CONTAINER-MANAGED ETL APPLICATIONS FOR INTEGRATING DATA IN NEAR REAL-TIME

Josef Schiefer
IBM Watson Research Center
Hawthorne, NY USA
josef.schiefer@us.ibm.com

Robert M. Bruckner
Vienna University of Technology
Vienna, Austria
bruckner@ifs.tuwien.ac.at

Abstract

As the analytical capabilities and applications of e-business systems expand, providing real-time access to critical business performance indicators to improve the speed and effectiveness of business operations has become crucial. The monitoring of business activities requires focused, yet incremental enterprise application integration (EAI) efforts and balancing information requirements in real-time with historical perspectives. The decision-making process in traditional data warehouse environments is often delayed because data cannot be propagated from the source system to the data warehouse in a timely manner.

In this paper, we present an architecture for a container-based ETL (extraction, transformation, loading) environment, which supports a continual near real-time data integration with the aim of decreasing the time it takes to make business decisions and to attain minimized latency between the cause and effect of a business decision. Instead of using vendor proprietary ETL solutions, we use an ETL container for managing ETLets (pronounced "et-lets") for the ETL processing tasks. The architecture takes full advantage of existing J2EE (Java 2 Platform, Enterprise Edition) technology and enables the implementation of a distributed, scalable, near real-time ETL environment. We have fully implemented the proposed architecture. Furthermore, we compare the ETL container to alternative continuous data integration approaches.

Keywords: Real-time data integration, middleware architectures, data warehousing

Introduction

The widespread use of the Internet and related technologies in various business domains has accelerated the intensity of competition, increased the volume of data and information available, and shortened decision-making cycles considerably. Consequently, strategic managers are being exposed daily to huge inflows of data and information from the businesses they manage and they are under pressure to make sound decisions promptly. Typically, in a large organization, many distributed, heterogeneous data sources, applications, and processes have to be integrated to ensure delivery of the best information to the decision makers. In order to support effective analysis and mining of such diverse, distributed information, a data warehouse (DWH) collects data from multiple, heterogeneous (operational) source systems and stores integrated information in a central repository.

Since market conditions can change rapidly, up-to-date information should be made available to decision makers with as little delay as possible. For a long time it was assumed that data in the DWH can lag at least a day if not a week or a month behind the actual operational data. That was based on the assumption that business decisions did not require up-to-date information but very rich historical data. Existing ETL (extraction, transformation, loading) tools often rely on this assumption and achieve high efficiency in loading large amounts of data periodically into the DWH system. Traditionally, there is no real-time connection between a DWH and its data sources, because the write-once read-many decision support characteristics conflict with the continuous update workload of operational systems and result in poor response time. Therefore, *batch* data loading is done during frequent update windows (e.g., every night). With this approach the analysis capabilities of DWHs are not affected. ETL approaches often take for granted that they are operating during a batch window and that they do not affect or disrupt active user sessions.

While this still holds true for a wide range of DWH applications, the new desire for monitoring information about business processes in near real-time is breaking the long-standing rule that data in a DWH is static except during the downtime for data loading. *Continuous* data integration aims to decrease the time it takes to integrate certain (time-critical) data and to make the information available to knowledge workers or software agents. This enables them to find out what is currently happening, decide on what should be done (utilizing the rich history of the DWH), and react faster to typical and abnormal data conditions (tactical decision support). Continuous and near real-time data integration for DWHs minimizes propagation delays from operational source systems of an organization, which are responsible for capturing real world events. This improves timeliness by minimizing the average latency from when a fact is first captured in an electronic format somewhere within an organization until it is available for knowledge workers who need it.

Over the past couple of years, the Java 2 Platform Enterprise Edition (J2EE) has established itself as major technology for developing e-business solutions. This success is largely due to the fact that J2EE is not a proprietary product, but rather an industry standard, developed as result of an industry initiative by large IT companies. Many ETL solution vendors make their platforms J2EE compliant by extending their products with Java interfaces and J2EE connectors (Sun Microsystems 2002). Although these solutions can integrate J2EE components into the ETL processing, they cannot take full advantage of a *J2EE application server* with its middleware services (e.g., resource pooling, caching, clustering, failover, load-balancing, etc.).

In this paper, we extend an existing J2EE application server with an *ETL container* that enables a seamless integration with other J2EE containers and the utilization of all available middleware services of the application server. Similar to J2EE Web applications, where servlets and JSPs take the place of traditional CGI scripts, our approach uses managed ETL components (so-called *ETLets*) that replace traditional ETL scripts. By extending a J2EE application server with an ETL container, organizations are able to develop *platform and vendor independent ETL applications* the same way as traditional J2EE applications. Developers are able to quickly and easily build scalable, reliable ETL applications and can utilize Java middleware services and reusable Java components provided by the industry.

The paper is organized as follows. In the next section, we discuss related work. The third section provides a discussion of continuous data integration systems. In the fourth section, we describe the architecture and implementation of the ETL container. The fifth section presents and evaluates results of using the ETL container approach. Finally, we conclude and discuss future work.

Related Work

As organizations interact on a real-time basis and business processes cut across multiple departments and business lines, the need for information integration leads to the adoption of *enterprise application integration* (EAI) approaches in which message-oriented middleware forms the backbone of the enterprise integration (Arsanjani 2002). EAI also deals with the guaranteed delivery of information. Data has to get to its destination without loss as fast as possible. However, EAI tools are not primarily designed for taking care of data integration, aggregation, and consolidation, especially at an enterprise level. EAI is important but it has yet to master the cross-stream integration layers that ETL already offers. EAI technology focuses on moving *small bursts* of information between systems using a bus architecture, and not on bulk data movement.

