Preocupados por la calidad

Inconcebiblemente hay muchos estudios, mucha tecnología para realizar los procesos, incluso propuestas distintas sobre la forma de implementar los procesos ETL, particularmente en términos de funcionalidad y rendimiento, incluso ajustando los procesos dentro de lo que se llamó "BPM) Business Process Management, sin embargo, hay muy pocas propuestas para garantizar la calidad de la información entregada como resultado de estos procesos. Curiosamente la pérdida de credibilidad de los datos entregados es uno de mayores motivos de fracaso de los datawarehouses.

Incluso cuando encontramos propuestas empresariales para la creación y construcción de test, estos se circunscriben al ámbito del tiempo de desarrollo, por ejemplo en http://www.pqatesting.com/our_ideas/blog/etl_and_data_warehouse_testing/ -2012-, en el que simplemente se habla de ideas podemos encontrar como conclusión que las organizaciones reconozcan que la fase de test es parte del ciclo de vida de los datawarehouses. Una evidencia, que, a juzgar por la literatura, a juzgar por lo que escribimos en los blogs y leemos en las investigaciones científicas está lejos de ser cumplida.

Incluso Theodorou, Abelló y Lehner[4], hablan de información cuantitativa que permitiría ver la calidad de los datos a partir de métricas de la calidad del proceso de software. La aproximación que vamos a proponer en una parte de la gestión de la calidad de la información va a ser más orientada a la calidad del dato desde el punto de vista de negocio. En cualquier caso, no cabe duda, que la aproximación de los 3 autores mencionados permitiría la creación de un cuadro de mando de calidad de la información que podría ser muy eficaz en la medición de este tipo de información.

Objetivos de madurez de ETL

En la industria de la informática, la innovación y el estudio de nuevas formas de solucionar problemas es una constante, pero a veces, hay que mirar a otras ramas de la industria e intentar resolver los problemas de la misma forma que se resuelven en otras ramas. Por ejemplo, las herramientas ETL podrían recopilar información de distribución de los datos, o aprovecharse de las que ya recogen los sistemas SQL y así responder de forma adaptativa. A la hora de gestionar su ciclo de vida, deberían estar más integradas en las estructuras de desarrollo de código fuente, ser más "*DevOps*", lo que implicaría que las piezas de código ETL deberían poder probarse, compilarse, subirse a entornos de test, ejecutar esos test de prueba, y en caso de éxito pasarse automáticamente a un entorno de producción, aprovechándose de las lecciones aprendidas en integración continua en el ámbito del desarrollo de software.

Piezas de una solución ETL eficaz

Tradicionalmente se han diseñado los procesos ETL como tres fases más o menos diferenciadas, la fase de extracción de datos, la fase de transformación y acomodación de esos datos y la fase de

carga del DataWarehouses. Algunos autores añaden una letra más (C) referida a la limpieza de datos. Es cierto que estas son las piezas fundamentales, pero para este estudio además han de cumplir las siguientes características

- **Automatizadas**. Muchos de los pasos que se hacen cuando se implementa un DataWarehouse son repetitivos con el único cambio del objeto que se sincroniza. Sin demasiada dificultad podemos asociar el proceso a patrones de software y aplicar esos patrones para la extracción y carga, y a veces para la transformación de la información.
- **Autodocumentadas**. La tediosa tarea de documentar el resultado de cualquier pieza de software se une al elevado número de objetos que generalmente se ven afectados en un proceso de ETL, para que la documentación que generamos tenga sentido, sea útil y no quede permanentemente desactualizado en un mundo cambiante, ha de ser automática.
- **Confiable**. Todos los sistemas tienen que funcionar con suficientes garantías, pero además han de estar hechos de forma que un cambio de los actores del equipo no suponga una situación calamitosa. En buena medida los patrones ayudan a evitar esta situación.
- **Comprobable**. Y no hablamos únicamente de conteo de registros entre orígenes y destinos, sino de que la información cuadre y tenga sentido desde el punto de vista de negocio. Un sistema de DataWarehouse puede decir los datos exactamente igual que en el origen, pero si las operaciones de limpieza no han hecho su efecto y los datos que tenemos están sesgados o manipulados, los destinatarios de nuestros informes no podrán confiar en los datos que ofrecemos.

