

THE EXPERT'S VOICE® IN SQL SERVER

Building a Data Warehouse

With Examples in SQL Server

Vincent Rainardi

apress®



Data Extraction

Now that we have created the tables in the NDS and the DDS, it is time to populate them. Before we can populate them, though, we need to retrieve the data from the source system. This is what data extraction is all about.

First I'll discuss the general principles of data extraction, different kinds of source systems, and several data extraction techniques. Then I'll introduce the tool that we will be using, SQL Server Integration Services. After introducing the tool, I'll then show how to use it to extract the data from the source systems in our case study: Jade, WebTower, and Jupiter.

Introduction to ETL

ETL stands for Extract, Transform, and Load. It is the process of retrieving and transforming data from the source system and putting it into the data warehouse. It has been around for decades, and it has developed and improved a lot since its inception.

There are several basic principles to understand when extracting data from a source system for the purpose of populating a data warehouse. First, the volume of data being retrieved is large, probably hundreds of megabytes or tens of gigabytes. An OLTP system is designed so that the data is retrieved in little pieces, not in large amounts like this, so you have to be careful not to slow the source system down too much. We want the extraction to be as fast as possible, such as five minutes if we can, not three hours (this depends on the extraction method, which I'll cover later in this chapter). We also want it to be as small as possible, such as 10MB per day if we can, not 1GB per day. In addition, we want it to be as infrequent as possible, such as once a day if we can, not every five minutes. We want the change in the source systems to be as minimal as possible, such as no change at all if we can, not creating triggers for capturing data changes in every single table.

The previous paragraph talks about a fundamental principle in data extraction. If there is only thing that you will take away from this chapter, I hope it is this: when extracting data from a source system, you have to be careful not to disturb the source system too much.

After we extract the data, we want to put it into the data warehouse as soon as possible, ideally straightaway, without touching the disk at all (meaning without storing it temporarily in a database or in files). We need to apply some transformations to the data from the source system so that it suits the format and structure of the data in the NDS and the DDS. Sometimes the data transformation is just formatting and standardization, converting to a certain number or date format, trimming trailing spaces or leading zeros, and conforming into standards. Other times the modification is a lookup, such as translating customer status 2 to Active or translating the product category "Pop music" to 54. Another transformation that is

frequently used in data warehousing is aggregation, which means summarizing data at higher levels.

We also want the data we put into the warehouse to be clean and of good quality. For example, we don't want invalid telephone numbers, an e-mail address without the character @ in it, a product code that does not exist, a DVD with a capacity of 54GB, an address with a city of Amsterdam but a state of California, or a unit price of \$0 for a particular audio book. For this purpose, we need to do various checks before putting the data into the warehouse.

Two other important principles are *leakage* and *recoverability*. Leakage happens when the ETL process thinks it has downloaded all the data completely from the source system but in reality has missed some records. A good ETL process should not have any leakage. Recoverability means that the ETL process should be robust so that in the event of a failure, it can recover without data loss or damage. I will discuss all this in this chapter.

ETL Approaches and Architecture

There are several approaches of implementing ETL. A traditional approach is to pull the data from the source systems, put it in a staging area, and then transform it and load it into the warehouse, as per the top diagram of Figure 7-1. Alternatively, instead of putting the data in a staging area, sometimes the ETL server does the transformation in memory and then updates the data warehouse directly (no staging), as shown in the bottom diagram of Figure 7-1. The staging area is a physical database or files. Putting the data into the staging area means inserting it into the database or writing it in files.

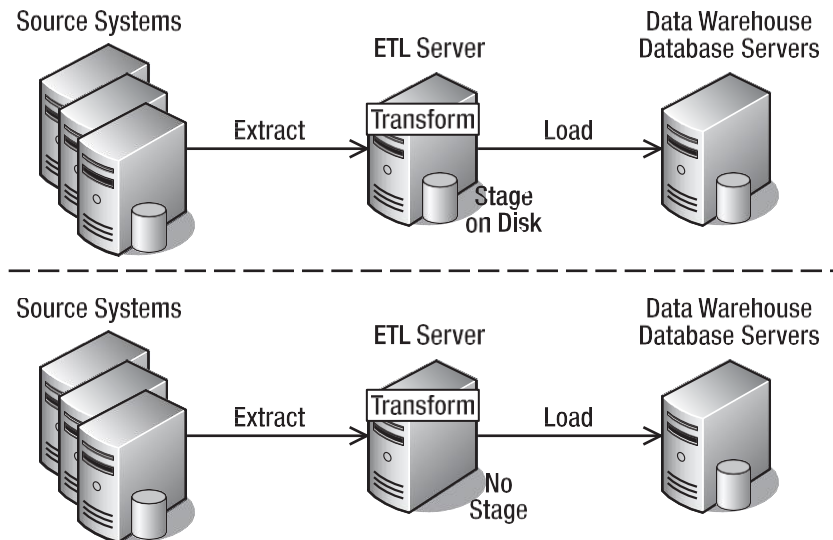


Figure 7-1. To stage on disks or do transformation in memory

Transforming the data in memory is faster than putting it on disk first. If the data is small enough, you can transform in memory, but if the data is big, you need to put it on disk first. Whether you can put the data into memory depends on how much memory the ETL server has.