The presence of *continuously changing* data is a fundamental change to the stable data snapshot paradigm of traditional DWHs. Data modeling, report execution, and information delivery may need to cope with this new paradigm. There are two different temporal characterizations of the information appearing in a DWH: one is the classic description of the time when a given fact has occurred; the other represents the instant when the information is actually intelligible to the system (Bruckner and Tjoa 2002). This distinction, implicit and usually not critical in on-line transaction processing applications, is of particular importance for (near) real-time DWHs. There it can be useful (or even vital) to determine and analyze what the situation was in the past, with only the information available at a given point in time. This goes beyond solving the problem of slowly changing dimensions (Kimball 1996). Existing models lack built-in mechanisms for handling change and time (as revealed by a survey of multidimensional data models in Pederson et al. 2001).

Data may take the form of continuous *data streams* (Babcock et al. 2002) rather than finite stored datasets. Fields of application include manufacturing processes, click-streams in Web personalization, and call detail records in telecommunication (Chen et al. 2000). By nature, a stored dataset is appropriate when significant portions of the data are queried again and again, and updates are small and/or relatively infrequent. In contrast, a data stream is appropriate when the data is constantly changing (often exclusively through insertions of new elements), and it is either unnecessary or impractical to operate on large portions of the data multiple times.

Vassiliadis et al. (2001) propose an approach that uses workflow technology for the management of DWH refreshment processes. The authors propose a meta model for DWH processes that is capable of modeling complex activities, their interrelationships, and the relationship of activities with data sources from a logical, physical, and conceptual perspective. Only the physical perspective of the meta model covers the execution details of the DWH processes. The logical and conceptual perspectives allow the modeling of complex DWH processes as workflows.

Most of the previously discussed approaches focus on data refreshment processes with batch-oriented data integration. Although the ETL container proposed in this paper also supports batch-oriented ETL processing, its major addition is the definition and efficient execution of processing flows for individual event types, which is crucial for continuous data integration and difficult to achieve with a workflow management system. Moreover, the ETL container provides efficient evaluation capabilities for fresh calculated business metrics that can also be used to control the ETL process.

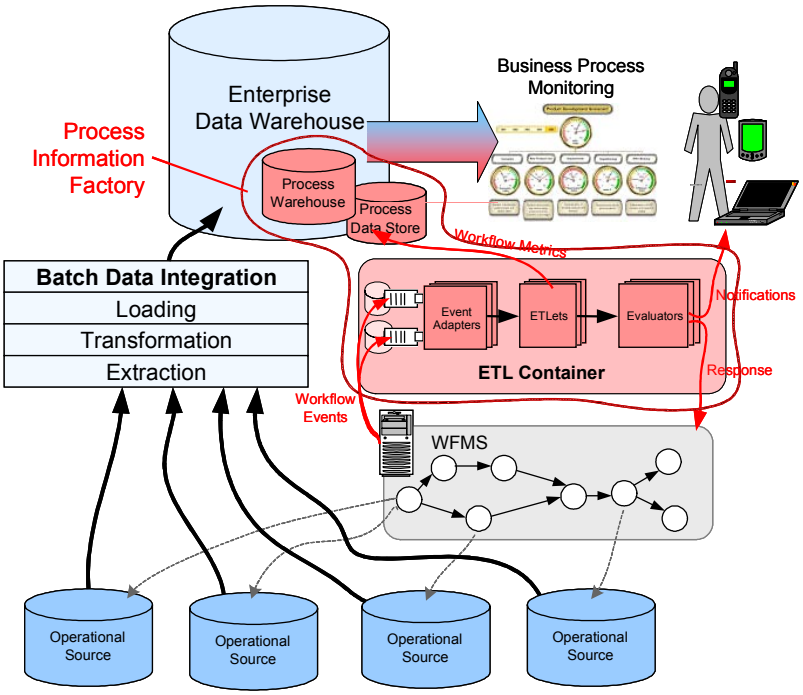

Figure 1. Process Information Factory with ETL Container

We use the ETL container as a solution for integrating workflow events. Figure 1 shows the ETL container as part of a *process information factory* for integrating event data continuously from a workflow management system (List et al. 2000). Arrows indicate a flow of data or control among these components. The highlighted components show the extensions to a conventional (passive) DWH environment. The ETL container ultimately transforms on-the-fly events from workflow management systems (WFMSs) into workflow metrics that are stored in the DWH environment.

ETL Subsystem

A DWH can only be considered (near) real-time, when essential parts of the data are updated or loaded on an intraday basis, without interrupting user access to the system. However, most ETL tools, whether based on off-the-shelf or custom-coded products, operate in a batch mode. They take for granted that they have free reign to drop and reload tables and conduct other major database operations without disturbing simultaneous end-user queries. During the update process of a traditional DWH, analytical queries are not processed. This period of time is often called the *update window*. Due to the constantly increasing size of DWHs and the rapid rates of change, there is increasing pressure to reduce the time taken for updating the DWH. There are three basic approaches to cope with this situation:

- **Minimize the update window.** Bulk loading tools achieve high-performance for data integration and therefore shrink update windows. These tools are able to generate data blocks, which are directly written to disk (i.e., they are added to the data files of the DWH database). However, this approach does not use database transactions for efficiency reasons. Therefore, bulk loading should not overlap with normal DWH workload processing in order to avoid interference and unexpected results. There has been substantial work done to identify optimal strategies for updating single materialized views of DWHs (Labio et al. 1999). These optimizations reduce the update window significantly by minimizing the whole update work (e.g., by avoiding recomputation of materialized views from scratch). Other approaches attempt to compute the exact delta for incremental maintenance (Ram and Do 2000).

- **Data/table swapping.** In order to isolate ETL tools from simultaneous end-user queries, data that changes throughout the day can be loaded into a parallel set of tables, either through a batch process that runs every few minutes or through a continuous data integration process. Once the load interval (e.g., five minutes) is up, the freshly loaded tables are simply swapped into production and the tables with the now stale data are released from production. This can be accomplished through the dynamic updating of views or by simple table renaming. The drawback of this type of *n-minute cycle-loading* process is that the data in the warehouse is not truly real-time. For applications where true real-time data is required, the best approach is to continuously trickle-feed the changing data from the source directly into the DWH.