Estas propiedades no han de estar únicamente disponibles durante el proceso de creación del software ETL sino que han de ser un componente troncal de nuestro producto generador de DataWarehouses en todo el ciclo de vida, particularmente cuando nuestro sistema se encuentra en un ambiente productivo y las decisiones de las empresas dependen, en buena medida, de la calidad de la información que estos sistemas proporcionan.

Test Framework

El objeto fundamental de este estudio es la creación de un framework de ejecución de test que responda a las necesidades de la construcción y mantenimiento de un DataWarehouse. Existen herramientas de pruebas que se han hecho estándar de facto en el mercado, jUnit para el mundo java, nUnit para el mundo Microsoft y especializaciones para BI como NBI. Todos estos frameworks sirven para que se puedan escribir un conjunto de pruebas que se ejecuten como parte del desarrollo de las piezas de software y contribuyen a la calidad del dato entregado al usuario final.

Orientación a la calidad: Sistema de test. Herramientas existentes

Herramientas de mercado como jUnit o nUnit en procesos ETL

Este tipo de herramientas están totalmente pensadas para el mundo del desarrollo ya sea en .NET o en el mundo java, y se basan invocar piezas de código y comparar resultados, fundamentalmente

para comprobar lo que se llaman pruebas unitarias. Se suelen escribir en C#, Java o cualquier otro lenguaje y resultan de demasiado bajo nivel para hacer pruebas sobre sistemas de DataWarehouse. De ahí la necesidad de utilizar alguna especialización, manteniendo el motor de pruebas que permita a los que construimos procesos ETL probar que el resultado de la ejecución de nuestros procesos no solo ha sido correcta desde el punto de vista de la ejecución sin errores, sino que el resultado, que los datos movidos, resumidos, limpios, o cualquier operación que haya sido necesaria realizar sobre ellos no ha desvirtualizado la información que contienen.

Los desarrolladores ETL no siempre están familiarizados con lenguajes de desarrollo como Java, o C#, de hecho, cuanta más experiencia tienen como desarrolladores ETL más lejos están de lenguajes de desarrollo tradicionales. Por eso necesitamos una aproximación que simplifique la creación de estos test, para no correr el riesgo de que simplemente no se hagan dejando la calidad a la suerte de que no falle ningún proceso, conscientes, de que la buena fortuna no ha de ser una cualidad de los ingenieros en Informatica.

Especialización NBI.io

Cédric L. Charlier, implementó como código abierto una pieza de software que se pone encima de nUnit y que permite escribir pruebas directamente como XML sin necesidad de escribir código en ningún lenguaje de programación. Un ejemplo de cómo es el código que propone, es el siguiente

```
<test name="Compare two query results">
  <system-under-test>
    <execution>
      <query>
        select * from FirstTable;
      </query>
    </execution>
  </system-under-test>
  <assert>
    <equalTo>
      <query>
        select * from SecondTable;
      </query>
    </equalTo>
  </assert>
</test>
```

Este tipo de pruebas abre un universo completo para los que desarrollamos herramientas ETL, permite entre otras cosas:

- configurar cadenas de conexión para distintos orígenes y destinos de datos
- Introducir tolerancias y redondeos
- Intervalos de valores
- Contar elementos de un Resultset
- Comprobar filas únicas
- Transformar columnas

- Comprobar rendimiento en tiempo
- Comparar con ficheros de texto
- Otras

Todas estas funcionalidades son clave a la hora de probar que nuestros procesos de Extracción Transformación y Carga funcionan según los requisitos, y suponen por si mismos un gran avance hacia la calidad del desarrollo de proyectos ETL. Sin embargo, en nuestro framework sentimos la necesidad de ir un paso más allá debido a que echamos de menos alguna funcionalidad más orientada no solamente a las fases de desarrollo sino a las pruebas cuando el DataWarehouse ya está en explotación.

