likely to cause the database CI process to fail, but even worse, you might be supplying broken code to other developers.

Isolate the CI Server environment

The CI Server should be running SQL Server of the same version as production, on a machine with no other service on it, and with restricted logins. Don't run your database CI processes on a server that also runs additional services, or where people are attempting to get other work done. Your CI processes will eat server resources, as they drop and re-create databases, load data, run tests, and so on. If you then add in the need to have this process for more than one development team, for more than one database or set of databases, all this points to the requirement for an isolated environment.

It will ensure that CI testing doesn't interfere with the work of others, and also that failures are related either to the CI process itself, or problems in the code you've modified within version control. Build or migration failures within the CI process should not be caused because someone was mistakenly reading data from the CI database or attempting to run code manually against the CI database. These types of false negatives will not help you improve your CI process or the quality of your code.

Database CI tools

Notionally, aside from a version-control system, the only other components required to begin database CI are a CI Server and a test framework. However, in order to fully support database CI, you'll also need a workflow system or some way to guarantee an appropriate response to a broken build. You will also need a reporting system.

CI server

While it is possible to build your own CI server process, it's a lot of work. Unless you have specialized requirements, it will likely be easier to take advantage of an established CI server such as TeamCity, Jenkins, CruiseControl, Hudson, or Visual Studio Team Services.

Each of these CI servers offers specialist functionality that may make one of them more attractive than another, within your particular environment. Plan for an evaluation period to test two or more of these servers in order to identify the one that works best for you.

At a minimum, your CI server process should:

- **Integrate with your version-control environment** – including the ability to deal with labels, branching, and versions.
- **Allow granular control over the workflow that determines how the database is built** – the order in which actions occur, and the ability to add additional functions such as calls to tests.
- **Maintain a record of builds** – in order to know the success or failure of any given run.
- **Offer connectivity to email and other alerting mechanisms** – so that when the process is fully automated, you can get reports on the successful or unsuccessful builds.

Test framework

Every time a developer commits a database change to the VCS, we will want to trigger a database CI process that performs a build and runs a small suite of tests to ensure the basic behavior of the database structure and code. For our nightly integration builds, we'll run more extensive integration and acceptance tests with a realistic data load. These tests need to relay information back to the build server; probably via NUnit-style XML output, which any modern CI solution can understand.

Clearly, we need a test framework to run the automated database tests. The options include:

- **TSQLt** – an open-source T-SQL testing framework that operates within SQL Server.
- **Pester** – a PowerShell testing process that can work with your SQL Server instance.

This topic was covered in more detail in Chapter 8, *A DLM Approach to Database Testing*.

Bug tracking, workflow, and messaging system

Failed builds and integrations must be written to the issue-tracking system with as much detail as possible regarding the potential cause. The issue-tracking system should be closely integrated with a VCS and an associated workflow system, such as that available through Team Foundation Server (http://preview.tinyurl.com/j7ankvy) to allow the team to manage and assign issues quickly and efficiently.

For example, we should never issue further commits on a broken build, for reasons we discussed earlier in this chapter, so a typical workflow might, in response to a failed build, put the VCS into a "locked" state that will prevent checking in any more work. The issue will be assigned to a developer, whose task is to check in a fix that resolves the issue as quickly as possible and returns the VCS to an "unlocked" state. The team will also need a developer messaging system such as Slack or Gitter, to receive instant notification of build issues.

We cover requirement of the issue-tracking system (e.g. JIRA, Bugzilla) in Chapter 11, *Issue Tracking for Databases*.

How to get started with database CI

In order to encourage the practice of database CI, I recommend introducing the following two database CI processes:

- **Commit-based or "fast" database CI** – this process runs every time we commit a change to the database. I refer to this as "fast," as it is a fast build of an empty database. Subsequent tests load only sufficient test and reference data to do the validation. The intent is to provide initial, fast feedback regarding the validity of your schema and code objects.
- **Overnight or "full" database CI** – a scheduled database build, typically nightly, followed by data load, for complete end-to-end process tests, relational integrity checks, testing of migration that involve data movement, testing of ETL processes, and so on.

As you start running these automated processes, you'll encounter various forms of failure arising from bugs in the code, problems with data migrations, failed ETL processes, errors in the integration mechanisms that build or modify the database from the scripts in the version-control system, and more.

As you encounter, or anticipate, issues that could cause a failed integration, you're going to want to set up tests that catch further occurrences, and progressively refine the validation of your integrations.

In common practice, application code is generally not unit-tested against the actual database but is better tested against a test harness that provides a "mock" of the database. This ensures that it is the application code that is unit-tested rather than a potentially erratic database. Unit tests are best done quickly without any external dependencies. However, this becomes a problem, as described earlier, if the teams neglect to fully integrate and test a working database until much later in the development cycle.

It is better practice, in my experience, to run frequent continuous and nightly database integration from as early as possible in the project. For the nightly database CI process, the application may or may not be present. If the database and application are developed in step, then it makes sense that the application will be wired up. However, in cases where this is not possible, some of the integration tests can simulate the application. For a website, for example, we can generally run in sequence all the calls to progress a basket through the checkout process. Strive for a tight coupling between the application and the database on the CI integrations and tests, but don't sacrifice the speed needed to ensure fast feedback.

Between the fast (commit-based) and the full (overnight) database CI processes, you'll have a complete mechanism for testing your databases at the point of development and creation. Let's explore the mechanisms necessary to make either of these processes work.

Fast database CI

The intent behind fast (commit-based) database CI is to enable lightweight, fast testing of the database's code objects and structures, every time someone commits a database change. It simply performs a complete build of an empty database. Each time we run the fast database CI process, it will drop any existing database, and then generate a script that will build a new database at the required version.

Without running any additional tests, a successful build is, in itself, an important "test" that our fast database CI process can repeatedly build any version of a database from the component scripts in the VCS. However, it's essential to have the fast database CI process trigger a set of simple tests that will provide your first, immediate, feedback on the new database code and structures that you've created or modified.

Since these tests run on every commit, the emphasis is on speed. Some of the tests may load the minimum necessary reference or static data required to run the test. We can perform more extensive or complex tests, with realistic data loads, as part of the nightly integration.

What to test during fast database CI

The whole point of the fast CI process is to get immediate feedback in the case of failures and issues with the code. This means not using data at all as part of the tests, or having the test itself load only the minimal amount of "static" or "enumeration" data it needs to run. It also means keeping the tests simple and direct, in order to facilitate the speed of each integration.

The fast database CI process will verify that the database can be built, check for code policies, run "assertion" tests on functions, and so on.

For example, we can run such tests as:

- **Unit tests on stored procedures and functions** – essentially, assertion tests that access a single database object ("*If I pass in these parameter values, this is what I should get back*").
- **Basic constraint tests** – check that necessary table constraints, such as **`CHECK`** constraints are working properly.
- **Naming standards** – do all objects, column names, and so on follow the specified naming convention?
- **Standard code validations** – such as ensuring the use of **`SET NOCOUNT ON`** in stored procedures.

Of course, without any real data in place, we cannot run tests to validate things such as data refactoring, or testing of ETL processes. We also may not test at this stage any processes that require complex multi-table interactions. Treat tests within the fast database CI the way data used to be treated in old-fashioned client server environments. Only run the tests you need and only when you need them. Run enough tests so that the each fast database CI run is useful for the immediate feedback it's meant to provide.

When to run the fast database CI process

The tests that comprise the fast CI process must return results fast enough that the team can have confidence in running them many times during the day. As the name implies,

the ultimate intent is that they run on every commit of any database change to version control. You can also have a fast database CI mechanism that fires the tests based on:

- Merge of changes within version control.
- Branch within source control.

This means, of course, that you must automate capture of changes within your database, or a set of changes, from your VCS so that on each commit, it autogenerates a script that will build a new database at the correct revision. The exact mechanisms for this will depend on whether you adopt the state-based or migrations approach to database modifications, as discussed in Chapter 7, *Database Migrations: Modifying Existing Databases*.

Full database CI

The full (overnight) database CI process uses a database with a realistic and representative data load. For many tests, we can simply use standard data sets, loaded from flat files. For some tests, you may wish to use a cleansed and obfuscated version of production data. However you load it, though, I'm an advocate of using test data that is as close a match as possible to the volume, values, and distribution found in production data. The closer the match, the more likely that your tests will help you validate your code and processes prior to deploying to production.

The final section of this chapter discusses challenges surrounding provision and maintenance of test data sets, depending on test requirements, database size, regulatory restrictions, and more.

What to test during full database CI

With the database created and loaded with data, the full database CI process will encompass a much more comprehensive set of functionality and integration tests.

It will, for example, test the following:

- **End-to-end application processes** that need access to many related tables and objects associated with these tables.
- **Complex table refactoring** that requires careful preservation of data, as it migrates between tables, or between the old and new incarnations of a table.

- **Security testing**, such as validating that the code being written doesn't rely on excessive permissions that won't exist when the application is run in production.
- **Functional tests** of complex or problematic code.
- **ETL processes**.

In short, any tests that you can run in this longer, scheduled process, that will ensure improved quality on the code that you deliver to the next stage of the data lifecycle management process as a whole.

When to run the full database CI process

All the required steps for your full database CI process have to be automated within your CI server software. The full integration process is triggered on a regular schedule. Full database CI testing will necessarily take longer, due to the data loading steps, for example, which is why it's typical to run it nightly. Some teams might schedule it to run a few times a day if it's viable, and if changes to the code or metadata are being made.

Again, I stress the need for every step of the CI process to be automated. If your database CI process isn't completely repeatable, with a reliable set of results, then the feedback that you are getting from the process through the failure or success of an integration is rendered invalid.

It's entirely possibly for a manual build or migration to succeed once, and then immediately after fail if you run it again manually and make a mistake. If you don't make a mistake, all you've tested is your ability to repeat a set of tasks. Automation removes that uncertainty and ensures that your feedback loop through the CI process is accurately measuring your ability to generate meaningful and accurate deployments.

Of course, no amount of automated testing is a substitute for subsequent, often manual QA tests, which ensure the business is happy with the software you're delivering. An automated test can only check for problems you can predict, and coverage is seldom 100%.

DLM advantages of database CI

DM aims to help to get your database changes out the door alongside your application changes in a timely manner, and in a manner that ensures protection of the information stored in the database.

A database CI process provides a mechanism that ensures immediate feedback on the changes you're making to your databases. It lets you track your database builds, migrations and testing and ensures that you're testing your deployment process early in the course of software development, which will help ensure more accurate and successful deployments to production.

Instrumentation and error reporting

As discussed in detail in Chapter 6, *Better Ways to Build a Database*, it is vital for the build process to be thoroughly instrumented, providing the details the developer needs, to pinpoint the exact cause of the problem, possibly including the full stack trace.

The same requirements extend to the CI processes. You should publish the mechanisms of the CI process: when the integrations run, which tests run, and when, how you deal with the results, and so on, and make sure they are understood by everyone involved in the process. When a build fails, developers must receive automated notifications via email, messaging, or alerts directly within their IDE, and the system should provide log files and as much detail as possible about the cause of failure.

Failures should be reported to the team in real time. More than one person on the team may be responsible for a given failure. The feedback loop must be universal for all your team, not just the person who has currently checked in some code. The development team leader needs to be able to delegate tasks within a project, including tasks within the CI process, using automated workflows to assign broken builds to specific development team members.

You should also make public the graphs and reports that depict your build frequency, the rate of success and the rate of failure, as well as the number and severity of the bugs you've captured through your CI process.

All of this data will demonstrate to developers, operations, and governance alike what processes fail regularly, and therefore where to direct improvement efforts.

Auditing and governance

The team need to consolidate in a centralized place the record of the integration process, such as statistics from various tools or build outputs. This would allow supervision of the integration pipeline and various metrics associated with it. There ought to be an artifact repository to store all build outputs for easy access.

An example of why it is important is when you need an audit report of when a particular change happened, and why. You could get some of this from version control, but the important point is when the change was first successfully integrated. The CI process has to be able to automate traceability and governance rules such as code policies. This will also help you arrive at better measures of technical debt, and get a hand on delivery date estimates.

The CI process must be flexible enough to require additional tests such as functional tests, integration, or performance tests as deemed necessary, before release of a database from development into the deployment pipeline. Of course, performance and scalability tests will occur after the code is released for deployment to QA, staging and beyond, but if you can do some of these in integration than the chances of an easy passage through the delivery pipeline are enhanced.

It's all about determining the correct tests necessary to ensure that the CI process is assisting in your overall delivery of changes to the system. The correct tests will help to ensure that auditing and governance processes are well covered, early in the process. As with all the other tests run through the CI process, it's about providing the right feedback, fast.

Faster issue resolution

If issues are discovered in production, IT operations should be able to fall back to more stable builds that are retained in their central management server (see Chapter 10, *Database Deployment and Release*). In the meantime, many of the practices implemented as part of database CI will prove critical to a team's ability to respond quickly to the issue and to be able to put tests in place that avoid them recurring.

If you have the luxury of being able to use obfuscated production data as part of full data CI testing, then the team will need to become proficient in re-creating the production systems in a safe environment. In a DLM system, it is quite possible that the operations team will take on the preparation of "production-scale" databases, since they have the security access and clearance to see the real data, whereas the developers would never have this. This is another incentive to involve the operations team in the integration testing.

The ideal case may be to have a production-parallel environment in which all the events are synchronized except for those that cross the system boundaries, which are mocked with fake endpoints. With this in place, the initial reaction to a production error can be to take a production snapshot, synchronize the production-parallel environment, and set about reproducing the problem. The team needs to reproduce the database in the state just before the error, which might involve undoing a transaction set or preparing some external data.