- **Continuous data integration.** Near real-time data integration manages continuous data flows from transactional systems and performs the incorporation to a DWH. The data integration process is performed in parallel with complex analytical processing in the DWH and therefore differs from traditional batch load data integration. It has to use regular database transactions (i.e., generating *inserts* and *updates* on-the-fly), because, in general, database systems do not support block operations on tables while user queries simultaneously access these tables. In addition, the data rate of continuously integrated data is usually low with an emphasis on minimized latency for the integration. However, bulk loading processes buffer datasets in order to generate larger blocks and, therefore, the average latency for data integration with bulk loading increases. Data integration based on frequent bulk loads can exceed the data latency requirements.

In the process information factory example, we need a continuously running service, which provides always-listening sessions. Obviously, continuous data integration cannot be as fast as bulk load integration. It is necessary to carefully identify critical data, which needs to be integrated constantly with minimized latency. In general, not all but only a relatively small amount of data represents transactions or other relevant information that must be captured continuously and "live" from transactional systems to be integrated in near real-time with the historical information of the DWH.

We need to accept real-time feeds of transactional data, which can be a burden for the systems involved if they work in a synchronous mode using a direct connection. Using an asynchronous architecture, operational systems are able to publish (*push*) their transaction data to message queues without being slowed down or even disturbed if a data integration process fails. Messaging middleware provides a buffered data transport and reliable delivery. The data integration tier is able to decide when to *pull* new data from operational sources.

Message queues provide several characteristics that are well suited for continuous loading. First, like the water in a faucet, the contents of a queue are able to continuously flow, or be held back, ready to flow at any moment. Second, queues can receive data from many distributed platforms and systems. Third, queues are reliable; they can guarantee the data delivery.

Since every event is significant, the loss of a single event can lead to a loss of synchronization in state between a source system and the DWH. Messages containing event (rather than state) information must be queued and removed after reading to ensure that every event is processed exactly once. Most message queuing systems also offer transactional support and persistence for transmitted messages.

ETL Container Architecture and Implementation

In this section we introduce a J2EE (Java 2 Platform, Enterprise Edition) architecture for a container-based ETL environment which enables a continuous integration of data from various source systems in near real-time. J2EE environments have containers that are standardized runtime environments that provide specific services to the components. Components can expect these services to be available on any J2EE platform from any vendor. For example, EJB (Enterprise Java Beans) containers provide automated support for transaction and life cycle management of EJB components, as well as bean lookup and other services.

Containers also provide standardized access to enterprise information systems (e.g., providing access to relational data through the JDBC API—Java Database Connectivity Application Programming Interface).

In our approach, we extend an existing J2EE environment with an ETL container that provides services for the extraction, transformation, and loading of the data into a DWH. The ETL container is a robust, scalable, and high-performance data staging environment, which is able to handle a large number of data extracts or messages from various source systems in near real-time. It takes responsibility for system-level services (such as threading, resource management, transactions, security, persistence, and so on) that are important for the ETL processing. The ETL container is responsible for the monitoring of the data extracts and transformations, and ensures that resources, workload, and time-constraints are optimized. ETL developers are able to specify data propagation parameters (e.g., schedule and time constraints) in a deployment descriptor and the container will try to optimize these settings. This arrangement leaves the ETL developer with the simplified task of developing functionality for the ETL processing tasks. It also allows the implementation details of the system services to be reconfigured without changing the component code, making components useful in a wide range of contexts. Instead of developing ETL scripts, which are often hard to maintain, scale, and reuse, ETL developers are able to implement reusable components for ETL processing. We further extended this concept by adding new container services, which are useful for the development and execution of ETL applications. Examples of such container services are a flow management service, which allows a straight-through processing of complex ETL processes, or an evaluation service, which significantly reduces the effort for evaluating calculated business metrics.

Figure 2 shows the architecture of a J2EE ETL environment with an ETL container, an EJB container, and various resource adapters. An *ETL container* is part of a Java application server and manages the lifecycle of ETL components, and also provides services for the execution and monitoring of ETL tasks. There are three component types that are managed by the ETL container: (1) event adapters, (2) ETLets, and (3) evaluators.

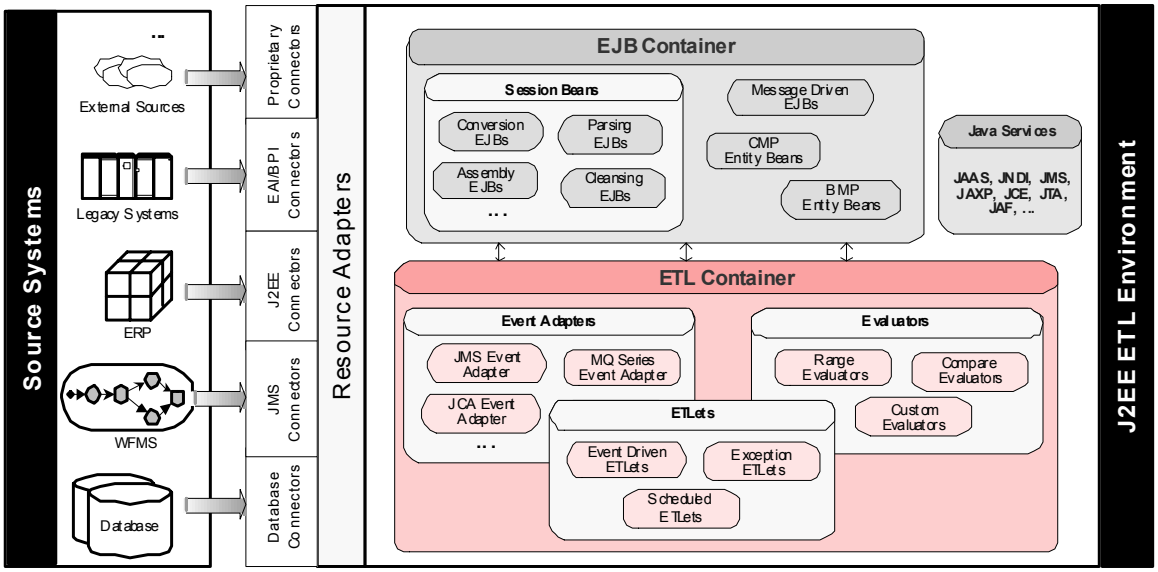

Figure 2. J2EE ETL Environment

Event adapters are used to extract or receive data from the source system, and they unify the extracted data in a standard XML format. *ETLets* use the extracted XML data as input and perform the ETL processing tasks. ETLets also publish business metrics that can be evaluated by *evaluator* components. Each of these components must implement a certain interface that is used by the ETL container in order to manage the components. The ETL container automatically instantiates these components and calls the interface methods during the components' lifetime.