The alternative to the two ETL approaches shown in Figure 7-1 is called Extract, Load, and Transform (ELT), as shown in the bottom half of Figure 7-2. In the ELT approach, we pull the data from the source systems, load it into the data warehouse, and then apply the transformation by updating the data in the warehouse. In the ELT approach, essentially we copy the source system (OLTP) data into the data warehouse and transform it there. People usually take the ETL approach if they have a strong ETL server and strong software that is rich with all kinds of transformation and data quality processes.

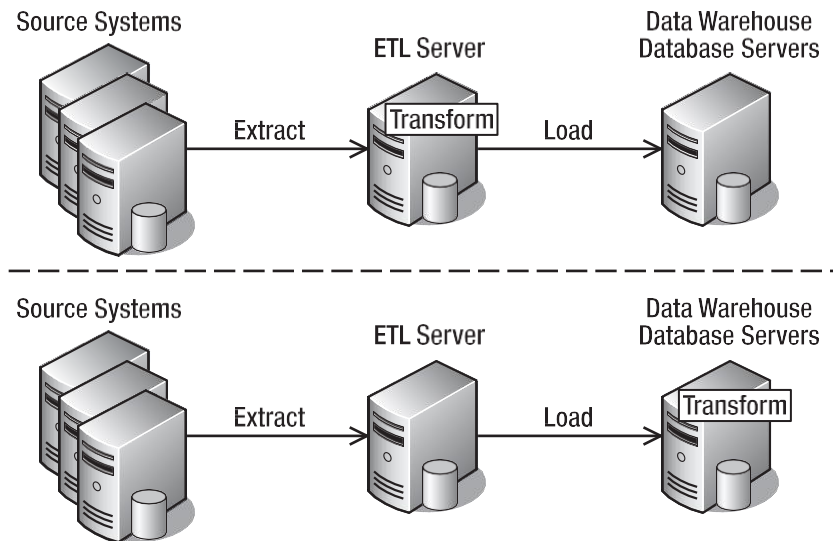


Figure 7-2. *ETL and ELT: choices of where to perform the transformations*

People usually take the ELT approach if they have a strong data warehouse database system, usually MPP database systems. Massively parallel processing (MPP) is a group of servers (called *nodes*), and each node has its own memory, processor, and disk. Examples of MPP database systems are Teradata, Netezza, and Neoview. When you cluster two or more SQL Servers, you get high availability and each node has its own memory and processor, but you still share the disks. In MPP database systems, each node has its own memory, processor, and disk. It is known as a *share nothing* architecture. An MPP database system is more powerful than systems with shared disks because data loading is happening in parallel across multiple nodes that have their own disks.

The main advantage of MPP database systems is that the performance increase is linear. If you put ten SQL Servers in an active-active cluster, the performance increases, but not 10 a single SQL Server. In MPP database systems, if you put in ten nodes, the performance is almost ten times the performance of a single node.

In terms of *who* moves the data out of the source system, we can categorize ETL methods into four approaches (see Figure 7-3):

- An ETL process pulls the data out by querying the source system database regularly. This is the most common approach. The ETL connects to the source system database, queries the data, and brings the data out.
- Triggers in the source system database push the data changes out. A database trigger is a collection of SQL statements that executes every time there is an insert, update, or delete on a table. By using triggers, we can store the changed rows in another table.
- A scheduled process within the source system exports the data regularly. This is similar to the first approach, but the program that queries the database is not an external ETL program. Instead, it is an *internal* exporter program that runs in the source system server.
- A log reader reads the database log files to identify data changes. A database log file contains a record of the transactions made to that database. A log reader is a program that understands the format of the data in the log file. It reads the log files, gets the data out, and stores the data somewhere else.

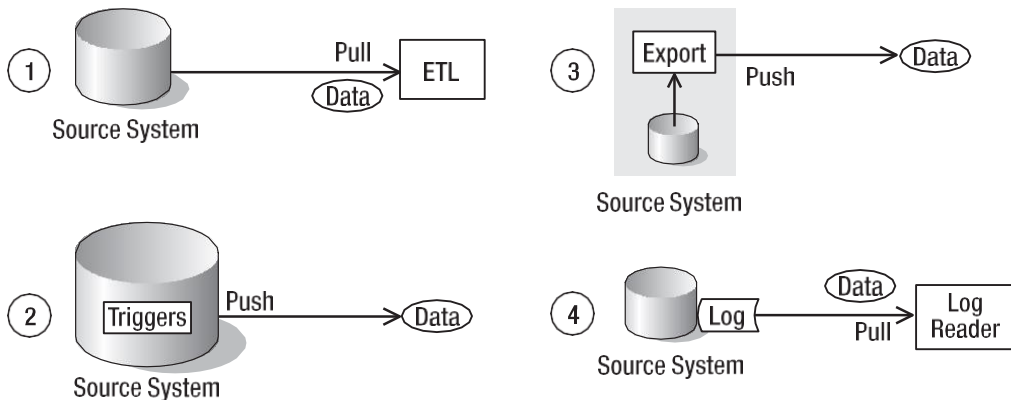


Figure 7-3. Four ETL approaches based on who moves the data out of the source system

In terms of *where* the processes that move the data out are executed, we can categorize ETL into three approaches (see Figure 7-4):

- Execute the ETL processes in a separate ETL server that sits between the source system and the data warehouse server. This approach provides the highest performance. The ETL runs on its own server, so it does not use the resources of the data warehouse server or the source system server at all. It is more expensive than the next two options because you have to purchase additional server and software licenses.
- Execute the ETL processes in the data warehouse server. This approach can be used if we have spare capacity in the data warehouse server or if we have a time slot when the data warehouse is not used (at night for example). It is cheaper than the first approach because we don't need to provide additional servers.