This done, we can write an integration test that fails at the earliest point that our problem is visible, whether in application state or in persisted, database state, in order to prevent regressions and to validate the fix.

More consistent and reliable database deployments

By repeatedly practicing during development how to integrate database changes and create a working database, at the correct version, you will be in a far stronger position to start releasing to QA much earlier in the development cycle, and therefore receiving earlier feedback from DBAs and operations staff on potential security, performance, compliance issues, and so on.

The goal is that the database "package" to promote a database from one version to another, which our CI process produces, will be used for all environments, from development through QA and staging up to production.

As discussed in more detail in Chapter 6, *Better Ways to Build a Database*, one step toward this goal is to have a clear separation between the data and the database configuration properties, specifying the latter in separate configuration files, tested by your CI processes, and providing them as part of the database release package.

Of course, there are other possible difficulties associated with promoting, to other environments, the database changes tested via our database CI process, such as consideration of SQL Agent jobs, use of technologies such as replication, change data capture, and so on, which will not be present in the development environment. We'll discuss these topics in Chapter 10, *Database Deployment and Release*.

Common challenges with database CI

Databases present a unique set of problems to a CI process. The persistent nature of data is one of the biggest problems. Unlike code, you can't simply throw away the old database and replace it with the new one without taking into account that there is, or may be, data within that database, that you're going to need to retain as part of your deployment because it's necessary information for the business. You can't simply toss it away. If you throw in issues around having to deal with potentially large sets of data, or the need to create or duplicate data for testing, you're adding considerable time to the database CI process as a whole.

You also have to deal with the fact that you can't create, in place, a branch, a second copy of a complete database, but have instead to deal with this through other means. You're also going to be faced with cross-database or cross-server dependencies. In short, the same problems that application code has when dealing with databases during testing.

All these problems add to the complexity of the CI process when dealing with databases. The following sections of the book offer suggestions and mechanisms for dealing with these issues.

Mechanisms for providing test data during database CI

For many purposes, you will use standard test data sets, so that the data is the same on every test. The full database CI process may start with the outputs of the fast database CI process, and then import a data set from files. For example, the steps to test a single end-to-end business process, which accesses the database, might look as follows:

1. Run the fast CI process (tears down existing database, creates a new build, runs tests).
2. Import standard data set, for example using a tool such as BCP, or equivalent.
3. Test the business process.
4. Repeat.

However, it will ease the passage of a database through the various "release-gates" on the way to production, if the team can perform rigorous integration and acceptance testing with realistic "production-like" data from as early as possible in the development cycle.

Therefore, if possible, you may opt to start the full database CI process with a restore of the appropriate version of the production database, with data that has been appropriately cleaned in order to ensure compliance with any laws or regulations. The production version would then need to be migrated to the version under test. In this case, your full database CI process will need to automate database restoration, migration, teardown, (cleaning up after a test run) and obfuscation (cleaning existing data to protect production systems). You'd then test all of the required business processes using the restored database.

Of course, depending on the size of the database and regulations regarding use of the data, it simply may not be possible to use "production data," even obfuscated, in the development environment. The following sections discuss some of the challenges of obtaining and maintaining production-like data sets, for CI testing.

Obtaining a copy of production

There are a number of mechanisms for maintaining a copy of a production environment for High Availability/Disaster Recovery (HA/DR) purposes, such as mirroring, replication, or using Availability Groups within SQL Server. However, all these processes limit the target system from making structural and data changes because you can only read from them, not modify the structure or the data within them, which means that they are not suitable for our purposes within the CI process.

This leaves some type of backup and restore, or a process of moving data from the production database to your test databases, such as using import. The best mechanism here is to use the restore process to set up your CI database. However, you have an additional mechanism that you must perform: before you expose the production data through the process, you need to perform data obfuscation and cleansing.

Data cleansing

If you have to meet regulatory requirements to restrict access to production information, then you can't simply take a copy of your production backup and restore it down to your CI environment. Because we're going to expose the results of failures of the CI process to the entire team, that alone could expose sensitive data. You also have to deal with that fact that you might be testing functionality such as sending an email based on data inside your database. You don't want to chance exposing your clients to test emails. Both these issues suggest that you must have a mechanism in place that cleans the data prior to using within your CI process.

There are three basic ways you could address this:

- Clean the production data as a part of the CI process.
- Clean the production data in a separate process, preceding the actual CI process itself.
- Use functional data-masking mechanisms such as the data masks built into SQL Server 2016.

Perhaps the simplest way is to have a completely separate process that cleanses your production data and then backs up the clean database for use within your CI process. The data-cleansing process will modify or delete data as needed to meet any regulatory requirements. You need to remove any risk that your CI databases could still have sensitive information within them and could expose that information to unauthorized people as part of a subsequent deployment process. All the data-cleansing mechanisms and processes should probably be developed by individuals within the organization who appropriately have permission to view production data.

Finally, you can use internal processes exposed through the SQL Server database engine, such as Dynamic Data Masking, as a way to ensure unauthorized people, which can include your CI process, only ever see appropriate data.

Regulatory restrictions

As discussed previously, if you involve the operations team in integration testing, they may assist in providing preparation of production-scale databases for CI testing, regulatory compliance permitting, in a safe and reliable manner.

Nevertheless, the use of production data may not be possible in some environments. Even cleansed and obfuscated data can sometimes reveal damaging facts if that data is accessed illegally. In such cases, the security implications for using such data in a development environment would probably make it unviable. The alternative is to import, from flat files, data sets that mimic as closely as possible the characteristics of the production data.

Data size

If you have a terabyte or more of data, you're just not going to want to go through the process of backing that up and restoring it, let alone attempting to cleanse it, in order to maintain your CI process. This is one of the more difficult problems to solve when working with databases within CI. Your solutions are frankly somewhat limited. You can take production data and,

as part of your cleansing process, delete a substantial amount of data in order to shrink your database down to a reasonable size. You can choose to use data-load mechanisms to create and maintain a set of test data. You can look to third-party tools to provide you with a mechanism to create smaller databases through compression or other mechanisms.

Of course, there are drawbacks to this. You're not going to have a complete set of production data, so you might miss certain unique aspects of data or data relationships within production in your testing. You're also not seeing any issues with data movement and maintenance that are simply caused by the size of the data you're moving around. Unfortunately, these issues are not easily addressed and may just be somewhat problematic for your CI process. In short, CI isn't going to solve or prevent every possible problem you might experience in your production environment.

Taking advantage of a third-party tool to create small, or instant, copies of databases can alleviate these issues, but it adds overall complexity to the entire process of arriving at clean data ready for CI testing. However, it can be the best bet for keeping the size of the database down while still allowing for a full set of production data to be used for testing.

Time required to perform integration testing

The scheduled database integration tests ought to perform a complete test of modifications to your database schema and code (ideally with the application). These tests will also need to ensure that any migration scripts work correctly and preserve existing data. This means you will have to take into account the amount of time it takes to deal with your data as part of your full CI process.

You'll have to measure and account for the time that a restore of your database takes. If you're automating data cleansing as part of the CI process, then this will also have to be added to the full CI processing time. You'll also have to allow time for data migration and for additional testing time, including validation that your data movement was successful and you haven't lost any information.

All this suggests you need to attempt to keep the amount of data under your CI process as small as you can. Refer back to the issue with large data sets, above. These are your primary mechanisms for getting the time down to complete your full CI process.

The time needed to deal with data is one of the main reasons I advocate so strongly having a two-step approach to CI testing. The fast CI process ensures that you can have a continuous stream of fast validation of basic functionality.

You can then do the work necessary to set up the full CI process and deal with time and data issues within a wider time-frame, supplying the feedback necessary to validate your code changes, but not negatively impacting your ability to get work done.

However, you can certainly move any and all types of tests that might involve larger data sets away from the CI process. Just understand that you're making the choice to move your tests further away from the development process, which can, in some cases, make fixing issues more difficult and time consuming. The whole point of setting up a CI process is to provide a set of mechanisms for testing early in the development process. However, there is nothing that proscribes moving these tests out. Each environment will have unique challenges. You'll have to figure out how best to deal with yours, following the guidelines laid out here.

Branches within source control

Branching code within source control is common. Chapter 5, *Database Version Control* discussed this process in a lot more detail. Suffice to say, at least three branches are common:

- What's in production now.
- Hot fixes getting ready for production.
- New development.

As the application code branches into these three or more different sets of code, you'll have to branch the database as well. The only mechanism I have found for branching a database is to create more than one copy of that database. This leaves you with only a couple of choices. You can create multiple databases within an instance of your database server, using different names, or you can create multiple database server instances. Your environment and approach will determine which of these is best for you and your processes. Just remember this is going to exacerbate the issues around the size of your databases since, instead of one 100 GB database, you're going to have three 100 GB databases to move around.

Cross-database dependencies

It's not uncommon to have a query that retrieves information from a database created by a different application. One could argue that secondary sets of data should only be accessed through a secondary service, thereby eliminating the need for a cross-database dependency.

Until those services are created, though, you might have to deal with multiple databases as a part of your CI process.

The first approach I take is to immediately adopt a mechanism supplied by application developers; I create a stub database. This is a database that only has the necessary tables, views, procedures, and data that I need test the database that is going through my CI process. If my testing only requires a few tables or certain data from those tables, there's no need in copying around whole databases. Just create the necessary information, in a minimal form, and maintain that as a part of your CI processing. Chances are you'll only ever need this for your full (overnight) CI. If you have to add it for the fast, commit-based CI process, make it an empty or near-empty database.

Another approach is to use ODBC links when dealing with cross-server queries. You can then easily mock them using text or Excel-based databases, even using T-SQL to query them.

On the other hand, if you find that you're going to need complete copies of secondary databases, then you have to treat them in the same manner that you're treating your database at the core of your CI process. You'll need to get them into source control, if they're not already there, and build a full CI process around them, if one doesn't exist. This means dealing with their data, data cleansing, branching, and all the rest. Don't forget that this will also add to the time necessary to complete your CI process, so be sure to take that into account when setting this up.

These are not the easiest types of problems to solve. It's just best to take into account that you will need to solve them, depending on the requirements of your system.

Summary

DLM aims to help to get your database changes delivered quickly, alongside your application changes, and in a way that ensures protection of the information stored in the production version of the database.

Database continuous integration uses tooling and scripting to provide immediate feedback on the changes you're making to your databases, and allow these changes to be more visible and testable. You can give the database all of the benefits that you get from application CI. It will mean that you can validate all database changes quickly, using a frequent, automated, repeatable process. It means that you can reduce human error, as well as improve the database code and schema quality. It will reduce time between database releases and, above all, ensure that

the quality of those releases is high, with a dramatic reduction in the number of database-related issues reported in QA, staging and production.

Perhaps most significantly, it will help the team adopt a culture of continuous improvement for the database, not just the application.No one is suggesting that creating a CI process for databases is simple or easy. However, it is doable. While databases present some unique challenges within a CI environment, none of these challenges are insurmountable. Just remember that you will have to deal with your production data in a responsible manner. Focus on the need to get feedback from the CI process as a way of improving the quality of the database code you're delivering.

10 – Database Deployment and Release

So often, the unexpected delays in delivering database code are more likely to happen after the developers initiate the release process. The necessary checks and tests can turn up surprises; the handover process can expose deficiencies. With good teamwork, planning, and forethought, though, the process can be made almost painless.

This chapter describes how a piece of database code can be moved to production, painlessly and quickly, through the necessary testing environments, and then via staging to releasing the database code into the production environment. The more of the process of release that you can automate, the smoother the process becomes. The smoother the process becomes, the faster it will be, and the need is to be able to get code into production as fast as possible while still protecting the production environment. The release process outlined here will make that possible.

The release process aims to create a speedy method for getting code out of development and into production while, at the same time, also attempting to protect the production environment to ensure uptime and data integrity. To reconcile these conflicting aims, DLM recommends that you must stick to the release process tightly, as it's defined, and bring as much automation to bear as you possibly can on that process. Automation is the key to providing speed and protection for your software releases.

Governance, delivery and release management

We'll start by defining just a few terms that we'll use.

The successful delivery process reaches the point where the new application, the new project, the new piece of functionality, is ready to be delivered to operations, to put it into the hands of the business people to help meet a business need. A **release** consists of a series of deployments of code after the development process has determined that this code is potentially ready for production use. A **deployment** consists of the steps required to support this process by getting new code into the environments necessary to support the release.

To deliver this release out of development and into production via a series of deployments requires **governance** to define a process that will ensure that it is done swiftly, securely and safely. This is what **release management** is about.

Next, we'll describe the different delivery environments that your code may need to pass through on the way to production and why you need to have at least some aspect of some of these delivery environments within your process.

Delivery environments and their purpose

There are a few extremely rare circumstances where it is viable to release straight from development into production. Except for those unicorns, the rest of us are required to go through some sort of process prior to releasing our code into production. This means that, at a minimum, we have a development environment as well as our production environment. However, for most businesses, especially mid-sized and larger businesses, even more testing and validation of your code will be required in order to ensure that it meets the specification, is scalable, and performs well. This will require other types of delivery test environments prior to release into the production environment.

Test environments

Let's first talk about some of the mechanisms that can be used to set up a testing environment and then we'll talk about the different kinds of testing environments that you might use.