EJB containers enhance the scalability of ETL applications and allow the distribution of the ETL processing on multiple machines. EJB containers manage the efficient access to instances of the EJB components regardless of whether the components are used locally or remotely. ETL developers can write EJB components for typical ETL processing tasks (data cleansing, data parsing, complex transformations, assembly of data, etc.) and can reuse these components in multiple ETL applications.

J2EE environments have a multitiered architecture, which provides natural access points for the integration with existing and future source systems (Sun Microsystems 2001). The integration tier (in J2EE environments also called *enterprise information tier*) is crucial for an ETL environment because it contains data and services often implemented by non-J2EE resources that can be utilized for the data extraction from source systems. Workflow management systems, databases, legacy systems, ERP (enterprise resource planning), EAI (enterprise application integration) systems, and other existing or purchased packages reside in the integration tier. A J2EE ETL environment provides a comprehensive set of resource adapters for these source systems. For instance, the J2EE platform includes a set of standard APIs for high-performance connectors. Many vendors of ERP or customer relationship management systems (e.g., SAP, Oracle) offer a J2EE connector interface for their systems. With our architecture, ETL developers can reuse existing high-performance, J2EE connectors and connectors of EAI solutions for the data extraction without worrying about issues like physical data formats of source systems, performance or concurrency. Moreover, the J2EE platform includes standard APIs for accessing databases (JDBC) and for messaging (JMS) which enables the ETL developers to access queue-based source systems that propagate data via messages.

Extracting Data with Event Adapters

The purpose of event adapters is to extract or receive data from source systems and to unify the different data formats. Event adapters translate all raw source data into an XML event format with a defined XML schema. Event adapters can accept event data asynchronously via messaging software or synchronously via a resource adapter (e.g., JDBC or J2EE connector). The first option is more scalable because it completely decouples the event source (e.g., a WFMS) from the event processing in the ETL container. Event adapters are running on their own threads and can receive and dispatch events in parallel.

Figure 3 illustrates event adapters receiving and dispatching events via an event dispatcher that assigns threads for the ETL processing with ETLets and evaluators. The components shown with round boxes are the ETL components that are managed by the container and have to be implemented by the ETL developers. The components shown with square boxes are internal container components that are used to conjoin all ETL components. Please note that the ETL developers never see or have to deal with these internal components. We show these internal components for illustration purposes only.

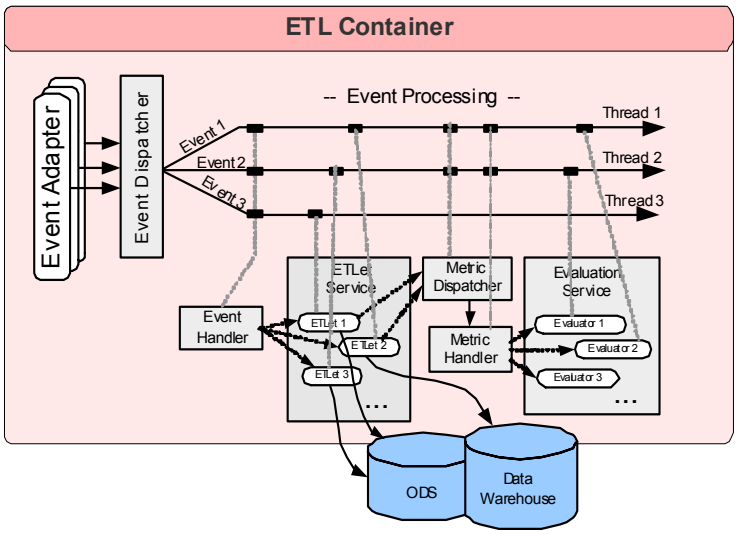

Figure 3. Multi-Threading in the ETL Container
(Adapted from J. Schiefer, J. J. Jeng, and R. M. Bruckner, "Real-Time Workflow Audit Data Integration into Data Warehouse Systems," *Proceedings of the Eleventh European Conference on Information Systems*, C. Ciborra, R. Mercurio, M. DeMarco, M. Martinez, and A. Carignani (eds.), Naples, September 2003.)

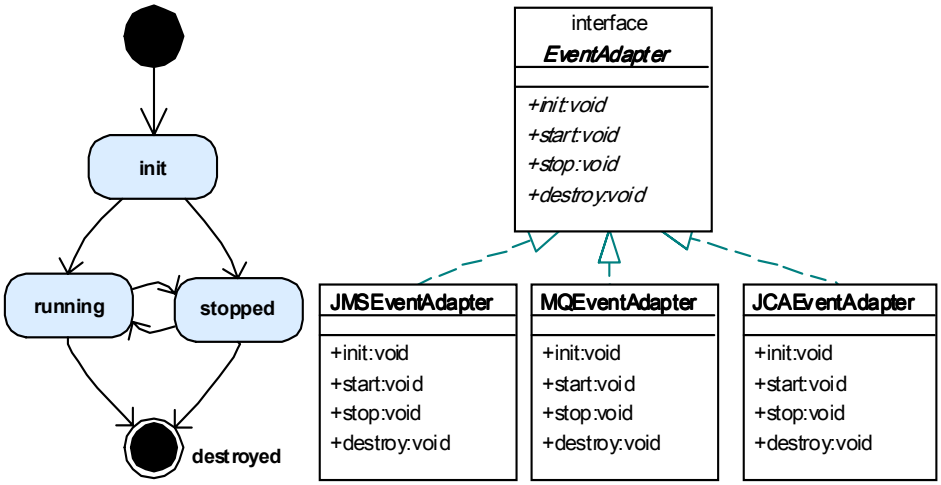

Figure 4. Lifecycle, Interface/Classes (Event Adapters)

In order to address overload situations, where not enough resources are available to instantiate ETLets for the event processing, the ETL container can block an event adapter temporarily. For instance, if there is no thread available for the processing of an incoming event within a specified timeout period, the event adapter is notified of the overload situation and can react to this situation individually. Figure 4 shows the lifecycle and interface of event adapters. All ETL components include an *init*, *running,* and *destroyed* state. Only the event adapters have an additional *stopped* state which enables the ETL container to stop the event processing.