- Execute the ETL processes in the server that hosts the source system. This approach is implemented when we need real-time data warehousing. In other words, the moment the data in the source system changes, the change is propagated to the data warehouse. This can be achieved using database triggers in the source system. In practice, this approach is implemented in conjunction with either of the previous approaches. That is, this approach is used only for a few tables, and the remaining tables are populated using the first two approaches.

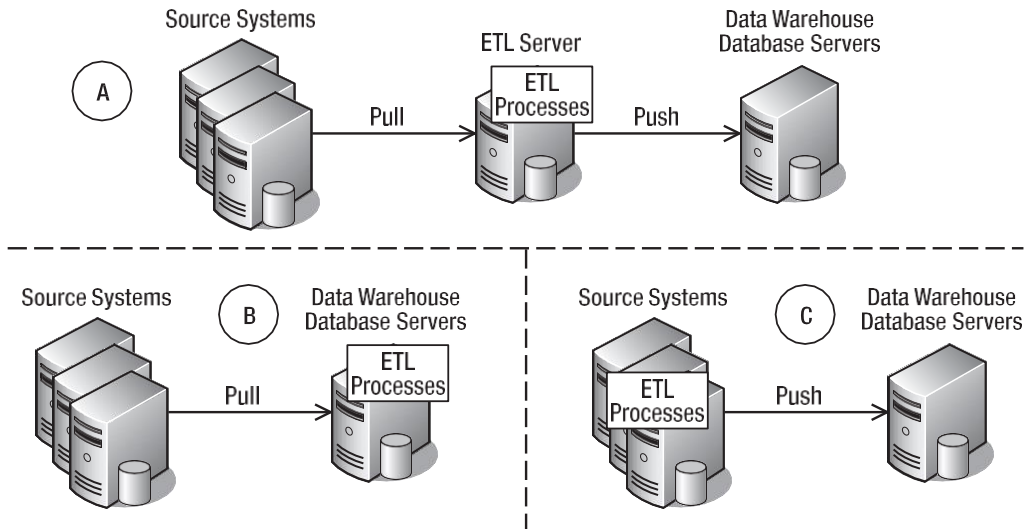


Figure 7-4. Choices of where to put ETL processes

General Considerations

The system we are extracting from may not be a database. It could be a file system, a queue, a service, or e-mails. If the data is in a database, usually we retrieve the data using ADO.NET, ODBC, OLEDB, JDBC, or proprietary database connectivity. These days, most databases are relational, but occasionally we come across hierarchical databases, such as Adabas and ISM, or sequential file storage, such as ISAM. To extract the data from them, we need to have the right database driver or write a data export script using the database-specific language.

If the data is in a file or files, it could be structured, semistructured, or unstructured. A structured file is like this:

ID	Date	Store	Product	Quantity
2893	2/1/08	32	A160	150
2897	2/4/08	29	B120	240

which is a fixed position file, meaning that the fields are in certain column positions. Or it could be like this:

ID|Date|Store|Product|Quantity

2893|2/1/08|32|A160|150

2897|2/4/08|29|B120|240

which is a delimited file. In this case, it is pipe delimited.

A structure file contains tabular (table) data, meaning that the data is essentially in columns and rows format. A semistructured file contains tabular data and nontabular data. A semistructured file looks like this XML file:

```
<order ID="2893">
  <date>2/1/08</date>
  <store>32</stores>
  <product>A160</product>
  <quantity>150</quantity>
</order>

<order ID="2897">
  <date>2/4/08</date>
  <store>29</stores>
  <product>B120</product>
  <quantity>240</quantity>
</order>

<customer ID="83">
  <name>John Smith</name>
  <email>jsmith@aol.com</email>
</order>
```

The first two sections (order data) are tabular, but the third one (customer data) is not tabular.

Unstructured data is like this: “On 2/1/2008 we received order ID 2893 from store 32, requiring 150 units of product A160. Three days later store 29 asked for 240 pieces of B120.” It is the typical content of e-mails. We extract the information in unstructured data using text mining.

Sometimes, the data extraction can be done only during certain times, such as after a batch, or it cannot be done at certain times, such as at backup times or during month-end processing. Sometimes, if we are lucky, we can access the source data at any time, such as when there is a secondary read-only server specifically provided for ad hoc queries and ETL. Table 7-1 lists the potential problems in data extraction, the scenarios, and the common techniques in those scenarios.

Table 7-1. *Potential Problems in Data Extraction*

Potential Problem	Scenario	Common Technique to Overcome
Data is available only at certain times.	A long batch schedule runs overnight followed by a big database backup. Extracting data during the day would slow the OLTP.	Negotiate a time slot with the OLTP DBA. Break the extraction into smaller chunks.
The OLTP database is not available on certain days.	On the first day of every month, OLTP runs a month-end batch for the whole day.	Skip extraction on that day. Tell users the reasons for this limitation.
The OLTP DBA is concerned that our ETL could “mess up” their system.	The project team consists of external consultants who are just learning the OLTP internals.	Ask the DBA to give “read-only” access. Ask the OLTP DBA to export data for you.
There is a concern that ETL will strain/overload the OLTP database.	There are no “quiet times” because the OLTP is a global database.	Read from the secondary OLTP database. Use a less frequent extract for large tables, such as only Sundays.
The OLTP DBA is concerned that they can’t stop ETL processes in an emergency.	There is an emergency situation in OLTP that requires server restart.	Tell the DBA they may terminate you at any time (tell them how) but rerun afterward (manual invoke).
They can’t start ETL processes.	There is an OLTP backup overrun.	Program the ETL to rerun automatically later.