Carencias de los sistemas de test actuales para procesos de Inteligencia de negocios

Los test unitarios o de integración que son los que usa la industria del software no son totalmente válidos en Inteligencia de negocios, incluso con especializaciones con NBI que adaptan mucho el camino para que sean válidos hay ciertas funcionalidades que no implementan y que echamos de menos.

Una de esas funcionalidades es la de la gestión de test dependientes.

Test Dependientes

En un DataWarehouse en donde los datos no están expuestos 3FN sino en formato de explotación para los sistemas de análisis, con sus redundancias, sus claves subrogadas y su definición de dimensiones y jerarquías, se hace necesario que se pueda profundizar en los niveles de un test en caso de que un test más genérico haya encontrado diferencias.

Imaginemos que tenemos un escenario de ventas, en el que leemos de un solo origen, el ERP de la empresa y llevamos nuestros datos a un DWH donde simplemente les damos forma multidimensional a los datos para su explotación eficaz. Si durante el proceso de movimiento de información perdemos parte de ella, podríamos comprobarlo comprobando la suma de importes por año. Si en un año encontramos una diferencia de importe, un test dependiente, debería bucear en los meses hasta encontrar el, o los meses en los que se producen las diferencias. De la misma forma determinados los meses que provocan el error, se podría bucear hasta los días, y encontrado el día -o los días- en concreto que producen las diferencias podríamos buscar todos los registros del objeto que producen esas diferencias. Conocer donde se producen esas diferencias, es el primer paso para arreglarlo.

Accionamiento de palancas en los test

En caso de errores, podríamos proporcionar la llamada a “palancas”, por ejemplo, en forma de procedimiento almacenado o de cualquier secuencia de comandos en un lenguaje, que pueda ser ejecutada en caso de error. Esa secuencia de comandos puede intentar arreglar o preparar la información que contiene errores e intentar solventar el problema por medios propios, o simplemente documentar el lugar en donde esos errores se encuentran e informar al responsable del dato, sobre la confianza o ausencia de la misma que se tiene en la información.

Json para la definición de pruebas

Las estructuras Json, presentes desde el nacimiento del llamado NoSQL han sido las elegidas para la creación de esta estructura de pruebas. Comparado con XML según el estudio referenciado en la bibliografía como [11] de Nurseitov, Paulson, Reynolds ,Izurieta, que concluye que JSON es más rápido y consume menos recursos que XML. Por eso y por estar en un mundo en el que hay que estar con los avances tecnológicos el formato JSON es el que hemos usado para la definición de nuestros escenarios de test.

El formato seleccionado obedece al siguiente patrón

```
{
   "thread":{
                        "ManagedThreadId":0,
                        "ExecutionContext":null,
                        "Priority":0,
                        "IsAlive":false,
                        "IsThreadPoolThread":false,
                        "IsBackground":false,
                        "ThreadState":0,
                        "ApartmentState":0,
                        "CurrentUICulture":"en-US",
                        "CurrentCulture":"es-ES",
                        "Name":null
                      },
   "ActualDifferences":0,
   "AbaFrameworkLogDb":null,
   "OnErrorStoreProcedureCall":{
               "ConnectionString":null,
               "StoreProcedure":null},
   "ActualSourceConnection":null,
   "ActualDestinationConnection":null,
   "CurrentErrorStatus":0,
   "executed":false,
   "ExpendedTimeseconds":0.0,
   "Errors":[  ],
   "TestName":"Test Name",
```

```
   "Category":null,
   "KeyColumnsNumber":1,
   "OnError":4,
   "OnSuccess":0,
   "Status":0,
   "ImplementationStates":1,
   "Source":"Connection String",
   "SourceRows":0,
   "SourceVariables":[   ],
   "DestinationRows":0,
   "DestinationVariables":[    ],
   "CascadeParametersOnError":true,
   "CommandTimeOut":0,
   "Destination":"Connection String"
   "SourceScript":"Command to execute on source",
   "DestinationScript":"Command to execute on destination",
   "TestType":null,
   "ToleranceMatrix":"0,0.01",
   "PerformanceTarget":0,
   "ExternalReference":null,
   "CompletePath":null,
   "MaxDifferences":0,
   "CascadeVariables":[    ],
   "DependentTests":[  { TestObject }         ]

}
```