ETL Processing with ETLets

After the dispatching of the standardized XML events in the event adapter, the ETL container invokes ETLets which have subscribed to the event. In order to achieve a high performance level, the ETL container uses a thread pool whose size is adjustable. ETLets run in multiple threads in parallel. However, all processing steps of an ETL processing flow usually run within the same thread. For one *event type* (e.g., ACTIVITY_STARTED events) there can be several ETLets that have subscribed to the same event type. In this case, the ETL container will invoke the subscribed ETLets in parallel. The ETLet components must implement one of the ETLet interfaces shown in Figure 5.

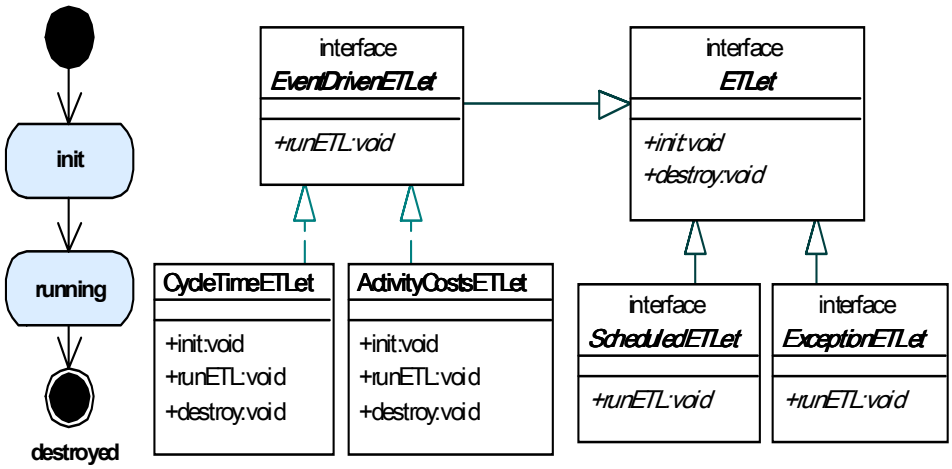

Figure 5. Lifecycle, Interface/Classes (ETLets)

The ETL container manages three types of ETLets: event-driven ETLets, scheduled ETLets, and exception ETLets. All ETLet types have a *runETL()* method with a different signature. ETL developers have to implement this method with the ETL processing logic. This processing logic can include any type of data transformation, the calculation of business metrics, and storing the metrics in a database table. ETLets can also publish the business metrics to allow the container to pass these metrics to the evaluator components that have subscribed to the *metric type*.

Event-driven ETLets can subscribe to a number of XML events that are dispatched by the event adapters and that are relevant to the processing logic in the runETL() method. For the subscription, the ETLet has to define either the event-ID or a matching XPath expression for the XML events of interest. For instance, the *CycleTimeETLet* determines the cycle time of business activities by subscribing to the ACTIVITY_STARTED, ACTIVITY_COMPLETED, and ACTIVITY_CANCELED events in order to be able to calculate the activity cycle times.

Scheduled ETLets are triggered by the ETL container in intervals at specific points in time. The schedule for the triggering is also configurable in the deployment descriptor of the ETLet. Scheduled ETLets can be used to perform recurring ETL tasks, for instance aggregating the daily order data after a business day.

Exception ETLets are a special kind of ETLets that are invoked when an exception is thrown within an ETLet of the ETL application and this exception cannot immediately be handled by the ETLet itself. For instance, this happens if a manual step is required in order to resolve the problem of the exception. Exception ETLets are used to store these exceptions and the triggering events that caused the exception in a file or database for a later manual correction of the problem. They also can be used for sending out notifications. For instance, an administrator can be notified via e-mail that an unhandled exception occurred in an ETL container. Exception ETLets must also be defined in the deployment descriptor.

Evaluation of Business Metrics with Evaluators

The metrics calculated and published by ETLets can be evaluated by evaluator components. An evaluation of fresh calculated business metrics can be very valuable because it can be used to create an intelligent response (e.g., sending out notifications to business people or triggering business operations) in near real-time.

Evaluators have the same lifecycle as ETLets (see Figure 6). They can be either implemented by ETL developers or act as proxies by forwarding the evaluation requests to rule engines for more sophisticated evaluations. In the first case, ETL developers have to implement the *evaluate()* method of the evaluator interface with the evaluation logic.

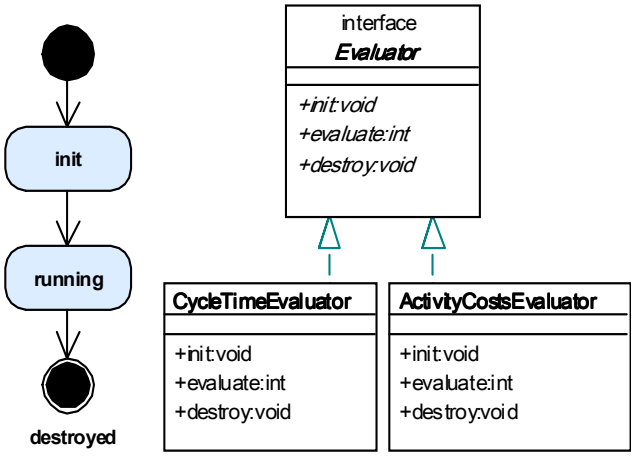

Figure 6. Lifecycle, Interface/Classes (Evaluators)

An evaluator can subscribe to a set of *metric types* that are defined in the deployment descriptor. For the subscription to metric types, there are two mechanisms available: (1) evaluators can subscribe to a set of metric types independent from which ETLets generated the metrics, and (2) they can also subscribe only to the metric types of a defined set of ETLets. The ETL container makes sure that an evaluator receives the correct metrics from the ETLets. Note that an evaluator is not aware of how a metric was calculated or which events triggered the metric calculation. An evaluator is automatically invoked by the container when a new metric of the subscribed metric type became available.