One of the most important things at this stage of the project (the beginning of ETL) is to get to the data, that is, to be able to connect to the source data. Sometimes it takes time to be in a position where we can query the source system (if it is a database) or read the files (if it is a file system). Sometimes the source database server is physically located on another continent and there is no connectivity to that server. I’ve worked on several projects where we had to arrange for the network connectivity to be opened, that is, to allow certain TCP traffic to flow through the firewall so that we could get to the data. Table 7-2 lists the common reasons for not being able to get to the source data, as well as their common solutions.

Table 7-2. *Reasons for Not Being Able to Get the Source Data*

Reason	Common Solution
There is no connectivity to the target database, probably because of no routing or the traffic is blocked on the firewall.	Ask the network engineer to open the firewall. Provide the source IP address, the destination IP address, and the port number to open. Use the NATed IP address if required.
The target database requires a specific driver. We cannot use ODBC, ADO.NET, or OLEDB.	Obtain or purchase the driver and install it on the ETL server.
The target server name is not recognized.	Try it with IP first. If this works, ask the network team to add the name on the DNS. Some database systems such as Teradata add a suffix to the name.

Continued

Table 7-2. *Continued*

Reason	Common Solution
We can reach the database but cannot query because we have no access to the tables or views.	Ask the OLTP DBA to create two accounts: a personal account for development and a functional account for production. Always ask for read-only access.
The database version of the OLTP is newer than the driver on the ETL server.	Upgrade the driver.
The RDBMS requires a specific language or tool, such as Progress 4GL.	Either obtain the ODBC version of the driver (which accepts normal SQL queries) or ask the DBA to write a script to export the data.

In the next three sections, I’ll discuss extracting data from databases, file systems, and other source types.

Extracting Relational Databases

After we connect to the source data, we can then extract the data. When extracting data from a relational database that consists of tables, we can use one of four main methods:

- Whole table every time
- Incremental extract
- Fixed range
- Push approach

Whole Table Every Time

We use the whole table every time when the table is small, such as with a table with three integer or varchar(10) columns consisting of a few rows. A more common reason is because there is no timestamp or identity column that we can use for knowing which rows were updated since the last extract. For example, a general decode table like Table 7-3 could be extracted using the “whole table every time” method.

Table 7-3. *A General Decode Table*

Table	Code	Description
PMT	1	Direct debit
PMT	2	Monthly invoice
PMT	3	Annual in advance
STS	AC	Active
STS	SU	Suspended
STS	BO	Balance outstanding
SUB	S	Subscribed
SUB	U	Unsubscribed
...		

This code-decode table is used throughout the system. Rather than creating a separate table for the payment code, customer status, subscription status, and so on, some business systems use a common table, that is, a single decode table serving the whole system. Of course, this is less flexible, but I've seen this implemented in a source system. Because there was no timestamp, no transaction date (it's not a transaction table), and no identity column either, there was no way we could possibly download the data incrementally, because we couldn't identify which rows were new, updated, or deleted. Luckily, the table was only about 15,000 rows and took about a minute to extract to the stage.

In my experience, a table that contains 1,000 to 5,000 rows (say 10 to 30 columns, or up to 500 bytes in width) on a 100Mbps LAN normally takes one to five seconds to download from DB/2 on an AS/400 using an iSeries OLE DB driver, or from Informix on Unix using an Informix native OLE DB driver into a staging table with no index in SQL Server 2005 or 2000 on a Windows 2000 or 2003 server, configured with RAID 5 for MDF and RAID 1 for LDF using 15,000 RPM disks, running on Dell 2850 with 8MB memory, performed using SSIS or DTS, with no transformation or scripting (just plain column-to-column mapping with data conversion). I cannot give you the formula of how to calculate the time it would take in your environment because different hardware and software configurations have different results, but I can give you a guide: you can estimate the time it would take in your production environment by measuring how long it takes to run the ETL routine on your development or test environment against the test OLTP system.

If the table is small enough, say up to 10,000 rows (this number is based on my experience; it may be different in your environment), it is usually quicker to download the whole table than to constrain the query with a certain *where* clause in the query. If we specify, for example, *select * from stores where (createtimestamp > 'yyyy-mm-dd hh:mm:ss' and createtimestamp <= 'yyyy-mm-dd hh:mm:ss') or (updatetimestamp > 'yyyy-mm-dd hh:mm:ss' and updatetimestamp <= 'yyyy-mm-dd hh:mm:ss')*, it will take the source database engine a few seconds to do the previous four comparisons with every row to identify the rows that we are looking for. This is especially true if the table is not indexed in any of those four columns (and most tables are not indexed in timestamp columns). It would be faster to put just *select * from stores* so that the source database engine can start returning the rows straightaway, without doing any preprocess calculation first.

For tables with fewer than 10,000 rows, I recommend you measure the time it takes to download the whole table and compare it with the time to download it incrementally with some constraints on. We can measure the time it takes to download by using SQL Profiler, by storing "before" and "after" timestamps into a table/file, or by looking at the SSIS log.