Testing environment setup

In a perfect world, testing wouldn't even be necessary. In a slightly imperfect world, testing would be easy to do, and the environment would exactly match your production environment, both in terms of processing power and in the quantity and quality of the data. However, none of that is true in reality. Instead, getting a testing environment set up involves a series of trade-offs that allow you to adequately simulate the production environment while working on smaller machines, frequently with less data that is also different from production.

The basic server setup for a QA environment is required to emulate the **functionality** of production, but it doesn't have to emulate its **performance**. This means that, if you're running

a particular version of SQL Server in production, you should also run the same version in testing. If you have Enterprise-level functionality in production that could be impacted by changes that are a part of this given release, then it's a good idea to have a testing environment with that same level of functionality, achieved through either installing the Development or the Enterprise version of SQL Server (this is not going to be an chapter on licensing best practices; to ensure your compliance, work with Microsoft). You would be unlikely to ever need to emulate all production functionality in all your various testing environments (we'll cover the different kinds of environments in the next section), but, in order for the QA/testing process to assist in protecting your production environment by providing meaningful tests against a production-like system, the behavior of your production systems will need to be emulated somewhere. Other than that, for most of the testing environments, it doesn't matter if you're running virtual or physical boxes. You could be running it all locally or within a Cloud environment. The physical set up of disks, memory, CPU, really won't matter that much in your testing environment, with a couple of exceptions which I'll outline in the next section. Server setup is easy. Data is the challenge in QA and testing.

For most purposes, the single best data set for testing is your production data. However, there are a large number of reasons why you're not going to be able to use production data within your testing environment. First, and most important, there are legal ramifications related to the exposure of certain kinds of data to unauthorized people. Next, depending on the functionality being developed and tested, you don't want production information within the test environment (imagine testing email processing with your production email list). Finally, the size of your production system may preclude having a full copy of the production system in your testing environment. For all these reasons, the data set you'll be using in testing can't be a precise copy of production.

Before we talk about the different mechanisms for setting up testing data, let's talk about some of the requirements for the data in a production environment. In order to support a repeatable, automated, set of tests, the data must:

- be small enough that you can quickly reset the QA environment for multiple tests
- have known data sets with known behaviors for test automation and repeatability
- represent the data in production accurately in terms of the type of data, its distribution across the database, and its functionality in relation to the code
- not expose any contractual or legal data in an inappropriate fashion
- have a structure that mirrors production in order to support testing of the release being deployed.

These limiting factors help determine which of the following methods you're going to use to set up your testing data.

There are a number of permutations and possibilities for creating and maintaining a testing data set, so don't feel like this list is all-encompassing. This is merely a starting point. Further, you may find that different environments at different times may need to use different versions of these methods or even combinations of these methods. Here's the list, in order of my preference:

- a backup of a clean copy of production
- a backup of a created data set
- a created data set for import.

Let's detail what each of these entails.

Clean production data

This is my preferred mechanism when the production data is not overly large. The process is fairly simple and easy enough to automate.

- Restore a production backup to a non-production server (possibly, your staging/pre-production server; more on that under the appropriate heading).
- Run an automated process to mask or modify any sensitive data.
- Shrink the database.
- Back up the database to a location separate from your production backups for use in your testing environments.

This approach is very simple to automate. Presumably, you have backups of production ready to go, and you can get them on a regular basis. How often you would schedule this setup of your QA data is completely dependent on your deployment process needs. That step should be easy to set up. Next, cleaning the data. This can be done through T-SQL, PowerShell, or SSIS. It's mainly a question of identifying the data that should not be moved out of the production environment and removing it, modifying it, or masking it within this copy of production. This step has to be automated in order to make it repeatable and reliable. You can't risk accidently releasing production information. The next step, shrinking the database, may not be necessary, but depending on the changes you've made as part of the data cleanup, the database probably contains less data in it. Shrinking it will make the process of restoring it to your QA system faster. Finally, you back up the smaller, clean database to a location so that it's available as part of your QA process.

Backup-created data

I'm calling it created data, but what I'm referring to is either an exported data set or a generated data set (automatic or manual) that is stored outside of a database. It's not production data. It's a different data set that you've had to curate or manufacture on your own. Here's the basic process:

- Create the data set.
- Build an empty database based on the version currently in production.
- Import the data set to the empty database.
- Back up the database to a location for your testing environments.

The hardest part of this process is not the task of creating the data set. The hardest part is maintaining the data set over time. You'll have to plan for changes to your data set when data structures change in production and as the distribution of production data changes. These will need to be taken into account. The next step of building an empty database should be made possible by pulling, from a known state, a defined version for your production environment using source control and labeling or branching. The next steps should be fairly straightforward. At the end, you have a database backup that is ready for use within any kind of testing environment, one that can easily be restored at the start of a testing process.

Import-created data

This is my least favorite approach because it's not as easy to automate across lots of environments and several deployments with multiple releases. However, depending on your circumstances, it might work better for you than using the backup as I've outlined above.
The starting point, and some of the pain, is the same. You'll have to curate or create a data set:

- Create the data set.
- Build an empty database in your test environment based on the version currently in production.
- Import the data set.

While it looks easier than the previous approach, it's actually more error prone. Presumably, with the previous approach, you're going to run it once a day or once a week or maybe even less. If it breaks, it breaks once, you fix it, and then you're good to go with all your testing deployments. Whereas with this process, if it breaks, it stops all deployments to testing until you get the process of data creation fixed again.

It's much easier to just use a backup of either clean production data, or created data, rather than trying to import the created data each time.

Regardless of the mechanism you use to create your test data, with an environment and test data in hand, you can automate creating your testing environment and have it accurately reflect your production environment.

Types of testing as part of release

Now, the question comes up: "What kind of testing are we talking about?"

Testing environment functionality

I'm very careful to separate the testing environment from the pre-production or staging environment. The needs and functionality of these two environments are different. I'll talk about staging in the next section. Your testing environment can serve a very large number of purposes. Each of these purposes might require you to set up a separate test environment, depending on the needs of the process and your environment. This list is also not all-inclusive, but represents the different types of testing that I've seen. In the order of its ubiquity:

- Quality Assurance
- Performance testing
- User Acceptance Testing
- Financial testing
- Integration testing
- Security testing
- Load testing.

Let's briefly run through these and note any exceptional needs they might have.

Quality Assurance

This is your standard testing environment. It's the fundamental and most common environment with simple processes where you have a development environment, a testing environment and production. The setup and mechanisms around this are pretty much as already outlined. In smaller shops, this acts as all the different types of testing environments, rolled into one.

Performance testing

Depending on the system, performance can be important or even vital. When performance becomes vital, it is essential to have a test environment where you can check performance separately from standard functional testing. The main thing about a performance-testing environment is that you do need to worry more about having the server configuration closer to production. It's not a requirement, but it is something you should consider.

User acceptance testing

The main reason for separating this type of testing from the QA testing is that it might take a lot longer than your QA cycle would allow for, and it involves exposing a wider range of people to the data. You may be doing QA releases once a day, but if the user acceptance testing isn't done, the process of resetting the QA environment could disrupt the tests. Other than the need for isolation and thorough data masking, there's nothing special about this type of testing environment.

Financial testing

I've seen situations where the finance department needed to be able to run multiple test processes on data. This requirement, like user acceptance testing, extends the testing window. Isolating this environment allows them to get their tests done separately in parallel. In some instances, I have seen this environment having actual, live production data. You may need to have a completely different security and data access regime if that's the case.

Integration testing

If you have a system with many integration points, meaning your application interacts with other applications, and/or your database interacts with other databases, you may need to ensure that you have a set of tests that validates that the changes reflected in the current release do not cause problems. The key issue with this type of testing is that you will need to be able to set up all of the various links between systems as part of your automated build of your QA environment.

Security testing

Security testing is frequently done as part of a QA process. This helps to ensure that changes won't compromise the security of the production environment. The challenge here is in adequately emulating the production setup without either causing issues with the testing environment or violating your production security.

Load testing

This type of testing is completely different from performance testing. In performance testing, you're validating whether the new functionality in this release either helps performance or doesn't hurt it. With load testing, you're intentionally finding the point at which the functionality being tested breaks down. The load testing I've done is either on an exact mirror of production, or on the production environment prior to releasing it to the business for the first time. Load testing on an environment that doesn't match production is likely to be a waste of time because the production environment will behave differently.

Pre-production/staging

I've separated the pre-production or staging environment (which I'll just call staging from here on out) because the purpose of this environment is very different to that of the QA/testing environments. Staging is specifically set up in support of a release management process to act as the door into your production environment. You might have more than one QA deployment in a day or week. You'll only have a single, successful, deployment to staging for any given release. The goal of this environment is to successfully test the release in a copy of production, immediately prior to releasing the product to production. That puts different requirements on this system.

- It must be a near match for the production environment.
- It must have data and structure as close to production as possible.
- It must be possible to reset it automatically.

Let's start with the need for this environment to match production as closely as possible. Because this is supposed to be the point at which a final release is tested prior to a deployment to production, you'll want this environment to mirror production as closely as possible in order to accurately test the production release. The data and structures also have to be as close as possible. In most cases, I've only ever used a restore of the production server, or a

cleaned up restore of the production server, similar to what I would do for a QA environment. If the size of your system is such that you can't have a staging environment, emulating a scaled-down version of production just won't do enough to justify having this as a separate step. Your QA deployments can then cover for staging.

Dealing with server-level objects

When you're defining a release for a database, most of the time the work will involve tables, views, and issues of data persistence. However, sometimes, you'll also be deploying new server functionality, added maintenance, or even import/export processes. All of that has to be taken into account when talking about a release. Further, each of the different environments you're dealing with will have a varying set of requirements concerning functionality and even security. Let's address a few of the issues that are going to have to be taken into account with your releases.

Server settings

For the most part, servers are something you set up once and then leave alone. There's very little maintenance normally required for a server. However, if you have particular server settings that you use, such as setting maximum memory, it's a good practice to do two things. First, script out each server setting you manipulate. Don't rely on the GUI as your mechanism for manipulating servers. This is because you want to always be in a position to automate your server maintenance and your server installations. Scripting all the changes is the best way to take care of that. Next, put these scripts into source control. Further, if, as part of your deployment process, you make different choices on setting up different environments, it's a good idea to script out all the changes and put that into source control, separated for each environment. Then it can be a part of your automation process.

SQL Agent jobs

DBAs use SQL Server Agent jobs for all manner of tasks, such as running background processes, maintenance tasks, backups, and ETL. Databases of any size will also have Agent jobs that are entirely part of the database functionality. Because of this, the scheduled tasks

that are part of the release should be stored in the development VCS rather than being owned by operations and stored in the CMS archive. Although there could be Agent job steps that have little to do with individual databases, many of them are involved with such jobs as involve ETL, replication, Analysis services, and Integration Services.

Agent jobs are generally, but by no means always, T-SQL batches. They can even be PowerShell scripts, ActiveX Scripting jobs or executables. SQL Server stores jobs for all databases on the SQL Server instance, all together, in the SQL Server Agent, in MSDB.

It is best to start with these jobs being in source control. It is not easy to unpick the scheduled jobs on a server with more than one database on it, to script out the ones that are appropriate to a particular database application.

It isn't a good idea to involve several logically separate databases with the one job unless it is an administrative server job such as backups or maintenance. For application tasks it is one job, one database. Otherwise, what happens when the database versions get out of sync?

Agent jobs often reference scripts in file locations. Different servers may have these in other locations so these need to be parameterized in the build script so that the build process can assign the correct file path.

It is wise to document the source code of Agent jobs along with the name and description of the job, and which database(s) it is accessing. Use the description to provide special detail, such as a PowerShell script on the server that is called from a PowerShell-based job and which should also be saved.

SSIS tasks that are called by SQL Agent must be in source control too, along with batch files that are called from job steps and need to be saved as well. Is that PowerShell script executing a file as a script block? (That means that the file must be saved as well.) Are server-based executable files involved? Someone needs to ensure all of this is in source control and then check they have not changed since the last deployment (the description field of the job step may help).

Just as with server settings, it is possible to use the GUI to set up and maintain SQL Agent jobs. While you might use the GUI in development of the jobs, each job, its schedule, and any and all steps within the job should be scripted out, and those scripts maintained in source control. Every aspect of the database should be treated with the diligence that you treat your code. The ability to get back to a previous version of a SQL Agent job is only possible if you store them into source control.

Once the jobs are in source control, you'll want to automate their deployment through the release process in the same way you will your database deployments. The difference being that there are few, if any, tools on the market that are going to assist in this part of the process. You'll need to work out your own mechanism for automation of the deployment of SQL Agent jobs.

Agent alerts

Agent alerts are generally more relevant to the server than the database, but they are part of the build process. For the application, it is a good practice to set up special alerts for severe database errors. Operations people will have definite opinions about this. Alerts should, for example, be fired on the occurrence of Message 825 (tried to read and failed), and a separate alert for each severity 19, 20, 21, 22, 23, 24, and 25. It is also wise to alert on severity levels 10–16, indicating faults in the database application code, usually programming errors and input errors. Usually, the end-user hears about these before the developers or operations people. There will also be various performance alerts, set up in order to alert the operator when a certain performance condition happens.

Alert notifications will change according to the type of installations, in order to specify the right recipient. For integration or UA testing, the database developers need to know about it, whereas in production, the production DBA will need to know about these. The build needs to take these alerts from the appropriate place, the development VCS or the operations central CMS archive.

Wherever they are held, we need to script all these alerts and, like jobs, we should use the name to signal which alerts belong to which database.