El objeto Test

Thread

De la definición anterior el objeto **Thread** es meramente una herramienta para permitir la ejecución en paralelo de test dependientes. Todo lo que contiene va exclusivamente relacionado con esa concurrencia. Internamente para comunicar los hilos del proceso he utilizado un objeto `BlockingCollection` que permite que hilos se envíen mensajes unos a otros gestionando la seguridad de lectura y escritura en memoria y estableciendo los pertinentes latches y bloqueos.

Que el motor de ejecución de test tenga una arquitectura paralela es importante para la velocidad a la que pueden ejecutarse las pruebas que se diseñen contra sistemas que pueden residir en servidores distintos.

Actual Diferences

Es un contador de elementos distintos que tras ejecutar el test se rellenará con el número de filas distintas entre origen y destino.

AbaFrameworkLogDB

Sirve para especificar la cadena de conexión si estamos en un entorno que ejecuta las pruebas contra un sistema desarrollado con SolidQ ABA framework. De ser así, el log de ejecución del proceso de test puede guardarse directamente en la base de datos de log y estar disponible para su consulta y documentación de forma automática.

OnErrorStoreProcedureCall

El objeto sirve para especificar una cadena de conexión y un procedimiento almacenado. El procedimiento almacenado en cuestión recibirá un parámetro que será a su vez una cadena que contiene un JSON como el siguiente


```
   "Rows":[
      {
         "Source":{
            "key":{
               "pkSalesOrderID":"43667"
            },
            "values":{
               "RevisionNumber":"8","OnlineOrderFlag":"False",
       "CustomerID":"29974","OrderDate":"31/05/2011 0:00:00","DueDate"
:"12/06/2011 0:00:00","SalesOrderNumber":"07/06/2011 0:00:00","Purchas
eOrderNumber":"PO15428132599","AccountNumber":"10-4020-000646"}
         },
         ",Destination":{
            "key":{
               "pkSalesOrderID":"43668"
            },
            "values":{
               "RevisionNumber":"8","OnlineOrderFlag":"False","Custome
rID":"29614","OrderDate":"31/05/2011 0:00:00","DueDate":"12/06/2011 0:
00:00","SalesOrderNumber":"07/06/2011 0:00:00","PurchaseOrderNumber":"
PO14732180295","AccountNumber":"10-4020-000514"            }
         }
```

```
        }
    ]
}
```

Este registro nos informa que ha encontrad una diferencia al comparar los registro en orden y que valores son los que hay que tratar. Tratando este JSON se podrían lanzar acciones correctoras sobre la base de datos de origen para que los registros bien formado esta vez fluyeran hasta, primero el área de Staging (Sea o no volátil) y el DataWarehouse en el paso final. Se use con ese propósito o meramente como información para levantar una alerta esta aplicación puede ser realmente útil en un entorno productivo.

Current Error Status

Success =0, Warning =1, Error=2. Identifica el estado de ejecución de un test. Aunque en principio los test solo pueden funcionar o fallar, el estado Warning se añade para reflejar los casos en los que, aun existiendo alguna diferencia, los niveles de tolerancia o el número de diferencias permitido hacen que la ejecución del test no se considere error.