Incremental Extract

The transaction tables in major organizations are large tables, containing hundreds of thousands of rows or even hundreds of millions of rows (or more). It could take days to extract the whole table, which is a very disk-intensive operation, decreasing the transactional performance on the front-end application because of a database bottleneck. It is not a viable option (because of the time required to extract), so we need to find a way to extract the data incrementally.

Incremental extraction is a technique to download only the changed rows from the source system, not the whole table. We can use several things to extract incrementally. We can use

timestamp columns, identity columns, transaction dates, triggers, or a combination of them. Let’s explore and study these methods one by one.

Imagine that the order header table in Jade is like the one in Table 7-4.

Table 7-4. *Jade Order Header Table*

Order ID	Order Date	Some Columns	Order Status	Created	Last Updated
45433	10/10/2007	Some Data	Dispatched	10/11/2007 10:05:44	10/12/2007 11:23:41
45434	10/15/2007	Some Data	Open	10/16/2007 14:10:00	10/17/2007 15:29:02
45435	10/16/2007	Some Data	Canceled	10/16/2007 11:23:55	10/17/2007 16:19:03
...					

This table is ideal for incremental extract. It has a “created” timestamp column and a “last updated” timestamp column. It has an incremental order ID column. It has an order date that shows when the order was received.

First we need to check whether the timestamp columns are reliable. A reliable timestamp means that every time the row in the table changes, the timestamp is updated. This can be done by examining the value in both timestamp columns and comparing them with the order date. If they contain dummy values such as 1900-01-01, blank, or null, or if they differ from the order date significantly (say by five months), or if the “last updated” date is less than the “created” date, it’s an indication that we may not be able to rely on them.

We can also do further checks by comparing the timestamp columns to the order ID column; that is, the created date of an order between order ID 1 and order ID 2 should be between their created dates. Note that this does not apply in situations where IDs are assigned in blocks to multiple DB servers. If the timestamp columns are in a good order, we can then use them for incremental extraction in SSIS as follows (see Figure 7-5).

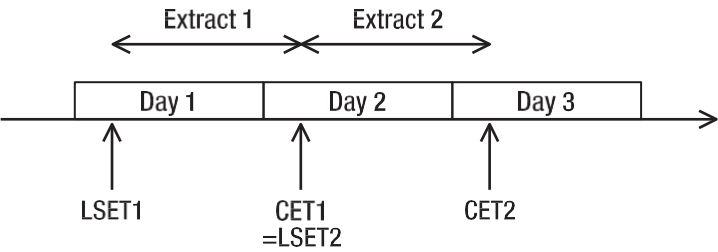


Figure 7-5. *Incremental extraction logic using LSET and CET*

Here’s how to extract incrementally using the current extraction time (CET) and the last successful extraction time (LSET):

1. Retrieve the LSET from the metadata database. LSET acts like a watermark. It memorizes the time when data was last extracted.
2. Get the CET, which is passed in by the top-level ETL package. CET is the time the ETL package started, not when the current task started. The purpose of using the start time of the ETL package rather than the task start time is so that all ETL tasks in the package have the same CET so it's easier to restart if the package failed.
3. Extract the data using *select * from order_header where (created >= LSET and created < CET) or (last_updated >= LSET and last_update < CET)*.
4. If the extract is successful, update the metadata database by writing CET as the new LSET value.

This logic is fault tolerant, meaning that if it doesn't run or it failed to run, we could just rerun it with no risk of missing the data or loading data that we loaded earlier. For example, if the ETL has not been running for two days because of a connection problem to the source system (or the extraction failed for other reasons), it will pick up two days of data because the LSET would still be two days ago.

The purpose of restricting the upper limit with the CET is to exclude orders that were created after the ETL process started. This way, the next time the ETL process runs, it will pick up the orders created after the ETL process started. If we don't put CET as the upper limit, the orders created after the extract begins will be extracted twice.

If the timestamp is not in good order, we can use the order date column. It is a transaction date that reflects when the event happened. But we need to be careful when using the order date column. Although the two timestamp columns are system generated, the order date column is input. Therefore, it is possible that the operative set it to last week's data instead of today, because the order happened last week but just entered into the system today (past-dated orders).

If we apply the previous logic to the *order_date* column, we will miss past-dated orders. To pick up the past-dated orders, we need to add a grace period, as follows: *select * from order_header where order_date >= (LSET - 28 days) and created < CET*. The grace period is obtained from the business rule implemented in the source system application as a restriction or validation; for example, if you try to put the order date as 29 days ago, Jade wouldn't allow it and would generate an error message. This is not ideal, compared to a reliable last-updated date column, so use it as the last resort and test for data leakage (which I'll discuss in a moment).

If there is no source application restriction on the order date, another way of doing incremental extract is to use the order ID as follows (see Figure 7-5, where the logic is the same):

1. Retrieve the Last Successfully Extracted ID (LSEI) from the metadata database.
2. Select *max(order_id)* from *order_header*, and put this in CEI variable as the current extraction ID.
3. Extract the rows between the LSEI and CEI as follows: *select * from order_header where order_id >= LSEI and order_id < CEI*.
4. If the extract is successful, store the CEI in the metadata database as the LSEI.

The previous discussion of fault-tolerant and missed orders still applies; that is, the LSEI provides a fault-tolerant mechanism, and the CEI prevents the orders created after the extract begins from being missed. The past-dated orders and the grace period are also applicable.