Server triggers

Server-scoped triggers track server-wide DDL changes, such as a database creation or drop, and login attempts to the current server. If specified, the trigger fires whenever the event occurs anywhere in the current server. Their main use is for server security and audit, but an application might use them for security purposes.

Linked servers

An application might use linked servers for ETL, distributed database layering or data transfer. They are sometimes used for archiving old data. We need to install these, with the application, and it makes things neater for the build process if we obey the following rules:

- One linked server, one database.
- Keep the source of the linked server in source control.
- Ensure that the creation or update is repeatable.

Security and the release process

The demands of security will require a high proportion of the work of automating a release through a variety of release environments. The definition of the release itself is made during development, so the main job is to shepherd around a predefined set of changes as defined by the release version in source control. However, each environment is likely to have different security settings. This security must be taken into account as you build and rebuild your environments. It also has to be taken into account when you deploy the code, since it might have different behaviors associated with it. Dealing with the differences in security can be somewhat problematic. There are some options available to help you automate this within SQL Server.

- Use of roles in databases as a method of defining security.
- Pre- and post-deployment scripts built into the automation engine.
- Contained databases.

Using a role is a good way to set up security because you can create a set of security behaviors that can be deployed with your database, regardless of the environment, and then all you have to worry about to give someone more or less access is moving them to a particular role. It makes the management of security much easier.

Pre- and post-deployment scripts are scripts that respectively run ahead of, or after, your deployment. They are a useful mechanism for dealing with security. You can create a T-SQL script that looks at the server name and uses that as the mechanism for choosing to how to set up security.

Contained databases allow you to define the users, roles, and their behavior within the database definition, and then have that stay within the database throughout the process. The security in this case would travel with the database through backup and restore processes, all without the need to maintain or create a login on the server. This isn't as universally useful as the other two tips, but it can come in handy.

Methods around automation of release

One of the goals when setting up a database deployment methodology should be automation. When you consider the number of deployments that you're going to be making, automation becomes a necessity. The mechanisms and tools used in automating deployments across environments don't differ much from the tools used for automating testing or automating a continuous integration process. The primary differences come from needing to take into account the fact that you have multiple different environments, with changing security, differing drives and server architecture. There are other differences that affect how any given script might run from one server to the next.

Scripts and scripting

Database deployments are done through T-SQL scripts. That's how the manipulation is performed. The choices you make when deploying across environments are affected by the way that your scripts get generated and run. One trick for T-SQL is to set up your scripts to run in SQLCMD mode. This is a mechanism that allows you to do several things that are designed to assist a deployment. First, you can run your scripts through your SSMS query window in the same way that they would be run if you were running them from the command line. Since you're planning to automate your scripts, probably using release management software or continuous integration management software, the command-line usage is assumed. Next, you can add additional commands such as **`:CONNECT`** to connect to a particular server as well as other commands to help you in your automation through stuff like where the output of scripts goes, subscripts to run, what to do on error, and more. Finally, SQLCMD allows you use **`:SETVAR`** which is a way to set variables. This is vital when automating scripts across multiple environments.

When the server name may be different, the file locations could change, or even the database name might be different, rather than having to edit each and every script, you want to be able to set variables and then simply modify the values in those variables. These can be set from the command line at execution time, in order to modify the behavior of your T-SQL without ever changing the T-SQL script in any way.

Automation engines

A lot of the same tools already discussed in Chapter 9, *Database Continuous Integration*, can be used when deploying across environments. You just have to take into account the various differences between the environments, and make handling them a part of your scripting process. However, one tool set is a little different, namely release management servers. There are a number of these on the market. Microsoft has a tool called Release Management Server built into its Visual Studio and also into its Team Foundation Services offering. More sophisticated and powerful third-party tools such as Octopus Deploy are also available.

These tools offer you two things:

- enhanced abilities for automating deployments across environments
- the ability to track what has been deployed to each environment.

Just to use Octopus as an example, it provides you with the mechanisms for creating multi-step deployments with the facility to mark different steps for inclusion on different release environments and/or different servers. This makes it possible to create a deployment process that takes into account all the various permutations that each environment requires. Further, you can easily set up multiple steps to allow for different types of deployments. All this can be automated through PowerShell within the tool and calls to the tool from PowerShell. This kind of behavior makes it very easy to set up multi-server automated deployments.

Then, the tool will track which deployment has been run on which servers. You can check whether or not a deployment was successful simply by looking at a report. You'll also be able to track the versions in use across all the different environments, since you're never going to see just one set of changes in flight toward your production systems. It makes it far easier and less error prone to be able to manage all these various and sundry systems if you know what has been deployed where, or even what has failed to deploy, and where.

Protections for production

The goal of all this is to get a release successfully into your production system. All the automation and all the testing are there to help ensure a deployment to production that doesn't fail. However, things can still go wrong. You might have unique data in the production system that responds poorly to a change script. You may see drift due to emergency patches or out-of-stream updates to the servers which can cause a deployment to fail. It's necessary to build protection for your production servers into the release process. Automating each of these protections as part of your deployment process is straightforward, depending on the protection. There are trade-offs with each protection system that require discussion.

Backups

The standard protection for databases on your servers is likely to be backups. Automating a backup before a deployment is a pretty standard practice in most environments already. You just need to be sure you're taking into account whether or not you're using differential backups, in which case, you will have to be sure to use **`COPY_ONLY`** for this out-of-process backup.

There are a couple of issues that come up when using backups as your protection mechanism for production. The first is time. It can take quite a while for a full backup to complete. Then, it can take just as long, or longer in some cases, for the restore to be run in the event of a failed deployment. You need to make sure that the time it's going to take for the restore doesn't violate any kind of Recovery Time Objective or SLA you have to maintain with your business.

Another issue that comes up with backups is when the failure of a deployment is found. Sometimes, it's found immediately. The deployment script fails and you need to immediately roll back any changes made. Sometimes though, you don't find the problem until days later. In this case, you can't simply restore the old database because so much other data has moved on. This is when you'll need to use a different approach.

Snapshots

Another kind of backup is available, depending on your hardware or your SQL Server instance. Some SAN systems have a mechanism to reliably create a snapshot of a database that can be used for a restore process. If you're running the Enterprise version of SQL Server you can perform a database snapshot. The beauty of these snapshots is that they're very fast to perform and very fast to recover from. The only shortcoming is the same as with backups. If the problem with the deployment is found later in the process, the snapshot is no longer a good recovery mechanism.

A/B deployments

Sometimes called Blue/Green deployments, the A/B deployment process is straightforward. You have two servers or two databases, one of which is currently online. You deploy all changes to the second database and then, after the deployment is successful, you switch the databases so that the updated one is now online.

This process works extremely well, but the shortcomings are obvious. You're going to have to have double the storage requirements. If you have a transactional system, you have to have mechanisms in place for coordinating the data migration so that the A and B databases/servers are in sync. In my experience, this doesn't work at all well with Online Transactional Processing (OLTP) systems. I have seen it work very well for reporting systems where the data is refreshed nightly, but no other data changes are made during the day.

Rollback scripts

A frequent process put in place is to have a script ready that will remove the changes you've introduced to the production system. This can be a very effective approach since you can take into account exactly the changes you're making, in order to provide a script that protects existing data. It makes the rollback somewhat safer than either snapshots or backups.

The main problem with the rollback approach is that you're going to have to spend just as much time developing and testing the rollback as you do developing and testing the deployment scripts. You don't want the first time that you run a rollback script to be on your production system. That completely violates the process of testing and automation that you've put in place. Further, rollback scripts can suffer from the same shortcomings as backups and

snapshots when the problem with the deployment is found long after the deployment is completed, making the rollback script invalid. This is especially true if the problem with the deployment is at all isolated. Instead, you're much better off using the following approach.

Fail forward

Along with backups and snapshots, my preferred mechanism is to use fail forward, or roll-forward scripts. In short, when you find a problem with a completed deployment, instead of attempting to undo that deployment, simply apply the mechanisms you've created for developing new code to quickly test and put into production an immediate fix to the problem.

With this approach, you don't want to abandon the traditional protections of a backup or a snapshot. Their utility is well tested and well defined. However, the overhead and time involved in creating and maintaining a rollback script is extreme. With all the testing and protection mechanisms you have in place for creating your automated release, it is very rare that you will need to use a rollback script. If you can be confident that you have a plan to get a fix in place rather than try to undo what you've done, then you will save considerable time in the development and deployment process.

An example release process

There are many ways in which you can mix and match various tools, techniques, processes, hardware, and software in support of a release process. Any process I define here is unlikely to work perfectly in your environment because of the variety of software development processes, tool-sets, or business requirements. However, in order to illustrate how you would bring together all the various tools and techniques outlined in this chapter, I want to illustrate one example of a process flow from development to production. Take this as an example, as a starting point for developing your own processes, not as an ironclad definition of the "right way" to do this.

Defining a release

The release environment that I'm going to work with assumes that we have all our database code in source control alongside the application code, and that within development we have a fully functional CI process in order to validate our code changes prior to release.

A release is defined as a set of code that has passed our CI process. If there are any failures in CI, then we cannot release this set of changes. The code is either branched or labeled at the conclusion of the CI process, and the QA team is informed that a release is ready.

QA

When a release is defined as ready, the QA team will decide if they want to deploy that release. If they're in the middle of testing another release, they may defer implementing this one. Although I've also had QA teams who just did a weekly build of the last good release and always worked from there.

However the QA team decides to take on a release, the process is initiated from Octopus Deploy. The process will restore the QA database from a clean copy of the production database that is created nightly through a separate automation process. The steps within QA are as follows:

1. Restore a clean copy of production database.
2. Get the defined release version from source control.
3. Deploy the release to the new QA database.
4. Run automated scripts that validate whether the deployment is successful.
5. Alert the QA team that a successful deployment is ready for testing.

If the QA build fails, an alert that there's something wrong with the latest release must be sent to the development team. Additional testing may be required. In the event of a failure in QA, the previous successful release is defined within Octopus, and it will be used to re-create the QA system so that the QA team can continue their work.

Staging

The staging system uses the same process as the QA system. The only real difference is the one additional step. Instead of simply deploying the release, an artifact must be generated. This can be as simple as a T-SQL script, or more complicated: executed as a sequence of scripts and commands. The artifact is what is used to perform the staging deployment. If the process is successful, that same artifact is used in production. This approach avoids using tools against the production server that rely on automation to ensure the deployment of the release in production. This is a level of paranoia that has served me well.

Production

Only appropriate aspects of the production deployment are automated. The creation of a database snapshot and the deployment of the artifact generated in staging are automated, but the execution of this automation process is triggered manually. Validation scripts must also be run in order to ensure that the production deployment was a success. If there is any failure at this point, then the snapshot is used to immediately restore the production instance. This approach does assume down-time during the release.

Summary

DLM aims to make all the delivery processes repeatable and reliable. The release process is central to delivery. The first stage is to determine beforehand, as part of the governance process, the details of the release process that is required in order to meet the requirements of the data and the application. Once everyone is clear about the tests, checks, alerts, logs, and release-gates, then these processes must be automated where appropriate. The release process is one of the most important to get right because it has to do two things that, to a degree, are diametrically opposed. It's attempting to create a speedy method for getting code out of the development and into production. It's also attempting to protect the production environment to ensure uptime and data integrity. Just remember, however you define your process, two things are necessary: you must stick to that process without deviating from the way it is defined, and you need to bring as much automation to bear as you possibly can on that process. Automation is the key to providing speed and protection for your software releases.

11– Issue Tracking for Databases

Any database development project will be hard to manage without a system for reporting bugs in the code, anomalies and incidents from live environments, and for keeping track of new features. In this chapter, we discuss the critical role of an integrated issue-tracking system in DLM.

An issue-tracking system is a repository of all the issues found in the application or product being developed, and a record of all the steps taken to remove the problem. Also recorded are the "amelioration" steps, the necessary precautionary steps to make sure the effect of any issue or defect is minimized if it is already in the production system.

An effective issue-tracking system is critical for any project, but is especially important for larger projects where communication is the key to rapid delivery. Within organizations where databases tend to be at the hub of a number of different teams and processes, there are many advantages to managing issue tracking properly.

Issue reporting

Database issues can be uncovered by a variety of test exercises, such as unit tests, build, integration test, performance tests, or scalability tests. As well as being identified by testing, issues can come from helpdesks, operations, and monitoring, as well as from feedback from the application itself, from existing end-users, or even reports in the media. As soon as an issue is reported, its existence is alerted to the team. This can be done by email, or preferably through a team-coordination tool.

When an issue is reported, it must be recorded. The report should, if possible, include the identity of whoever reported it, when and how it was found, and the steps to reproduce it. Enough of an investigation must be made to make a reasonable attempt to answer the following questions.

- **Reality**: Is the issue confirmed, or is it an artifact, possibly due to faulty hardware, incorrect installation, network issues, and so on?
- **Ramifications**: How important is the issue? Is this, for example, going to impede the work of the development team? Will it affect time to delivery?

- **Complexity**: How much time and resources will it take to fix? How complex is it, and how easy to reproduce and test?
- **Severity**: Does the issue affect production systems (e.g. does it cause a security vulnerability; is there a risk of financial loss or legal action against the company)?
- **Location**: Where in the database application, as near as possible, is the issue occurring?

In the light of these judgments, several actions need to be taken. A severe issue that affects production systems will require a very different scale of action to a peripheral edge-case that is relatively harmless and unlikely to be seen in production.

At this stage ownership needs to be assigned.