Executed

Un valor lógico que consultar y que indica si la prueba ha sido ejecutada o no.

ExpendedTimeseconds,PerformanceTarget

Contiene el tiempo necesario para ejecutar el test en segundos. Es un valor en coma flotante que puede compararse con el valor de objetivo de rendimiento para determinar un error por motivos de rendimiento. Es decir, por no cumplir con las expectativas de tiempo necesario para su ejecución.

Errors

Contiene la lista de Errores que se han producido durante la ejecución de las pruebas

TestName

Identificador del test, sirve para diferenciarlo de otros, se guarda en los logs y puede resultar útil cuando se quiere descubrir donde se han podido producir problemas de ejecución.

Category/ TestType

Como TestName sirver para caracterizar el test, por ejemplo podrían crearse categorías “pruebas Unitarias”, “rendimiento”, “Test de negocio” etc. Simplemente permitirá distinguir en el log el tipo de pruebas que se hacen, su éxito o fracaso etc. Es en definitiva una forma de agrupar test.

KeyColumnsNumber

Por defecto vale 1, expresa el número de columnas (ordenadamente) que representan la clave primaria de un objeto, es decir le sirve al motor de ejecución para saber que objetos han de ser únicos.

De momento no se comprueba esa unicidad, simplemente se utiliza para la comparación de objetos.

OnError

Especifica las acciones que hay que tomar en caso de que se produzca un error en la ejecución de las pruebas

Ignore = 0,

Fail = 1,

RunDependentTestAndIgnore = 2,

RunDependentTestAndFail = 3,

RunDependentTestAndReturnResult = 4

OnSuccess

Indica las acciones que ha de ejecutar el motor cuando un test funcione. Los valores posibles son

ExitWithSuccess = 0,

RunDependentTests = 1

Status

Indica el estado del test. Sirve para habilitar o deshabilitar la ejecución de algunos test sin tener que modificar el programa que los genera, los valores que permite son

Inactive = 0,

Active = 1

ImplementationStates

Indica el estado de implementación del test. Puede que un test esté sin implementar pero que tenga test dependientes implementados que queramos ejecutar. En la generación automática de estos test en ocasiones podemos generarlos al 100% en otras solo alguna parte. En el caso en que se genere solo una parte del test se marca el estado de implementación como no implementado y así puede estar como parte de los test sin ejecutarse hasta que un desarrollador lo termine de implementar. Por ejemplo, podemos en el cubo generado escribir automáticamente el query que mira las ventas por año, pero no podemos relacionar esas ventas, de forma automática con el origen, dado que en la fase de Transformacion no tenemos control alguno sobre los cambios que se han podido realizar. Los valores que se permiten son:

NotImplemented = 0,

Implemented = 1

Source/Destination

Sirven para expresar las cadenas de conexión necesarias para ejecutar comandos en el origen y en el destino.

SourceRows/DestinationRows

Tras la ejecución del test tienen un contador de las filas que han testeado en origen y en destino.

SourceVariables/DestinationVariables

Permiten que pasemos un conjunto de valores, de variables a los comandos que se vayan a ejecutar en el origen o en el destino.

SourceScript/DestinationScript

Contienen el query que específicamente se va a ejecutar en origen y destino y que va a servir para la comprobación de los registros y va a determinar si son o no iguales.

CommandTimeOut

Especifica el tiempo que daremos para que se complete la ejecución de un comando antes de dar un error por timeout, por defecto vale 0, lo que indica que se ejecutará esperando el tiempo que sea necesario.

ToleranceMatrix

Es un string en el que separadas por comas podemos establecer para cada uno de los campos que devuelvan nuestros queries la tolerancia (la máxima diferencia) que permitimos en sus valores.