How about deletion? How do we know which orders have been deleted? The source systems don't delete the order header records from the previous table. Instead, the orders that have been canceled are marked as canceled in the order status column. This is the ideal condition for ETL, which is known as a *soft delete*, meaning that they don't physically delete the record in the table but mark them only in certain columns.

If for some reason the rows are physically deleted, then there are two ways we can detect the deletion:

By comparing the primary key between the source table and the warehouse table: If we find a primary key that exists in the warehouse table but not in the source table, it means that the row was deleted from the source system. We can then mark the data warehouse row as deleted.

Using deletion trigger; that is, when a record is deleted, the trigger inserts a row into an audit or event table containing the primary key of the deleted row: The ETL then reads the audit table incrementally to find out which rows were deleted from the source system, and we mark the data warehouse rows as deleted.

In addition to detecting deletion, we can use triggers to detect updates and inserts too, which benefits us because it provides a means of capturing the data changes in the source system so we can extract incrementally. We can create separate triggers for delete, update, and insert. A trigger is the most reliable approach in ETL. It is the best way to detect the changes happening in the source systems. The drawback of installing triggers in the source system is an overhead of about 20 to 100 ms per trigger depending on the complexity (these are typical values; I've seen 16 and 400 ms too). This may or may not be a problem depending on your OLTP application, that is, whether it can wait that long or not for a database transaction to complete.

There is one thing we need to be careful of when installing triggers that insert the primary key of the deleted row into an event table: multiple updates. If we write the primary key only of the updated row, we may not be able to get the changed value if there is another update before we read the row. Here is the scenario: in the customer table, row 37 contains the name John. That row was updated at 10:47, changing the name to Alex. The update trigger fired and inserted the primary key of row 37 into the event table. At 10:48, row 37 was updated again, changing the name to David. The trigger fired again and inserted the primary key of row 37 into the event table again. When we read the event table, there were two rows there, both pointing at row 37. When we read the customer table, the name is now David. There is no way we can get the intermediate value (Alex) because it has been overwritten.

In some situations, it is acceptable not to get the intermediate value, but in certain cases it is not acceptable. If it is not acceptable, we can create an audit table that contains all the columns of the source table. When the update trigger fires, it inserts a row into the audit table containing the values of all columns of the updated row, as well as inserting a row into the event table containing just the primary key and a timestamp. Compared with implementing an event table, it takes more overhead to implement an audit table, but we can see the full history of data changes.

Fixed Range

If it is not possible to extract the whole table because the table is too large—and it is not possible to do incremental extraction, for example, because there are no timestamp columns or the timestamp columns are not reliable, because there is no reliable incremental identity column, and because it is not possible to install triggers in the source system—there is one more approach we can do. We can use the “fixed range” method.

Basically, we extract a certain number of records or a certain period of time. For example, say we extract the last six months of data, based on the transaction date. As before, we get the duration of the period from the source application if there is a restriction on the front-end application. For example, once the month-end processing is done, the rows can’t be changed. In this case, we can download the last five weeks of data every time the ETL process runs or where the transaction date is after the month-end date.

If there is no transaction date column in the table and we cannot extract the whole table because it is a huge table, we can use the system-assigned row ID to extract a fixed range, such as the last 100,000 rows. What I meant by row ID is a hidden column in every table containing system-assigned sequential values. Not all database systems have row IDs; for example, Oracle and Informix have row IDs, but SQL Server and DB/2 don’t. (In DB/2, the row ID is a data type, not a hidden column.) When using the row ID, we have no restrictions on the front-end application, so we need to monitor the source system and find out how many we need to extract every time. Download the primary key column(s) every day, and compare between each daily download to detect the changes. Identifying new rows and deleted rows by comparing primary keys is quite straightforward. Let’s have a look at Table 7-5.

Table 7-5. *Comparing Daily Download to Detect New and Deleted Rows*

10/1	10/2	10/3	10/4	10/5
5467	5467	5467	5467	5467
8765	8765	3422	3422	3422
	3422	6771	6771	6771
			1129	1129

In this table, 8765 was deleted on 10/3, 6771 was created on 10/3, and 1129 was created on 10/4. But identifying updates is more difficult. For example, it seems that there are no new or deleted rows on 10/5, but in fact there is an update on row 3422. Updates can be detected using checksum, but we need to download the columns that we want to compare to the stage first, and this will be time-consuming if the table is large. Assuming table1 contains yesterday’s data and table2 contains today’s data, we can do the checksum comparison as illustrated here:

```
create table table1 (col1 int, col2 int, col3 int)
insert into table1 values(3434, 4565, 2342)
insert into table1 values(2345, 3465, 6321)
insert into table1 values(9845, 9583, 8543)
go
```

```

create table table2 (col1 int, col2 int, col3 int)
insert into table2 values(3434, 4565, 2342)
insert into table2 values(2345, 8888, 8888)
insert into table2 values(9845, 9583, 8543)
go

alter table table1 add col4 as checksum(col1, col2, col3)
alter table table2 add col4 as checksum(col1, col2, col3)
go

select * from table1
select * from table2
select * from table1 t1
where not exists
( select * from table2 t2
  where t1.col4 = t2.col4 )
go

```

After we are able to identify the changed rows, we can now define how far back we need to download the data every time the ETL process runs, that is, the last 10,000 rows, the last 100,000 rows, the last three weeks, the last two months, and so on.