A lifecycle is normally attached to an issue. There are a variety of workflow systems around the issue-tracking and fixing process. A project manager will seek to make sure that issues are fixed in order of importance, but relative importance is always a matter of judgment.

Issue-tracking systems provide valuable feedback to managers and other parts of governance who need to get a feel for the backlog of a project, and even some of its technical debt.

Problems caused by poorly-controlled, reactive issue tracking

At its most rudimentary, an issue-tracking system is little more than a "dumping ground" for bugs. It means that bugs and other issues are less likely to get lost, but no other benefits are gained. Some organizations fall into the habit of keeping track of issues by sending round a flurry of emails when an issue occurs or a new requirement is raised. Although email can be useful for following up on specific issues, tracking issues by email does not scale, keeps details hidden, and prevents issue-tracking activities from integrating easily with other tooling. Common problems with this approach include:

- **Issues tend to be recorded in an unsystematic way** – issue descriptions are poor; vital information such as the steps needed to reproduce the issue are skipped.
- **Issues tend to be "owned" by the tester** – whoever finds the issue is assumed to be responsible for ensuring that the issue is fixed.

- **Issue backlog grows rapidly** – no system in place for allowing the developer the time and resources to fix the issue; no adjustment of deadlines and "diamond points" for delivery of functionality.
- **Issues fester** – hard to determine when an issue is fixed. Again, this requires careful testing and assignment of time to do the work.

With this instinctive, chaotic approach to issue tracking, the chore of bug-fixing can get unbalanced because the allocation process is so haphazard. The developers most willing to fix bugs experience the brunt of the burden, and risk slipping behind on their development work. In the worst cases, these developers begin to be considered guilty by association, and even accused of creating the bugs. The "blame game" can get entirely out of hand.

It is much more effective to have a process in place that can be supported by an issue-tracking tool so that the process is, at least, repeatable.

Integration into a DLM approach

We see the most benefits of database issue tracking when it is integrated with the issue-tracking systems of applications, and when other tools in the database lifecycle can report issues to it. These contributing tools might include the version-control system, documentation systems, user feedback systems, ETL systems, downstream reporting and analysis, and so on.

Any issue-tracking tool that we choose should have an API so we can easily integrate with other tools and extend the behavior of the tool when new approaches require it. Whatever is used, we must be able to automate aspects of reporting and issue management via the API.

When an integrated and efficient issue-tracking system is in place, it becomes instinctive to check it after any change to the production system, such as the application of patches. A sudden increase in reports of issues can quickly alert to a deployment problem even if there is no apparent link between the symptoms of the issue and the original action.

Integrating the issue tracking of application(s) and database

It is often not straightforward to pin down the exact cause of the issue. A database problem can, for example, cause mysterious application problems that are not immediately attributable to the database. Also, issues that seem to be caused by the database are often eventually found to be within the application, and the converse is just as true. It sometimes happens that a database problem can cause mysterious application problems that are not immediately attributable to the database, but can be quickly spotted by a database developer, such as when database warnings are discarded by the interface.

If database issue tracking is integrated with the application issue tracking, then database experts, whether they are developers or DBAs, can help a great deal in quickly working out what is causing the issue and where it is happening.

A common problem we see in IT departments is that each team has been free to choose its own issue-tracking tool. The development team usually chooses a lightweight tool that integrates with version control but that has little support for production incidents, while the operations/DBA team tends to choose a tool that excels at service management but has little or no support for version-control integration or Agile/iterative software development. The result is that knowledge is difficult to share, and cooperation between DBAs/operations people and developers suffers in consequence.

The issue tracker you choose to support DLM should be accessible to both development teams and operations/DBA teams, as well as to application support staff, and it should have the functionality to support all their requirements. Tickets need to be visible and shareable between developers and operations/DBA teams (where a "ticket" means the details of a single issue, bug, or requirement that we track with our issues-tracking tool). Ticket workflows need to be configurable and flexible; at a minimum, tickets must be cross-linkable between dev-focused areas of the tracker and ops-focused areas, even if dev areas use different workflows to ops/DBA areas.

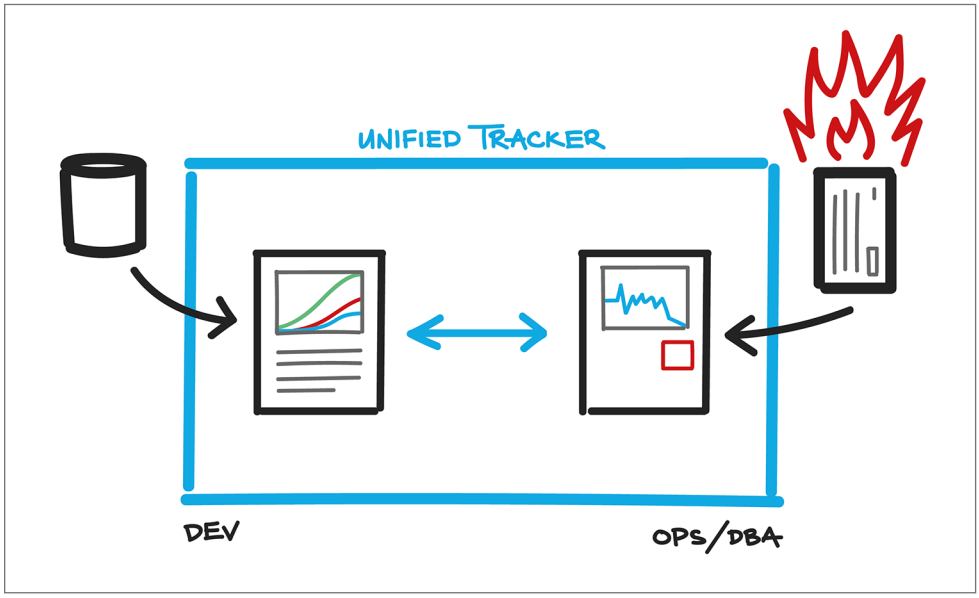

Figure 11-1

Sometimes, it's not possible for the two teams to use the same issue tracker, due to vastly different requirements. Some dev teams work best off of simple prioritized lists, whereas the operations team need to track a lot of different moving pieces that all need to be dealt with at once. In such cases, at least give each team access to the other's system, so they can collaborate.

It's very important that bug amelioration (workarounds) should be visible to support, and also that the right people are able to see issue reports that could end up being financially or legally damaging.

Whichever tool you choose to use, make sure that knowledge and awareness of issues are not kept in separate silos, and that the way in which the tool is configured helps with collaboration and communication between different groups.

Be wary of issue-tracking tools designed for service desks

Service desks with an IT operations team are often set up to issue tickets associated with support calls from a single individual. Default ticket-access permissions tend to hide ticket details from everyone except the person who first happened to report the problem. In my experience, these tools are not suitable for use as issue trackers for team-based development.

Atlassian JIRA (http://preview.tinyurl.com/l5c9ctc) is a commonly-used tool for issue tracking, for shops that have mature software delivery approaches, such as DLM and CD. Other strong tools are Team Foundation Server (TFS), a manual Kanban board, and AgileZen.

Atlassian JIRA has third-party integrations and programmability via an API, making it easier to configure different workflows for different teams – Agile, Kanban, Service Desk – so avoiding a very common trap of having a different issue-tracking tool in each department.

VCS integration

The most obvious advantage that is offered by a DLM approach to issue tracking is in the integration of the issue-tracking system into version control. If a change to an object is made in response to an issue report, then the ID of the issue is placed in the header of the code. When this is saved into version control, this alerts the tester, who will then need to confirm that the new build fixes the issue.

Issue trackers that integrate with version control can provide end-to-end traceability for database and application changes. For example, the issue tracker tool you choose should be able to scan commit messages in your VCS for ticket IDs. This means that, within the web UI, comments and collaboration on commits can be seen alongside the commits, which helps to make technical decisions flow through version control.

VCS integration with issue tracking: examples

GitHub's integrated issue-tracking system looks for GitHub IDs in Git commit messages of the form **#Num** (e.g. **#123** or **#71**), and auto-links to the corresponding ID from within the application (see http://preview.tinyurl.com/kk9kg7u).

The private code-hosting tool CodebaseHQ (http://preview.tinyurl.com/m7b2ee9) uses [touch: NUM] as the syntax in commit VCS messages for referring to issues in its tracker; a commit message of **"[touch 71] Fix the URL formatting"** will link the commit with issue #71, adding details of the commit to the issue tracker.

Other tools have similar schemes.

Integration with customer feedback tools

When a team practices short, frequent cycles of database software delivery, based on customer requests and feedback, it becomes important to create direct links between these requests and specific issues in the tracking system. The team is on the lookout for opportunities to link issues and user requests, as reported using a tool such as UserVoice (https://www.uservoice.com). If a feature is not widely used, according to usage tracking statistics, then the request or reported bug can be de-prioritized.

In essence, we extend the traditional definition of a bug or issue to encompass "deficiencies" in the way that existing features work, i.e. indications that the software doesn't do what the users need it to do.

Make it as easy as possible to process issues

The team needs to make issues as easy as possible to process, thereby avoiding the buildup of a massive backlog that nobody then has the energy to fix.

We'll cover a few ways in which we can improve the issue allocation and processing efficiency here, but it will also involve efforts to improve the instrumentation of database code and processes, and improve the quality of code reviews.

Use meaningful issue descriptions

The title of the ticket should contain enough detail that other team members can understand the context of the ticket. Generally, it should generally enable the team to do top-level prioritization of tickets in the issue tracker without looking at the full details.

Titles such as "Test failures" and "Stored procedure needs updating" do not convey much useful information. Better examples include:

- Data load fails when number of locations exceeds 12000.
- Extend the Customer table to include Twitter handle.
- Recent new index on `DeliveryPreference` causes weekly data extract to fail.

Cross-link to other tickets where appropriate and include screenshots, sections of log messages, and extracts of configuration files where these provide context for other people looking at the description.

Include details of how to reproduce the problem

Issue tracker bug or incident tickets often have a "steps to reproduce" section. These are immensely useful when attempting to debug the problem, and also help to remind the person who creates the ticket to double-check that the problem really exists.

Use it! It helps to avoid duplicates and false alarms. Also, after you have written out the steps to reproduce, follow the steps yourself again, in order to verify that the problem is indeed exactly as you describe.

Tag issues that relate to audit and compliance

For tickets that may relate to areas of the system that are geared for the requirements of auditors or compliance people, use search tags that help aggregate groups of related tickets. For example, if you are creating or updating a ticket that relates to personally identifiable information, then consider using a tag like "PII" or "SensitiveData" so that people who need to know about these areas can be informed easily.

Make all tickets visible to all people and teams

Everyone involved in building and operating the software systems should be able to search and view every issue. At least all people in the technology department should have access, perhaps not to modify, but certainly to comment on all tickets.

As a rule, perhaps 1 in 500 tickets might contain sensitive data, but such tickets should be dealt with on an exemption basis or, better, with the details moved to a secure area. The security needs of a tiny fraction of tickets should not override the need for transparency and sharing between teams. This implies that "service desk" setups (for example, to fix problems with the CEO's Blackberry or iPhone) should be separate from issue trackers for core software systems. It also means that if parts of the delivery and operations process are outsourced, some teams will need to use issue trackers of either their client or their supplier.

Automate tests to re-create bug conditions

The development team needs to explore ways of creating automated tests to re-create bug conditions before bugs are fixed, to speed the process, and to instrument the database in such a way that alerts are fired if the bug recurs, or similar bugs appear. For example, in order to reproduce the conditions for a bug, we may need to create an automated test that generates

memory pressure on the target database server. Automation of testing will free up testers to allow them to participate in the end-of-life of a bug by certifying that it can no longer be reproduced as a result of the bug fix. See Chapter 8, *A DLM Approach to Database Testing* for further coverage of this topic.

Have a very clear escalation strategy

Database bugs can cause corruption and data loss that can be very damaging to the organization. Most likely, your business needs the problem solving now, if not sooner.

Enter: the hotfix. A hotfix is a single, cumulative package that is used to address a problem in a software product. DBAs deal routinely with hotfixes that are applied to SQL Server to fix a specific problem such as a security vulnerability.

DBAs are also used to doing controlled changes to production databases, most commonly to fix a performance problem. A hotfix is a perfectly normal outcome from an issue in order to resolve it. It is merely a workflow item in the resolution of an issue and has to be tracked in the same way as a patch, an amelioration, or a new release.

At some point in the lifecycle of an issue, the decision has to be made as to when to apply the fix to the database system. It can just be included in the next release. It may have to be done as a hotfix to the current production version as well as being enshrined in the next version of the database. In certain cases, the hotfix solution will be different from the permanent solution.

Sometimes, though, they have to be done urgently outside office hours. These hotfixes must then be retro-fitted into the issue-tracking system so that the change to the system that was made is not lost to the system. Any subsequent change to the source code needs to be tagged to the hotfix so that everyone is reminded of why and when the change was made. The system must be told that it was added in retrospect to prevent a security/auditing alert.

Automate documentation of fixes

If the documentation process that associates a fix with a particular release is not automated then it can quickly become too time consuming. A database developer can develop many ways of minimizing the chore. The comment headers of routines such as views, functions, and procedures are usually the most convenient place to apply notes about the issue that a change resolves. These can then be read via SQL code and used to update the issue-tracking system. If the database developer uses a temporary user identity for each issue, it is easy to

subsequently read the default trace to extract all the DDL code that was used to make the change. This can then be pasted into the issue report.