Deployment of an ETL Application

The J2EE platform allows ETL developers to create different parts of their ETL applications as reusable components. The process of assembling components into modules and modules into ETL solutions is called *packaging*. The process of installing and customizing the ETL components in an operational environment is called *deployment*. The J2EE platform provides facilities to make the packaging and deployment process simple. It uses JAR (Java archive) files as the standard package for modules and applications, and XML-based deployment descriptors for customizing components and applications. Although deployment can be performed directly by editing XML text files, specialized tools best handle the process. For the packaging and deployment of an ETL solution, two types of deployment descriptors are used for the modules (see Figure 7):

- **EJB deployment descriptors (ejb-jar.xml).** An EJB deployment descriptor provides both the structural and application assembly information for the EJBs that are used in the ETL application. The EJB deployment descriptor is part of the J2EE platform specification (Sun Microsystems 2002).

- **ETL application deployment descriptor (etl-app.xml).** The deployment descriptor for ETL applications lists all ETL components (event adapters, ETLets, evaluators) and specifies the configuration parameters for these components. This includes general configuration information about the ETL application and ETL components (e.g., name, description), the implementation classes for the ETL components, data propagation parameters for the event adapters (e.g., connection parameters), configuration and ETL processing parameters for ETLets (e.g., triggers, published metrics), and evaluation parameters for evaluators (e.g., evaluation thresholds). During the deployment, the deployment team has to adapt the deployment descriptor settings to an existing DWH environment.

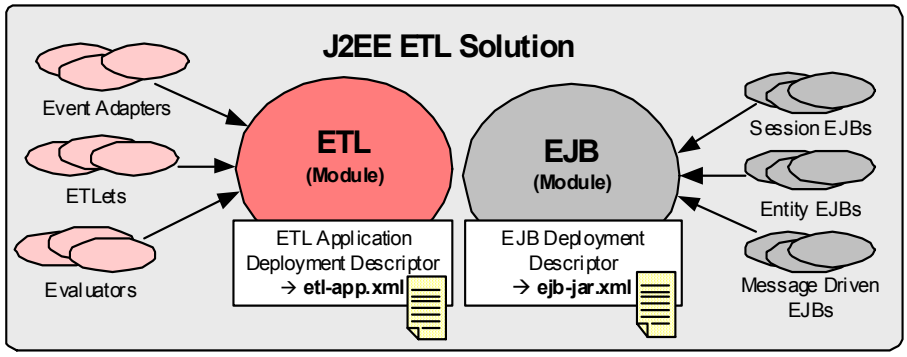

Figure 7. ETL Solution Deployment

Experiences and Comparison

This section gives an overview of our experiments with the ETL container. Furthermore, we built another continuous data integration system using a fundamentally different design approach and compare it to the ETL container approach.

Event Detection for Operational Source Systems

There are several methods to detect changes of operational systems (Todman 2001) such as database log file analysis, snapshot comparison, materialized view maintenance, database triggers, and audit trails. In the following we discuss two methods used in our experiments.

- **Modifying operational systems.** If the underlying database system supporting an operational application is relational, then it is possible to capture the changes by using *database triggers*. Figure 8 shows the trigger definition for an inventory table. In this scenario, our inventory system (running on a Microsoft SQL server) consists of several tables and a central table that stores the inventory levels for all articles. Every change (e.g., an item is sold) is captured by the trigger, which takes the relevant data and calls upon the stored procedure *send_msmq_msg*. This procedure sends the message to the specified *message queuing* server using DTS (data transformation services) objects. A similar solution could be an operational system based on an Oracle database using *Oracle Advanced Queuing* or other messaging middleware.

- **Analysis of audit trail.** Some operational applications maintain audit trails to enable changes to be traced. For example, workflow management systems (WFMSs) provide an audit trail about state changes of workflows, but include only a limited level of detail data, which can be a constraint in practice. In our second scenario, we use an integrated utility called *QTool* in order to read audit trail files and write one message per audit trail dataset into a message queue. We will briefly discuss *QTool* next.

```
CREATE TRIGGER ETL_Inventory ON [dbo].[Inventory]
FOR INSERT, UPDATE AS
DECLARE @ArticleID Varchar(36)
DECLARE @Level Varchar(21)
DECLARE @Msg Varchar(100)
DECLARE myCursor CURSOR READ_ONLY FOR SELECT
CONVERT(CHAR(36),ArticleID) AS ArticleID,
CONVERT(CHAR(21), Level) AS Level FROM inserted
OPEN myCursor
FETCH NEXT FROM myCursor INTO @ArticleID, @Level
WHILE (@@FETCH_STATUS = 0) BEGIN
SELECT @Msg = @ArticleID + ';' + @Level + ';' +
CONVERT(CHAR(23),getdate(),121)+CHAR(13)+CHAR(10)
EXEC send_msmq_msg 'Server\private$\ETLqueue', @Msg
FETCH NEXT FROM myCursor INTO @ArticleID, @Level
```

Figure 8. Trigger for Operational System (Transact-SQL Syntax of SQL Server)

Implementation of the Second Prototype

The ETL container is one solution to the problem of continuous data integration. It is easily adaptable due to its J2EE-based architecture. However, since *off-the-shelf database systems* provide specialized data load utilities, we built a second system based on the Teradata database. It makes use of a Teradata-specific data load utility (*tpump*) in order to achieve continuous data loading.