ExternalReference

Los test podrán estar relacionados con procesos de negocio o con procesos técnicos que eventualmente pueden tener una clave o un código que permita identificarlos. Este campo sirve para recibir ese código y que sea informado en el log de ejecución. De esta forma se puede saber por ejemplo si un proceso de negocio que se compone de un número conocido de test ha pasado con éxito, warning o error en un momento del tiempo, dado que en el log se guardará esta clave externa.

MaxDifferences

Permite especificar el número máximo de registros distintos a partir del cual el test se dará como erróneo. En situaciones en que la calidad de los datos sea super importante puede ser 0, en casos en que un puñado de registros no supongan un problema grave, puede establecerse un valor que permita que si el número de diferencias es menor, pero distinto de cero el test acabe como Warning y no como error.

CompletePath

Este atributo ya pertenece al conjunto de atributos que nacen por permitir test que dependen de otros test (implementado como un objeto contiene un array de objetos del mismo tipo). Cuando se ejecuta un test dependiente, se añade a esta variable el path de su “padre” de esta forma podemos saber en que nivel de profundidad se ha ejecutado un test en concreto.

CascadeVariables

Además de las variables que se pueden pasar a objetos de origen/destino, los test dependientes van a recibir siempre los valores de los registros "padre" que han sufrido un error de esta forma pueden utilizar esos valores para filtrar los datos y circunscribirse al ámbito que marque su padre.

Para que quede más claro, supongamos que hacemos un test que resume las ventas por año. El Source Script podría tener un aspecto tal que "*SELECT YEAR(FECHA) as Anio,SUM(VENTAS) as Importeventas FROM ORIGENVENTAS GROUP BY YEAR(FECHA)",* el test hijo tendría que hacer una agrupación por año y mes, pero debería filtrar solamente por el año que ha fallado. Para conseguir eso en Casade variables los valores de Anio que hayan fallado se guardaran en esta colección como {"anio":"valor"} y se sustituirán en la consulta por meses donde ponga $anio$ por el valor del año que falló y donde se escriba $valor$ por el valor de ese año. Así pues el hijo podría escribirse como "*SELECT YEAR(FECHA) as Anio, MONTH(FECHA) as Mes, SUM/VENTAS) as Totalventas FROM ORIGENVENTAS WHERE YEAR(FECHA)=$anio$ GROUP BY YEAR(FECHA),MONTH(FECHA)*" y si fallara el año 2018 la consulta se quedaría como YEAR(FECHA) = 2018. De esta forma los test dependientes nos van a permitir profundizar a través de las jerarquías que existan en nuestros sistemas hasta llegar al máximo nivel de granularidad.

CascadeParametersOnError

Esta variable booleana marca si la colección CascadeVariables va a ser rellanada con los datos del padre o no, puede valer "true" o "false"

DependenTest

DependentTest contiene un array de test que a su vez tienen todas las propiedades que hemos descrito, incluso, otros DependentTests. De esta forma garantizamos que podemos tener el número de niveles que sea necesario, pero que será, en cualquier caso, un número finito en cuanto instanciemos los test de un problema concreto.

Multiples Tests en un solo fichero

Las pruebas se pueden escribir manualmente rellenando un documento de texto con la estructura JSON descrita en los puntos anteriores. Si fuésemos a escribir test individuales la aproximación descrita sería suficiente. Sin embargo, en muchas ocasiones se realiza un conjunto amplio de test sobre el mismo origen/destino por lo que reaprovechar cadenas de conexión, que son generalmente largas y tediosas de teclear, parece una idea razonable.

Para la implementación el objeto conexión tiene la siguiente estructura JSON

```
{"connections":     [  {   "ConnectionName":"SOURCE1",
                           "ConnectionString":"....."
                    }   ],
"Test":[ { ...
          "Source":"$SOURCE1$",
                 ..} ]
}
```

De esta forma especificando solamente una vez la conexión se reaprovecha en cuantos test sea necesario.

Se use esta funcionalidad o no, la estructura es válida para generar un conjunto de test y lanzarlos al motor de ejecución para que se encargue de realizar la operación de ejecución.