Perform "autopsies" for future protection

Each issue should have an autopsy which looks at ways of protecting the database system against similar problems, to find out what lessons can be learned. For example, if a table is successfully accessed by a user that was not supposed to have access rights, then it is worth considering if there are other related vulnerabilities that exist, or whether the entire user-access strategy is wrong. The accent is on prevention.

Reports need to be driven by the essential information of describing what happened, when, who fixed it, and how they went about doing it. If a similar incident happens subsequently, this helps the debugging process a great deal.

There is more that can be done, of course. It is possible to triage bug and incident reports, looking for repeating patterns, related tickets, and longer-running problems. Track incident priorities (P1, P2, P3, etc.) along with metrics such as Mean Time to Detect (MTTD), Mean Time to Resolve (MTTR) and Mean Time between Failures (MTBF).

Your goal is to improve the system across the board, not just in specific areas. Regular ticket reviews help to gather undisputable metrics that form the basis of informed decisions rather than guesswork.

Summarizing the benefits of DLM-based issue tracking

Keeping track of new features, testing anomalies, and incidents from live environments is an important part of DLM. In fact, issue tracking addresses all the TRIM aspects of DLM:

- **Traceability and visibility** – issue tracking provides a rapid way to search for known problems or possible causes, surfacing details of system behavior.
- **Repeatability** – issue tracking helps us to focus on capturing enough detail to be able to reproduce problems.

- **Improvability** – issue tracking provides a focus for improvements.
- **Measurability** – issue tracking helps us to track metrics around different kinds of problems and requirements.

As the teams gradually integrate issue tracking into a DLM approach, it becomes accepted as being intrinsically part of the database development process and the management of the process becomes a team responsibility.

Issue tracking provides traceability

Effective issue tracking helps us to provide this traceability by linking version control commits to an issue-tracking ID, and by associating sub-tasks with an over-arching issue tracker ticket ID. The issue tracker becomes part of a "single source of truth" for traceability and audit. Essentially, we can build change management off of issue tracking, though they shouldn't be the same system, since the former requires the information in a different form (an executive summary and less detail, typically).

However, for software systems that are subject to external regulation, such as in finance, healthcare, credit card payments, and so on, or internal regulation, such as for due diligence prior to a takeover, an efficient issue-tracking system can often provide proof that changes were made only according to an approved requirement or request.

Issue tracking helps us to make better decisions

To give value, the issue-tracking system must provide a coherent way for changes to be linked together so that people can easily discover the wider context for themselves and therefore make decisions from a more informed viewpoint. If this isn't done, it can happen that, by the time that DBAs receive a request for change or a schema update script, the DBA has very little context for the change; the business driver for the update has been lost somewhere upstream, perhaps with the development team.

Issue tracking should provide insights into our software systems

A good issue-tracking system will help a development team that is already working well. Otherwise, it will become a tiresome chore that produces white noise. They are most valuable when they can quickly show who made any particular change, and why. They also sometimes turn the nagging suspicion that a bug "rings a bell" and might have happened before into a golden route to fixing the issue. If the system is easily searchable it is possible to avoid re-fixing almost identical bugs over and over again. We often need to look for trends or past examples of issues in order to help us to understand current or future problems. Good issue-tracking practices can help to provide these deep insights into our software systems, allowing us to draw inferences about behavior and incident causes.

Summary

By adopting good issue-tracking practices, we aim to reduce problems relating to testing, deployment, and live operation of database-driven systems. Over time, we are able to identify areas where the cost of maintenance is high, where deployments regularly go wrong, and where we have the wrong skills-mix within a team. By collecting data and information methodically in an issue-tracking system, we can address these problems in concrete terms with statistics and charts, rather than relying on people's fallible memory.

12 – DLM for ETL Systems

Any database that is supporting an organization is unlikely to be self-sufficient, getting all its data only from the one or more applications that it serves. It will probably also be getting data from a variety of other sources and publishing its own data to other destinations. These Extract-Transform-Load (ETL) processes tend to grow in an unplanned, organic way and often cause trouble, both in production and in deployment. DLM systems allow all the teams to come together to ensure that ETL systems meet all requirements.

In the past, databases were considered to be "silos" of information, supporting a single application. In the last 30 years, they have developed to become participants in a rich ecosystem of data, publishing only the data for which they are the primary source, and subscribing to all other data. Because of the essential unreliability of networks, data tends to be pulled into the database, and pushed out in batches, and checked as part of the "pulling in." Hence the term "Extract-Transform-Load."

A lot of thought is put into understanding data within organizations, and there are established disciplines within Information Technology, such as Master Data Management or MDM (http://preview.tinyurl.com/hcnqe6p) and Data Lifecycle Management to create an understanding of the nature and source of data within organizations, and how and where it is processed and exchanged. This often leads to "data warehousing" where the data is collected in a sort of self-service data compendium. To do this effectively requires a meticulous knowledge of the type of data, format, usage, ownership, and confidentiality. As an alternative model of exchanging data, Microservices and SOA have been seen as ways of avoiding the curation of data; they actually put the onus of serving "owned" data on the individual service.

This puts ETL at the heart of DLM, since the three activities, governance, delivery and operations, need to work closely together to make it happen effectively; not just in the active phase of development but throughout the life of the database. IT governance has the task of maintaining a logical model of the data within an organization, the context within which individual applications exist, and ensuring that the handling of data is appropriate, and conforms to legal and regulatory frameworks and guidelines. Delivery and operations have to work together to create a system that works reliably, securely, and efficiently, and is easy to maintain and monitor. All three groups need to ensure that there is a common vocabulary in discussion and documentation so that misunderstandings are avoided.

ETL without design

The introduction of DLM might seem an unnecessary and expensive overhead to a simple process that can be left safely to the delivery team without help or cooperation from other IT activities.

However, few organizations, when designing their OLTP systems, give much thought to the continuing lifecycle of the data, outside of that system. They don't consider how they are going to transform and aggregate data from the various sources around the organization into a central data source that, through data mining, can help support and drive their business decisions. Even in cases where many core processes around database development and testing are automated and managed, the ETL system remains in the "Wild West."

The result is chaotic ETL, featuring periods of intense and often frustrating effort around critical data-loading periods. ETL processes will typically run long, and often fail completely. The business will complain that the data is incorrect or not available when required. The ETL team will respond by making ever greater efforts to ensure that the overnight batch completes in time. Eventually, however, the truth begins to dawn that the problem is due to the fundamental lack of design in the system.

If this situation sounds familiar, then this chapter will help you understand the specific challenges associated with ETL systems, and the DLM patterns that work well when attempting to develop, manage and maintain them.

Tripping over terminology

I have direct experience of the confusions, and sometimes communication breakdowns, between teams brought about largely by semantic misunderstandings. ETL is a classic case where the ETL team tend to use terms differently from other teams.

With ETL systems we want to build a data set, not an application, so the challenges and the terminology are different. In ETL systems, people often refer to "production" not as a live system serving traffic to end-users, but rather as the means to generate ("produce") the next batch of data. Likewise, in ETL systems, the build phase comes at the end of the ETL process when we build the reporting data set, whereas in transactional systems, the first thing we typically do is to build the software from version control.

In ETL, we typically stage the new data in a staging area before pushing the data to the reporting (live) system; this means that in an ETL context, staging has a very specific meaning which may not align with the term "staging" used for transactional systems, where staging is often a synonym for pre-production.

Overview of ETL systems

ETL refers to the process of transferring data in bulk from one system to another. ETL systems extract and transform large volumes of source data into a new data set, and then load that data into a new data store, for subsequent querying. This process happens because we ingest data from a variety of sources (some internal, some external to the organization), and we need to reformat or regularize the data fields before we can do further processing downstream. Source data might include:

- Order placement data from an eCommerce web application.
- Data from a customer engagement system such as a CRM like Salesforce.
- Anonymized patient care data from hospitals across one country or region.
- Customer financial data relating to credit-worthiness.

It may come as data files from other applications, from open data sources on the Internet, from services on the Intranet or Cloud, or from a type of data warehouse.

ETL sounds like three distinct phases but, in fact, the phases overlap and it's not necessarily a serial process. The users of ETL systems are always other software systems or people involved in data analysis, rather than interactive customers or end-users.

Figure 12-1 offers a highly simplified depiction of an ETL system. Each black arrow represents an ETL task as data arrives from various sources, is transformed, and loaded into a data warehouse. The ETL process often combines data from multiple different providers and, in many cases, an ETL destination can also be a data source for another ETL process. For example, the SSAS cube may be a data source for an upstream OLTP database, providing aggregated and calculated values for key business metrics, as the raw data rows are archived.

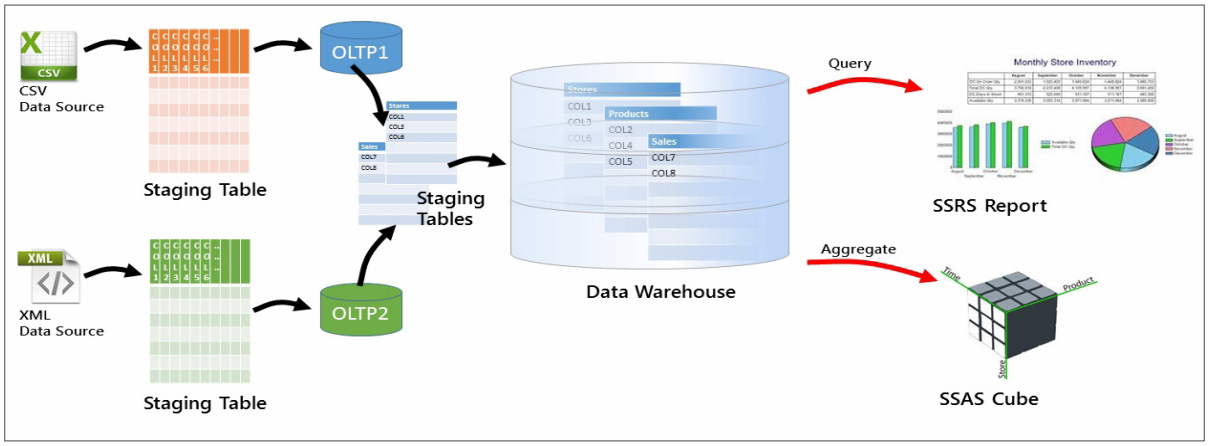

Figure 12-1

In Figure 12-1, we transport the data into staging tables in the staging area. At this point, we may well just "dump" the data into a staging table without any check constraints, using a fast, native bulk export method such as provided by bulk copy program (bcp) utility.

The recipient server has the responsibility of checking and, if necessary transforming, the data, and can sometimes delegate this task to a separate server or to the ETL process itself. If the database is using a staging table, we transform and load data row by row into our target database, in this example a relational database. For this task, we will use an ETL tool, such as SSIS. It will need to perform a range of transformation tasks, depending on the target, such as:

- merging columns
- splitting a column
- pivoting data
- data type conversion.

We will also perform some data validation at this stage. For example, we must define constraints on each of the columns in our target database, to ensure the data confirms to our business rules. If a row fails the constraint conditions, it is written to a log table for inspection, and the process continues. We will also apply or maintain all appropriate indexes on the target tables.

ETL workflows

Compared to transactional systems, ETL systems do not really have an ongoing "live" state, but instead typically operate on a data-load cycle that might be daily, weekly, or monthly, usually containing a critical period where data must be loaded within a specific time-frame in order to meet internal or external deadlines. For instance, one ETL team may have to process data for an internal BI system, delivering once a week for a finance meeting of the executive team, while another ETL may need to meet a monthly deadline imposed by a market regulator to report on the previous month's transactions.

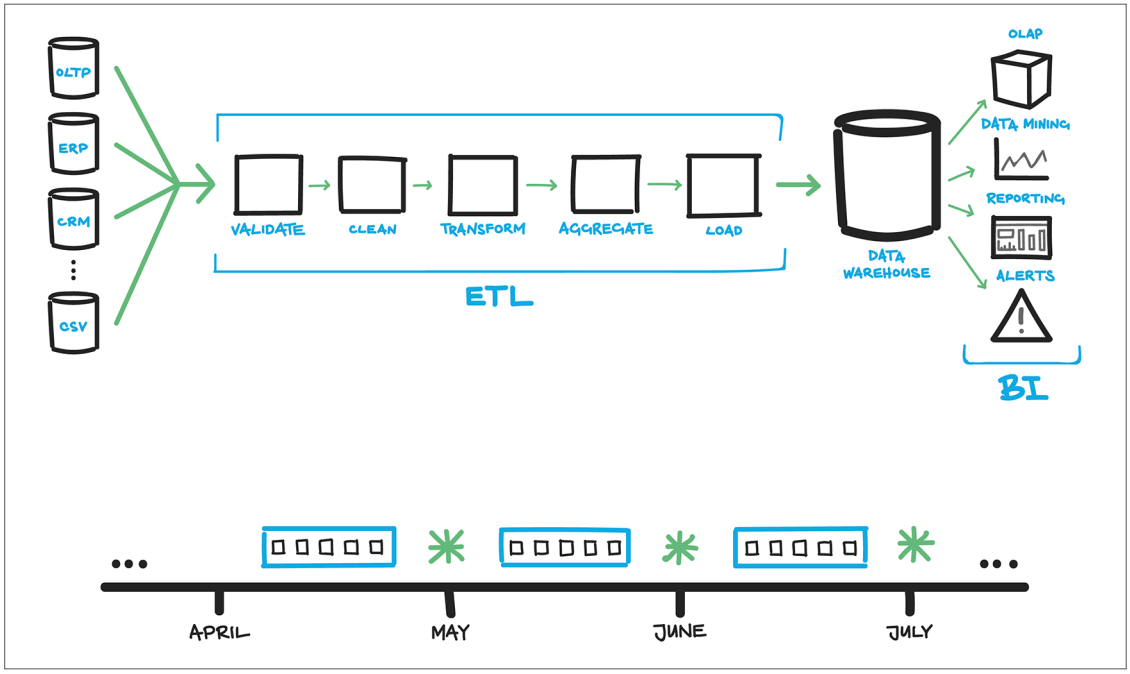

Figure 12-2

In the example shown in Figure 12-2, the live environment is the reporting databases used by data mining, business intelligence, data alerting, and so on. To make sure that access to this live reporting data is not interrupted, ETL teams typically use a staging area, where data quality tests are carried out before the data is loaded to the reporting environment. Data quality is assessed at Extract, Transform, and Load phases before presenting to the reporting environment.