Related Tables

If a row in the source table is updated, we need to extract the corresponding row in the related table too. For example, if order ID 34552 in the order header table is updated and extracted to the data warehouse, the rows for order ID 34552 in the order detail table need to be extracted to the data warehouse too, and vice versa. For example, if a row in the order detail is updated and that row is extracted into the data warehouse, the corresponding row in the order header needs to be extracted too.

This is also the case for inserts and deletes. If a new row (a new order) is inserted into the order header in the source system, the corresponding order detail rows need to be inserted into the data warehouse order detail table too. If a row is marked as canceled (soft delete) in the order header in the source system, the corresponding order detail rows need to be canceled too. We can do this in the data warehouse application too, but ideally it is done in the data warehouse database. If a row is physically deleted in the order header, it needs to be marked as deleted in the data warehouse order detail.

To do this, we identify the changed rows in the first table, and then using the primary key and foreign key relationship, we identify the rows in the second table, and vice versa. For example, in the case of the order header and order detail tables, we find the changed rows on the order header first, then we identify the changed rows in the order detail, and finally we extract *both sets of rows* from *both tables* into the data warehouse.

Testing Data Leaks

If we do either incremental extraction or period range extraction, it is essential that we test for data leaks. Say we think we have extracted all changes in the source system into our data warehouse. Now let's test it. We need to build the incremental or fixed-range ETL and run it every day (or four times a day or any other frequency). After a few weeks, we compare the number of rows between the source system and the data warehouse. Then we identify whether there are any missing rows or missing updates by comparing the checksum. That is, table1 contains the data from the ETL that has been running for two weeks, and table2 contains the data from the source system as it is today, so we compare the checksum columns from both tables:

```
select * from table2 where not exists
(select * from table1
 where table1.checksum_column = table2.checksum_column )
```

The reverse is also true. For rows that exist in table1 but do not exist in table2, use this:

```
select * from table1 where not exists
(select * from table2
 where table1.checksum_column = table2.checksum_column )
```

If we don't have any missing rows or any missing updates, then our incremental ETL process is reliable. Let it run for a few more weeks while you develop other parts of the DW system, and then do the previous test again. If you find missing rows, check your ETL logic, the LSET, the CET, and so on. If the ETL logic is correct, then consider another approach; for example, use a trigger to capture data changes or use a fixed-period extract if there is no reliable timestamp column.

If we extract from a relational database, it is very important to always test for data leaks. If we missed some rows or some updates, it means that the data in our data warehouse is not reliable. Remember that no matter how good the data warehouse functionality is, if the data in our warehouse is wrong, then it is unusable.

Extracting File Systems

The most common type of *files* acting as sources in the ETL process are flat files. Two examples of flat files, fixed-position files and pipe-delimited files, were shown earlier in the chapter. Flat files are common because they provide the best performance. Importing or exporting from a flat file is probably the fastest, compared to importing from other types of files (XML, for example). SQL Server's *bulk insert* and bulk copy utility (*bcp*) work with flat files.

bulk insert is a SQL command to load data from a file into a SQL Server table. *bulk insert* is executed from within the SQL Server query editor like a regular Transact SQL command. A typical usage of *bulk insert* is to load a pipe-delimited file into a table:

```
bulk insert table1 from 'file1' with (fieldterminator = '|')
```

The bulk copy utility is executed from a command prompt. To use the bulk copy utility to load data from a pipe-delimited file into a table, a typical usage is as follows:

```
bcp db1.schema1.table1 in file1 -c -t "|" -S server1 -U user1 -P password1
```

It is quite a common practice in data warehousing for the source system to export the data into flat files, and then an ETL process picks them up and imports them into the data warehouse. Here are some of the things you need to consider when importing from flat files:

- Agree with the source system administrator about the structure of the file system. This includes the file naming convention (fixed file name such as *order_header.dat* or dynamic file names such as *order_header_20071003.dat*), the directory structure (one directory per day), and so on. *Guidelines:* In my experience, it is more useful to use dynamic file names so we can leave the last *n* days of extract files on disk before deleting them.
- Make sure you have access to the agreed-upon location. It could be a shared network folder. It could be an FTP server. Whatever and wherever the location is, you need to have access to it. You may need permissions to delete files as well as read the files. Find out what user ID you must use to connect to the file server or FTP server. Do not underestimate this, because it can take a long time. *Guidelines:* Start raising the change requests for having read-write access to the file location as soon as possible. Do this for the development, QA, and production environments.
- Agree on the process; for example, after processing each file, do you need to delete the files or move them to an archive directory? If you move them to an archive directory, who will be deleting them from there later? How long will the files be stored in the archive directory? Will it be a year, six months, and so on? *Guidelines:* Determine the archive period based on backup procedure. That is, don't delete from disk before it's backed up to tape. Modify the extraction routine to include the movement of extracted files to the archive directory and to delete archived files older than the archive period.
- Agree on failure handling. For example, if for some reason the source systems are unable to produce the files yesterday, when their export process works again today, will the files be placed in the yesterday directory or the today directory? If both the yesterday and today data goes into the today directory, will it be two files or one file? *Guidelines:* I'd put them in the today directory. The main principle of failure handling is that the ETL process must be able to be rerun (or miss a run) without causing any problems.
- Agree on the frequency that the source systems run their export. Is it once a day, four times a day, and so on? *Guidelines:* This loading frequency depends on how often they export and on how often the business needs to see the data. The ideal loading frequency is the same as the export frequency (loading: put the data into the DW; exporting: retrieve the data from the source system).