Our integrated utility *QTool* (Bruckner and Braito 2003) provides the following functionality: (1) queue administration, management, and monitoring of Microsoft Message Queuing (MSMQ), (2) message queue feed functionality (reading data from files), and (3) a job scheduler that controls and coordinates parallel executions of the instances of the database load utility (i.e., *tpump*). The job scheduler supports the automatic hand-off of control between *tpump* instances and handles the end-of-job processing. The load utility *tpump* generates SQL-statements on-the-fly and sends them to the database, which executes them as single transactions using row level locking. *QTool* currently offers support for MSMQ has been designed, but is not limited to using Teradata's *tpump* utility. The *QTool* scheduler acts as a job scheduler and is responsible for starting and stopping *tpump* instances and for managing the post-job processing (e.g., processing error tables). The loading jobs are configured and alternately called to enable continuous data integration from a message queue into a DWH. If one *tpump* instance ends or even if it fails due to an error, a second instance gets immediately access to the message queue and continues the data load without interruption. The scheduler makes sure that there is always one instance running and a second one waiting.

Settings of Experiments

We use two different types of source systems as discussed above. The first source system is an inventory system running on a Pentium IV, 2 GHz, 512 MB RAM, WinXP, and SQL Server 2000. Every change to the central inventory table fires a *database trigger*, which sends a message to the messaging server containing the changed inventory level. The second source system is a WFMS running on the same machine. It generates *audit trail* data, which are read by *QTool* and feed into the message queue.

The messaging server is a Pentium II, 400 MHz, 256 MB RAM, Win2K server, running MSMQ Version 2.0. We have also done experiments with MSMQ 3.0 on a WinXP machine. The load server is an Athlon XP 2100+, 512 MB RAM, WinXP, alternately running *QTool* or the *ETLContainer* (with IBM Websphere application server, Version 5).

We are using typical DWH database systems in our *ETL container* experiments: DB/2 Version 8.1, SQL Server 2000, and Teradata V2R4.1. Since QTool uses the *tpump* data load utility, we had to use a Teradata database for the second prototype running on a dual-CPU SMP system (2*1GHz, 2 GB RAM, RAID). The systems are LAN-connected via Ethernet (100 megabit/sec). If continuous data loading concentrates on data freshness for certain data classes with very low data rates rather than throughput, an ADSL connection (upstream 64 kilobit/sec) between the load server and the DWH database is also sufficient. This might be applicable for highly distributed environments.

Experiences and Comparison

It is fact that continuous data integration (generating SQL-statements on-the-fly) cannot be as fast as bulk loads. Since both implemented systems still offer potential for (technical) performance optimizations, we do not provide detailed evaluations. However, the comparison of the presented ETL container and the second approach shows some interesting results. They can be summarized as follows: If you require configurability, flexibility, and complex event handling during continuous data integration, the ETL container is the best option. If one does not require platform or database independence, but needs better performance for continuous data loads, the second approach (utilizing special-purpose data load utilities from database vendors) is best.

In our experiments, we were able to process up to 70 events per second. Every message contains about 100 bytes, which results in up to 500 MB of continuously integrated data per day with relatively low resource utilization at the underlying database system. Complex data transformations (based on ETLets) and metric evaluations will decrease the performance. Better hardware with multiple processors and better network connection will increase the performance. Since the design of the ETL container is component-oriented and the system makes use of the parallelism provided by an application server, it is very scalable.

Both prototype systems provide robust continuous data loads; the ETL container provides sophisticated exception handling for ETLets, while the *QTool* scheduler controls the execution of loading jobs and makes sure that *tpump* instances are running alternately (an error condition initiates an immediate take-over by the other instance).

The ETL container approach has several **advantages** over traditional ETL solutions:

- The J2EE platform provides interoperability with many source systems. Existing Java resource adapters (e.g., J2EE connectors, JDBC, JMS) and many Java-compliant EAI connectors can be utilized for the data extraction.
- It supports complex transformation and data processing tasks. All tasks are implemented in Java. Therefore, we can take full advantage of the J2EE platform and container services.
- The ETL container supports straight-through processing for ETL tasks. ETLets are executed in parallel and controlled by the flow management service of the container. ETLets can also use EJB components to distribute the ETL processing.
- The usage of Java threads for the ETL processing and lightweight flow management allows the processing of a large number of data extracts or messages concurrently in near real-time.
- The ETL container provides a clean separation of the extraction logic (event adapters), transformation logic (ETLets), and evaluation logic (evaluators), which makes the components pluggable and an ETL application more extendible.
- The evaluation capabilities of the ETL container can be utilized for automatic responses to source systems or for notifications.

- ETL vendors are able to develop prepackaged J2EE ETL solutions that can be extended and customized for organizations. The ETL applications are platform independent and can be deployed in various DWH environments.

However, the ETL container has several **drawbacks** compared to the database-oriented approach (based on a monitoring tool and scheduler, like *Qtool*):

- The flexibility and the capabilities of the ETL container reduce performance. If we integrate data with basic data transformations (converting strings to numbers and checking data conditions), the *QTool/tpump* environment processed 69 events per second, the ETL container only 41. Another experiment (based on the Process Information Factory) does complex transformations of workflow events from a supply chain process. Incoming events are processed by selecting several datasets from a database, calculating complex business metrics with Web services and inserting the data into the DWH. This experiment is possible with the ETL container (which processes five events per second on average), but was not feasible with *tpump*.

- Of-the-shelf data load utilities often provide a limited but easy-to-learn scripting language for programming the data transformation tasks. The development of ETL applications for the ETL container requires Java and JDBC/SQL development skills, because data transformation tasks of ETLets have to be written in Java.

- The advantages of massively parallel database systems (e.g., Teradata) can be better utilized by *database-specific* data load utilities than by external programs, which access the database through standard interfaces (e.g., JDBC).