Typical problems with poorly-managed ETL processes

When ETL processes are failing or overrunning, the development team will often take a long time to admit there is an issue. When the business complains that the data is unavailable or incorrect, the developers will tend to explain it away as an unusual event due to unexpected issues that caused the batch to overrun.

Production staff such as DBAs who are tasked with supporting the system will try to keep the system running and attempt to optimize the component ETL processes but, due to the cyclic nature of ETL, it is easy for teams to get trapped within a reactive cycle. Fundamental improvements in the design of the ETL processing pipeline can be difficult to justify or prioritize because the bottlenecks are often heavily data dependent; problems are blamed on the data rather than the ETL process's capability to deal with that data.

I call this the "JFDI" approach to ETL and the following paragraphs characterize the three most common problems associated with this approach, and the resulting "chaotic" ETL processes.

Lack of knowledge of upstream and downstream systems

Typically, I've found that each team has little to no knowledge of the data processing and reporting requirements of the teams that work upstream and downstream in the "data pipeline." This means that upstream changes to the data schema frequently break downstream ETL and reporting systems.

The ETL processes will also tend to be fragile, and prone to failure as a result of any changes in the format of the source data, or due to "bad data" in the source. Often these problems don't manifest until a large proportion of the source data has been processed. When it happens, the team have little choice but to make manual fixes to the data, or ETL processing stream, and then run the whole process again. Often, adjustments to the ETL process have a subsequent effect on the BI, reporting, and alerting systems that rely on this data.

Long and unpredictable ETL processing times

When an ETL process exceeds its allotted time, rogue processes will be identified and an attempt made to optimize them but these often ignore the underlying problem that there is another process running at the same time that is interfering with it. I've seen cases where a modest, nightly batch-load job took 4 hours to complete. This had never been questioned by the team; it was "just how long it took." Deeper investigation revealed that the load process was running at the same time as a job that ran database integrity checks. Simply by tweaking the scheduling, we reduced the time for the batch load to 1 hour.

In the production system, as multiple processes attempt to insert data into, and read data from, the target database, there will tend to be blocking and deadlocking. As the number of data sources, and the volume of data, grow, so the ETL processes grow ever longer, until it is no longer possible to complete them in the allotted time-window, meaning that the data is not available in the target database when the business needs it, and downstream reporting processes fail and have to re-run.

Long "fail-slow" ETL test runs

Many ETL systems are not amenable to fine-grained testing. When testing the ETL process, the team might, for example, restore a backup of the full source data set, perform an "end-to-end" ETL run, and then run a few tests in the staging area in order to validate that their processing rules worked as expected. It often means that tests can run for a long time before eventually failing. This also leads to long cycle times for ETL improvements and fixes.

Taming ETL systems with DLM

Next, we'll review steps we can take to help tame, and then systematically improve, our ETL processes, and establish the "TRIM" pillars of DLM:

- **T**raceability/visibility – processes are visible to all teams, and to the broader business, from very early in the project.
- **R**epeatability – processes are automated, tested and, therefore, predictable and repeatable.

- **I**ncremental Improvement – processes are connected by workflows that provide the necessary information to drive continuous short cycles of improvement.
- **M**easurability – processes are instrumented and logged so that errors and deviation from predictable behavior are corrected quickly.

Improving process visibility and measurability

One of the first steps is to explain the issues to the business and give them the means to judge how well you can fix it. Of course, visibility also includes the sense of instrumenting the systems to provide timings of the various jobs and any other metrics that can give a warning of impending problems.

Document existing knowledge

The first job is to collect and make visible to the business all existing knowledge about the system. The support staff will almost certainly have some statistics about what has been happening. Often they will have reports of long-running SQL statements from the overnight run which they have been looking to optimize. This is a good start, but make sure that this is documented. Usually a few processes will be found to be causing major issues due to their long-running. Often these processes are not the underlying cause but are being blocked by others running at the same time. After a few iterations of optimization, the system can be improved. A note should be taken of problem tasks and how long they take to run which will be important information for a strategic solution.

Publish performance metrics

As soon as possible, start to introduce some basic monitoring to gather data about each major task, and especially when the batch system starts and completes, and whether there are any errors. Each task needs to have a log of the start and end times. This should come from the scheduler – the SQL Server Agent maintains this in the history tables but this needs to be extracted into a log table for review.

Make this visible to the business, preferably with graphs, so that they can see when the system is available and can also start to see decreased batch processing times as progress is being made. It will also take some pressure off support staff and start to rebuild the relationship with the business.

Instrument ETL processes

A key factor in the long-term success and continual improvement of our ETL processes is **instrumentation**.

For each task, we need to be able to "flick an on/off switch" to start collecting the performance and runtime metrics that will allow us to track the progress of tasks, troubleshoot failures quickly, and establish baselines by which to gage recent performance compared to past performance.

If the duration of each ETL task is not monitored, then the team will inevitably fail to spot that an ETL job has suddenly, or gradually over time, increased in duration. Clearly, a variability in the data volume must be accounted for, but a measure such as "Duration per 1000 records" is something to watch over time to catch performance deterioration.

In addition, it's important to track the delays between ETL steps, particularly if there are manual or manually-triggered processing steps in the ETL workflow. There is little point in performance-tuning an ETL transform step from 2 minutes down to 40 seconds if the following step is routinely not triggered until 3 or 4 hours have passed because the step is waiting for a member of the overworked team to inspect the output.

There are several possible ways to instrument ETL processes. In the *Resources* section at the end of this chapter you will find articles describing how to build instrumentation into SSIS packages, as well describing a tool called **DTLoggedExec**, a separate instrumentation tool for SSIS.

However, a general-purpose ETL instrumentation technique on the Windows platform may just require the use of performance counters and PowerShell. This allows existing tools to monitor the progress and behavior of the ETL operation.

Improving predictability

The foundations of predictable and stable ETL processes are:

- proper planning for expected data volumes
- understanding every ETL task that needs to run, and when it can run without blocking other tasks

- parallelizing data extraction and digestion as far as possible
- agreeing a data interface "contract" to prevent schema changes breaking ETL processes.

Plan around data volume

One of the difficulties of many ETL processes is the volume of data that must be handled. The ETL team need to be able to estimate the volume of data that the system needs to load now, and will need to load in a year from now, in order to plan for both current performance and scalability into the future. When ETL processes are chained, or run in parallel, a single process can, by running for an unexpectedly long time, affect other processes so that the total time period escalates.

Prevent excessive blocking/deadlocking between tasks

A fundamental challenge for ETL systems is that they will often be running various tasks, with varying frequencies and durations. Some of these tasks need to run concurrently in order to complete the required processing in the allotted time window. Extraction tasks can run as frequently every 5 minutes, depending on the reporting requirements. Likewise, transform- and load-related tasks will each need to run on a set schedule. Often, different tasks will compete for data in the same table. For example, at the same time today's data is being imported into relational tables in the target database, another process may be reading data out of these same tables for export to an SSAS cube. In minimizing blocking as far as possible, timing and scheduling of the various tasks is crucial.

In his article, *The ETL from Hell* (see the *Resources* section), Nigel Rivett describes a tactical approach to streamlining a chaotic ETL system, based on establishing and documenting the following:

- **Processing time windows** – when does the data need to be available and when can processing start?
- **A comprehensive task list**, with dependencies.
- **External dependencies** – e.g. on availability of external data source.
- **Internal dependencies** – dependencies on other ETL tasks.

- **Grouping of tasks into "business threads"** – for example, in Figure 12-1, one thread is the set of tasks required to extract data from the csv source, transform and load the data into the staging tables, and transform and load the staging data into the data warehouse.
- **Measure "serial" processing time** for each thread – establish size of problem.
- **Identifying and tuning problem statements** – add logging to a trace table for statements that may be an issue. Include number of rows affected, start and end time of a task, plus statement start and end times.
- **Efficient batch scheduling/parallelization** – with automatic scheduling of each task, based on dependencies within fixed time windows, with built-in logging and alerting.

The most important task is to get as much baseline information as possible. How long is a process likely to take? What is unusual? It is a good idea to alter the schedule for a run so that a problem task runs on its own. This will give an idea of how long it needs, and also whether or not it is affected by other tasks that are running on the system.

Parallelize data extraction

Some ETL systems have evolved to require that all data is ingested before the transformation phase begins. Sometimes this is necessary, but in situations where several independent data sets are being imported, this serialization can be unnecessarily costly if a problem is found in the extraction/ingestion phase.

In order to detect problems with inbound data as soon as possible, parallelize the extraction of data from independent data sources. When loading large volumes of data from a single source, it may also be possible to break down that source data file into multiple smaller files and process them in parallel.

Parallelizing data extraction as far as possible helps the team adopt a "fail fast" capability. In the case of problems, they can choose to re-run just the failed data set, or even go ahead with the ETL processing for the rest of the data, rather than being forced to re-run the whole import.

Likewise, when querying (extracting) data in the staging tables, for loading data into the data warehouse, we should take advantage of database-level parallelization.

Prevent upstream data schema breaking downstream reporting systems

A very common problem with many ETL systems is a lack of predictability, and a resulting fragility that means frequent changes to the data scheme break the ETL process, so that the data is not available to upstream or downstream reporting processes when it is required.

Typically, the ETL team fits "between" the source transactional or third-party data and the reporting teams, such as BI. ETL teams often have limited communication with teams that build or operate upstream systems on whose data the ETL processes rely. For external upstream systems, i.e. those belonging to other organizations, this is understandable if unfortunate, but in many organizations the ETL team has limited influence over changes to internal upstream systems too. For instance, a team developing a new user profile feature on a web-facing eCommerce system may have limited or no awareness of the needs of downstream systems such as ETL, and make a change to the OLTP data store that will break the ETL process on next run.

If a supplier of the data for ETL frequently changes the data format, it can be very difficult for the ETL team to distinguish real errors in the ETL processes from errors relating to the changing data, effectively leading to a continuously "moving target" for the ETL team. This tends to be a particular problem when the data supplier is actually a paying client of the organization for which the ETL team works. The client will often supply the sales data, consumer credit data, purchasing data, and so on, in whatever form they like and expect the data processing organization to "just deal with it," since they are paying for that service.

This is really a business problem that has a negative impact on the ETL team's ability to improve the quality of the ETL processes. Without a schema or agreed data format, however loose, we cannot sensibly test the input data to an ETL system, and so we are beholden to apparently random errors that are actually the result of the changing data format. This unrecognized data dependency between OLTP and ETL/BI systems can lead to conflict between, not just teams, but entire C-level budgets (CIO, COO, CTO, and so on).

Insist on data format / schema standards for ETL

A key aspect of fail fast for ETL is to have clear data format or schema standards against which incoming data can be tested. In essence, the consumer of the data, such as the BI team, agrees a data contract with the supplier of the data, such as the ETL team who, in turn, agree a data contract with the internal or external supplier of the data.

A lot of this work should be done during the early governance stages of a project. The governance activity will need to identify the downstream consumers of the data in the planning stage, and agree by discussion the full interface, based on the corporate data model.

The format specifications act as living documentation for the ETL system, detailing the nature of the data and also the areas of the inbound data likely to be problematic.

Agree data format changes by negotiation, not compulsion

Neither the ETL team nor the data supplier, especially if that supplier is also a client, should dictate changes to the data format, as this leads to avoidable wasted effort dealing with mismatching data. Instead, agree an approach where the impact of changes is understood and both sides work together to minimize the effects of new formats.

If a supplier cannot produce data to an agreed format, create an additional step in the ETL process to sanitize and check that data specifically, carefully tracking that extra processing time.

Use contract testing to avoid breaking downstream systems

A well-proven, although little-known, development pattern for groups of data producers and consumers is the "consumer-driven contracts" pattern, which "*imbues providers with insight into their consumer obligations*," meaning that a team that provides a service or data gets just enough of an insight into the needs of a team whose software consumes their service or data.

Consumer-driven contracts work like this: each team provides to the team immediately upstream a schema definition for incoming data, plus a test harness so they can test that any ETL changes don't break the agreed data contract.

Consumer-driven contracts

The use of consumer-driven contracts to guide testing of explicit and implicit data contracts is based on ideas first articulated by Ian Robinson of ThoughtWorks in 2006, although used for many years prior to that by some teams. Martin Fowler discusses these ideas on his website at: http://preview.tinyurl.com/klur8v6.

The test harness might take many forms but, for example, the text-representation of most types of data, such as a postcode, or ISO-format currency value, can be tested by a regular expression (regex).

A regex is, after all, merely a formalized language to describe text patterns. If the downstream consumer can tell the supplier of the data what the format is, then it reduces the likelihood of misunderstanding.

The provider of the data, such as the ETL team, will use a test harness supplied by the BI team, for example, to test any changes to the ETL process, getting early indication if their changes will break the downstream processes. When plugged in to an automated deployment pipeline, the consumer-driven contract tests can act as a reliable early-warning system to upstream providers. If the tests fail (go red in the CI system), then the team providing the data or service knows that they need to fix something because, from the viewpoint of the consumer, something has broken.

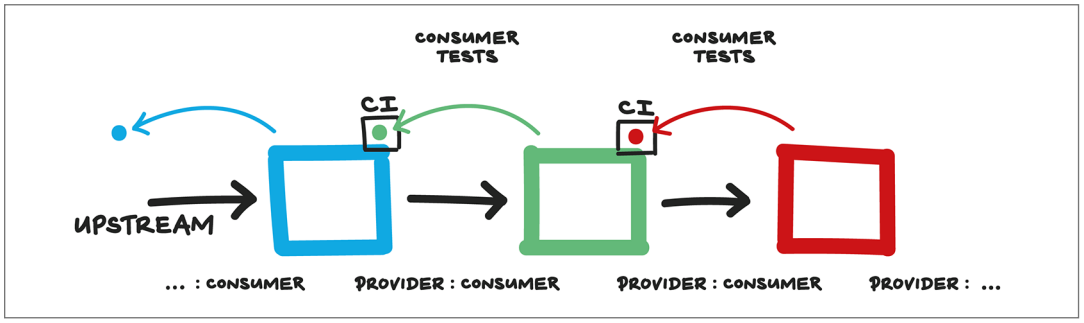

Figure 12-3

Similarly, the ETL team will act as a consumer of the data from upstream databases, providing a test harness to the development team responsible for the order processing application; the development team then uses the test harness from the ETL team in their deployment pipeline in order to check that any changes relating to the order application have not broken the downstream ETL processes.

Enforce validation checks during extract phase

Some ETL processes, like many fragile software systems, assume perfect, sanitized input data, and therefore may fail if unexpected data is ingested during the extract or transform phases. However, the failure may occur several minutes, hours, or even days, into the processing phase, necessitating the processing to be re-run after manual data fixes. This is a fail-slow approach that can waste a significant amount of time.

For instance, if we know that data from one supplier is often "flaky" for customer telephone number data, then we build an explicit check for that data field for that supplier's data. By identifying faulty or problematic data early, we buy more time for fixing the problem, either within the ETL team or by pushing back to the supplier for improvements. If the incoming data contains errors detected by our schema or format standards, we can often shunt the data to a queue and carry on with other tasks until the data is fixed.

As discussed in the earlier section, *Overview of ETL systems*, one simple approach to data validation is to define constraints on the columns of the target tables and divert to a logging table any rows that fail to meet the constraint conditions.

Continuous improvement of ETL systems

Having tamed a misbehaving ETL system, we can begin during the various development cycles to introduce the DLM techniques that engender a culture of continuous improvement.

Implement fine-grained testing for each ETL task

Many ETL systems are not amenable to fine-grained testing at each stage, but require a full end-to-end run in order to validate the processing rules. Of course, all ETL systems require full end-to-end testing; you'll get many unpleasant surprises if you don't check against the full expected load. However, if every test run is end-to-end, then it can lead to long cycle times for improvements and fixes. It also means that greater numbers of ETL programmers are needed than would be necessary with increased test coverage due to extensive rework of data loads.

We need to decompose the end-to-end ETL run into smaller, isolated sections, each of which can be run or re-run independently of the other stages. To help with this, we store the results from intermediate stages, at least until the full ETL run has completed, and possibly a while longer, to help with defect fixing.

Alongside stage decomposition, we can use a standard, test-first approach to developing the transformation and processing code:

1. Write a failing test that describes what the output *should* be but is not yet (as we have yet to write the code!).

2. Write the simplest code that makes the failing test pass, using a small data set targeted at that scenario.
3. Refactor the code to improve its modularity and focus.
4. Repeat from 1.

With sufficiently small and focused data sets, it's possible to evolve the ETL logic using a test-first approach without needing full end-to-end tests; one major benefit is that the tests capture clearly our expectations around what the resulting data should look like at each stage, helping us to reason about the processing required.

Consider using SQL "Unit Test" frameworks for ETL code

Some ETL teams prefer to move processing logic out of the database and into application code, allowing them to use standard unit test frameworks such as JUnit, NUnit, and so on. For ETL teams that prefer to keep processing logic in some flavor of SQL, sensible use of a SQL unit test framework can help us to decompose our processing steps *and* retain confidence that all the steps are still working as expected, even when we make changes weeks or months later.

In conjunction with small, decoupled, focused tests, we can use a SQL Unit Test framework for our ETL code (these frameworks are not really testing at the same unit level as application code is tested, but the "Unit" name has stuck).

The following table details some of the more common unit test frameworks and their availability by database vendor.

		DB2	Oracle	PostgreSQL	SQL Server
tSQLt	http://tsqlt.org/				Yes
SQL Test	http://preview.tinyurl.com/cbx987y				Yes
pgTAP	http://pgtap.org/			Yes	
SS-Unit	ttps://github.com/chrisoldwood/SS-Unit				Yes

		DB2	Oracle	PostgreSQL	SQL Server
SQL Unit	http://sqlunit.sourceforge.net/	Via JDBC	Via JDBC	Via JDBC	Via JDBC
utpPL SQL	http://sourceforge.net/projects/utplsql/		Yes		
SQL Developer	http://preview.tinyurl.com/jcdv78l		Yes		
SQL Server Test	http://preview.tinyurl.com/hforfkp				Yes
TST	https://tst.codeplex.com/				Yes
Dell Code Tester for Oracle	https://www.toadworld.com/products/code-tester		Yes		
Ruby-plsql-spec	https://github.com/rsim/ruby-plsql-spec		Via Ruby		
Db2unit	https://github.com/angoca/db2unit	Yes			

The essential premise with SQL Unit Test frameworks is that the stored procedure or function is the unit of testing, and that the parameters to the function or procedure act as the boundary.

Use small, focused data sets for testing specific conditions

A problem typical to many ETL systems is that teams either are not given the time or do not have the necessary experience to create specific data sets for testing purposes. There is often a sense that the whole data set is needed in order to exercise the system sufficiently for meaningful pre-live testing.

Using a backup of the full data set each time for testing tends to imbue the system with a kind of mythical aura that works against reason and decomposition, and hides, from both the ETL development team and the users (BI), details of how the system really behaves. This seems

to be because, when the team tries to use a smaller data set, certain problems are not caught during testing, and these omissions are then blamed on the lack of the full data set rather than the lack of specificity in the smaller data set. In other words, a series of smaller, more focused data sets can and should be used instead of the full data set for the majority of data use-cases, and the full data set used only to identify new edge cases for upstream testing.

This is a general problem where an organization's approach to testing is not advanced. It is not a problem that is specific to ETL alone, but becomes particularly problematic for ETL if the data volumes are large.

Fund ETL as enabling a key capability, avoiding time-limited projects

The "project" funding model, with pre-allocated budget, a fixed scope, and a fixed deadline tends not to work well for the ongoing development and improvement of ETL systems (this holds for many kinds of software systems too, including BI). The reasons for this relate to the motivations of project managers and others involved in the initial project: because they are being measured on "time + scope + budget" rather than the effectiveness of the software system that results from the project, they have little incentive to improve the system, and every incentive to push through workarounds in order to reach their deadlines.

Avoid project-based funding for ETL systems, and instead prefer a model that drives continuous evolution and improvement, with a funding stream over 2 or 3 years. ETL provides an organizational capability around data-driven decision making or data-rich applications, and should be funded as befits that capability.

Use Value Stream Mapping to highlight areas for improvement in ETL pipelines

Value Stream Mapping is a Lean manufacturing technique popularized by Toyota and used to identify delays and bottlenecks in the flow of materials and execution of activities in the course of manufacturing a product. In the case of physical manufacturing, a Value Stream Map captures activities and delays from the raw materials, through the factory or assembly plant, to the transportation of the goods to the end customer.

Value Stream Maps help to target improvement efforts by highlighting the areas where time or materials are being wasted, allowing the organization to improve the areas that are actually bottlenecks, rather than guessing at where improvements should be made. It turns out that we can make use of Value Stream Mapping to help us identify waste and delays as

we develop our software, a technique advocated by Lean software experts Mary and Tom Poppendiek back in 2003 in their book, *Lean Software Development* (http://preview.tinyurl.com/lmufky2).

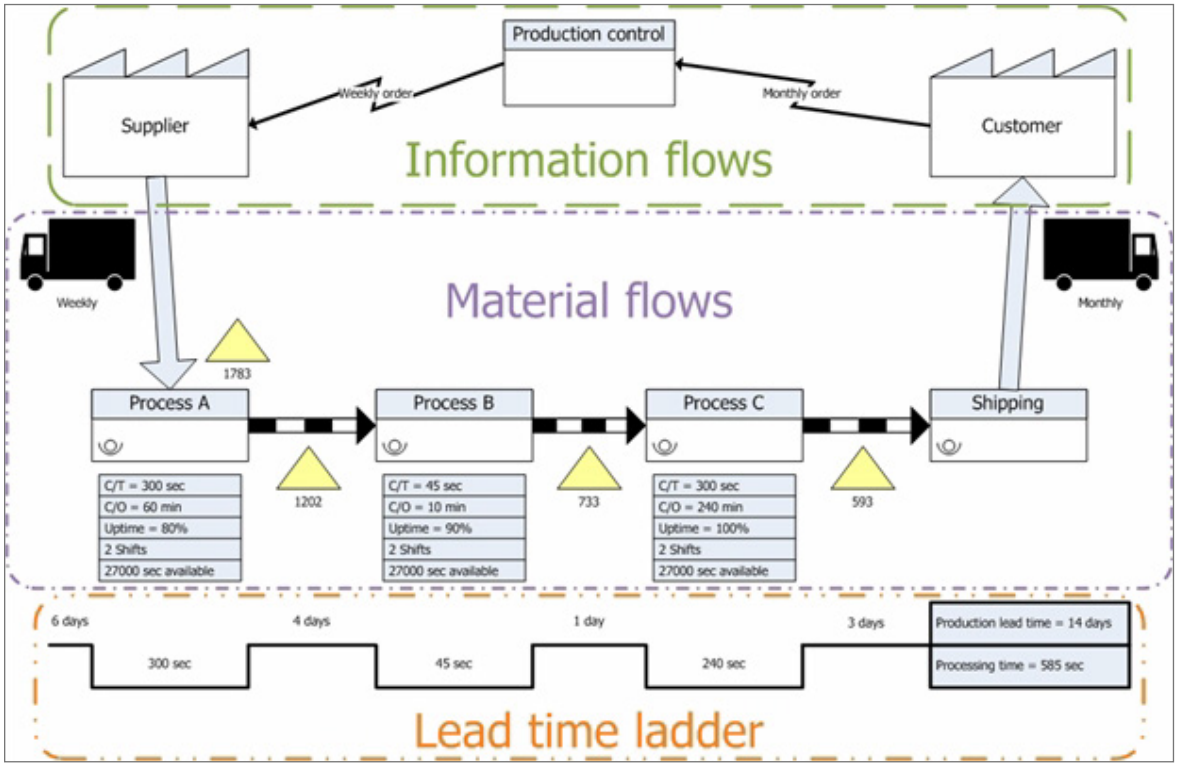

Figure 12-4 [Attribution: Daniel Penfield, CC https://creativecommons.org/licenses/by-sa/3.0/deed.en]

The sequential/pipeline nature of ETL lends itself very well to Value Stream analysis, highlighting delays and finding bottlenecks that can be related easily to business cost, in terms of rework, time-to-completion, or risk (such as missing a data processing deadline). For instance, if a team regularly waits for an ingestion task to complete, and that task blocks them from doing other work, then the task may be a candidate for parallelization. On the other hand, if an ETL team regularly has to wait 2 or 3 days for a new test server from the IT team, then that wait time can be identified as "waste" and instead of automating a task, we would look at shortening the wait time for the test server.

Using a Value Stream Map fits well with the use of a deployment pipeline for ETL.

Summary

One of the advantages of putting a process such as ETL into the context of the lifecycle of the database is that the obvious is less likely to be missed. If our governance process is working well, we can design effective systems based on our knowledge of the appropriate source of data, its constraints, limitations, volume, rate of change, ownership, and security requirements. We know the service requirements and the importance of resilience. With the help of operations, we can deliver ETL systems that are easily monitored, that alert appropriately when things go wrong, and are sufficiently economical with network, memory, and CPU that they can expand to serve increasing workloads.

ETL systems can never be allowed to expand organically without stricture, like a coral reef. The result is almost inevitable chaos, usually at out-of-office hours when batch processes are generally scheduled. ETL is an aspect of the database where it is very easy to make a case for a collaborative approach based on delivery, governance and operations, to produce a system that is effective throughout its operational life.

Resources

How to get ETL horribly wrong by Feodor Georgiev: http://preview.tinyurl.com/mra4q7f.

The ETL from Hell by Nigel Rivett: http://preview.tinyurl.com/myru686.

SSIS Design Pattern – ETL Instrumentation by Andy Leonard: http://preview.tinyurl.com/lhzovo8.

DTLoggedExec, a tool for instrumenting SSIS by Davide Mauri: http://dtloggedexec.codeplex.com/ and http://preview.tinyurl.com/y7o2nkqp.