Summary and Future Work

A traditional DWH focuses on strategic decision support. A well-developed strategy is vital, but its ultimate value to an organization is only as good as its execution. As a result, deployment of DWH solutions for operational and *tactical* decision-making is becoming an increasingly important use of information assets. In this paper we have briefly discussed traditional *ETL (extract-transform-load)* tools, which often provide only partial solutions to the challenge of continuous data integration with low-latency and high data freshness. A DWH can only be considered (near) real-time, when all or essential parts of the data are updated or loaded on an intraday basis, without interrupting user access to the system. However, most ETL tools, whether based on off-the-shelf or custom-coded products, operate in a batch mode; they often take for granted that they are operating during a batch window and that they do not affect or disrupt active user sessions.

We have described a continuous, near real-time data integration approach. The *ETL container* takes responsibility for system-level services (threading, transactions, resource management, etc.). This arrangement leaves the component developer with the simplified task of developing the ETL functionality. It also allows for reconfigurability of implementation details of system services without changing the component code, thus making components useful in a wide range of applications. Instead of developing traditional ETL scripts, which are hard to maintain, scale, and reuse, ETL developers are able to implement Java components for the ETL process. Therefore, we can make full use of the J2EE platform and are not limited to typical scripting languages of traditional ETL solutions.

ETLets are similar to ETL scripts, but they run in a container of a Java application server. ETLets are executed in parallel and managed by an ETL container. With ETLets, a lightweight Java thread, rather than a heavyweight operating system process handles each request. An ETL container controls and monitors the processing by optimizing the settings specified in the deployment descriptor. Furthermore, it provides container services for the evaluation of business metrics and for the management of ETL processing flows. The overall ETL container architecture has a clean separation of *extraction* logic (event adapters), *transformation* logic (ETLets), and *evaluation* logic (evaluators).

At least for the near future, ETL and EAI systems will not necessarily merge functionally. Best-of-breed ETL tools will not be able to compete with best-of-breed EAI systems simply because the technological gaps are still too wide. In the near future, EAI and relational database systems will try to increasingly integrate traditional ETL functionality, whereas ETL systems will retreat (Figure 9). Nonetheless, there will be a convergence of ETL and EAI functionality.

Therefore, we plan to further improve the ETL container based on our experience from the alternative approach of using off-the-shelf database load utilities as well as investigating and analyzing typical EAI systems.

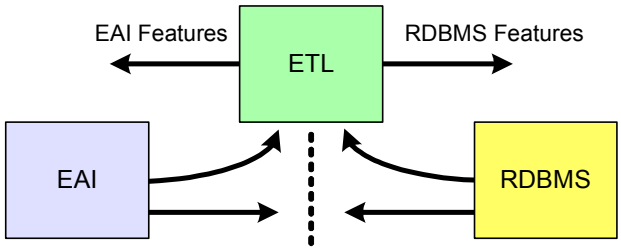

Figure 9. ETL Evolution

Several future projects are in progress on this research. We are building a distributed and clustered environment for ETL containers that allows them to work together in a server farm. We are also developing an evaluation framework that allows rule engines to be seamlessly plugged into the ETL container for more dynamic metric calculation and evaluation. Furthermore, we want to add new container services, which are useful for ETL developers, such as services for caching, concurrency control, and security.

References

Arsanjani, A. "Developing and Integrating Enterprise Components and Services," *Communications of the ACM* (45:10), October 2002, pp. 30-34.

Babcock, B., Babu, S., Datar, M., Motwani, R., and Widom, J. "Models and Issues in Data Stream Systems," in *Proceedings of the Twenty-First Symposium on Principles of Database Systems*, Madison, Wisconsin, May 2002, pp. 1–16.

Bruckner, R. M., and Braito, R. "QTool V2.5 Documentation," Technical Report, Institute of Software Technology, Vienna University of Technology, January 2003.

Bruckner, R. M., and Tjoa, A M. "Capturing Delays and Valid Times in Data Warehouses—Towards Timely Consistent Analyses," *Journal of Intelligent Information Systems* (19:2), September 2002, pp. 169-190.

Chen, Q., Hsu, M., and Dayal, U. "A Data-Warehouse/OLAP Framework for Scalable Telecommunication Tandem Traffic Analysis," in *Proceedings of the Sixteenth International Conference on Data Engineering*, San Diego, CA, March 2000, pp. 201–210.

Kimball, R. *The Data Warehouse Toolkit*, John Wiley and Sons, New York, 1996.

Labio, W. J., Yerneni, R., and Garcia-Molina, H. "Shrinking the Warehouse Update Window," *ACM SIGMOD Record* (28:2), June 1999, pp. 383-394.

List, B., Schiefer, J., Quirchmayr, G., and Tjoa, A. M. "Multi-dimensional Business Process Analysis with the Process Warehouse," in *Knowledge Discovery for Business Information Systems*, W. Abramowicz and J. Zurada (eds.), Kluwer Academic Publishers, Boston, 2000, pp. 211-227.

Pedersen, T. B., Jensen, C. S., and Dyreson, C. E. "A Foundation for Capturing and Querying Complex Multidimensional Data," *Information Systems* (26:5), 2001, pp. 383-423.

Ram, P., and Do, L. "Extracting Delta for Incremental Data Warehouse Maintenance," in *Proceedings of the Sixteenth International Conference on Data Engineering*, San Diego, CA, March 2000, pp. 220-229.

Schiefer, J., Jeng, J. J., and Bruckner, R. M. "Real-Time Workflow Audit Data Integration into Data Warehouse Systems," in *Proceedings of the Eleventh European Conference on Information Systems*, C. Ciborra, R. Mercurio, M. DeMarco, M. Martinez, and A. Carignani (eds.), Naples, September 2003.

Sun Microsystems. "J2EE Connector Specification 1.5," 2002 (available online at **http://java.sun.com/j2ee/connector/**).

Sun Microsystems. "Designing Enterprise Applications with the Java 2 Platform," Enterprise Edition, Second Edition, 2001 (available online at **http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e**).

Todman, C. *Designing a Data Warehouse*, Prentice Hall, Upper Saddle River, NJ, 2001.

Vassiliadis, P., Quix, C., Vassiliou, Y., and Jarke, M. "Data Warehouse Process Management," *Information Systems* (25:3), 2001, pp. 205-236.