- Agree on the file format. Will it contain a column heading, is it going to be a delimited file or a fixed-position file (delimiter is better, because the file sizes are smaller), what is the delimiter (don't use commas because the data can contain a comma; a pipe is a good choice), are you enabling Unicode (the 2-byte characters), what's the maximum line length, and what's the maximum file size? *Guidelines:* In my experience, it is not good to use tabs, commas, colons, backslashes, or semicolons. I usually test the OLTP data for pipes (|). If the data does not contain a pipe, then I use a pipe as the delimiter. Otherwise, I use a tilde (~) or this character: ¬. Both are rarely used/found in the data. Also test for newline characters, and if you find any, replace them when exporting.
- Agree on which columns are going to be exported by the source system and in what order. This is so that you can prepare column-to-column mapping for your import routine, whether you are using SSIS or bcp. *Guidelines:* For simplicity, consider leaving the order as is but choose and export only required columns. Refrain from exporting all columns. This is useful to minimize incremental extract size, especially if the table is very large (more than 1 million rows) and the data changes frequently.

If you are importing spreadsheet files such as Excel files, the guidelines are more or less the same as the previous ones. The main difference is that you don't need to worry about the delimiter. You can't use *bcp* or *bulk insert* for importing Excel files. In SQL Server we use SSIS for importing Excel files. Spreadsheets are quite commonly used for source files because they are widely used by the business users. For example, annual budgets or targets may be stored in Excel files.

Web logs are the log files of a web site, located within the web servers. Each web log is a text file containing the HTTP requests from the web browsers to the servers. It contains the client IP address, the date and time of the request, which page was requested, the HTTP code reflecting the status of the request, the number of bytes served, the user agent (such as the type of web browser or search engine crawlers), and the HTTP referrer (the page this request came from, that is, the previous page from which the link was followed).

People are interested in web logs because they can give useful browsing and shopping information in an e-commerce web site, that is, who browsed for what and when. The purpose of extracting from web logs into a data warehouse (instead of using web analysis software) is to integrate the web traffic data with the data already in the warehouse, such as to monitor the results of a CRM campaign, because the page requests in the campaign have additional query strings identifying the particular e-mail campaign from which they originated.

Web logs come in different formats. Apache HTTP Server 1.3 uses Common Log Format and Combined Log Format. IIS 6.0 uses W3C Extended Log Format. Once we know the format, we can parse each line of the log files into a separate entry/field, including the client IP address, username, date, time, service, server name, and so on. We can then map each field into a column in the data warehouse target table and load the data.

Although they're not as common as flat files and spreadsheets, database transaction log files and binary files can also be used as a source of ETL processes. Database transaction log files (in Oracle this is known as *redo logs*) are utilized by applying the transactions in the log files into a secondary copy of the database by using log shipping or by reading the files

and processing them using specialized software (a different tool is used for each RDBMS). Transaction logs are very useful because we do not touch the primary database at all. They come at no cost to us. The transaction logs are produced and backed up for the purpose of recovering the database in the event of a failure. They are there doing nothing if you like. By reading them and extracting the transactions into our data warehouse, we avoid touching the source database. And the logs are quite timely too; for example, they contain a constant stream of data from the database. This is probably the only extraction method that does not touch the source relational database.

To do this, you need a specific tool or software that can read the transaction log files of your source system database. Different database systems have different log file formats and transaction log architecture. An example of such a tool or software is DataMirror. If the source system is a SQL Server, we can apply the transaction log files into a secondary server using log shipping. Then we read the data from the secondary server.

Binary files containing images, music samples, film trailers, and documents can also be imported into a data warehouse by using tools such as ADO.NET (or other data access technologies) and storing them as binary or varbinary data types in SQL Server tables.

These days, XML files are becoming more and more commonly used as source data in the ETL. We can use SSIS to read, merge, validate, transform, and extract XML documents into SQL Server databases. To do this, we use the XML task in the SSIS control flow and specify the operation type. Validation can be done using the XML schema definition (XSD) or document type definition (DTD). To do this, we specify the operation type as Validate in the SSIS XML task. Note that the DTD is now superseded by XSD. We can also use XML bulk load COM objects or OpenXML Transact SQL statements to read and extract XML documents. To use OpenXML, we call *sp_xml_preparedocument* first to get the document handle. We can then do *select * from openxml (doc handle, row pattern, mapping flag) with (schema declaration)* to read the content of the XML document.

Extracting Other Source Types

Relational database and flat files are the most common types of source data for the ETL processes of a data warehouse system. I have also briefly mentioned spreadsheet files, web logs, hierarchical databases, binary files, database transaction logs, and XML files. Other types of source data are web services, message queues, and e-mails. I'll discuss these types in this section so you can understand what they are, how they are normally used, and how to extract them in our ETL processes into a data warehouse.

A *web service* is a collection of functions that perform certain tasks or retrieve certain data, exposed by a web interface to receive requests from web clients. Well, we don't really extract a web service, but we *use* a web service to get the data. In a service-oriented architecture (SOA) environment, we don't access the database directly. Instead, the data is "published" using a collection of web services. We can, for example, request a list of products (with their attributes) that were updated in the last 24 hours.

The benefit of using a web service to get the data is that the source system can have a single, uniform mechanism to publish its data. All the consumers of this data are sure that the data they receive is consistent. I have been on a data warehouse project where the source system had already published its data using a web service so the data consumers (the data warehouse is one of them) can just tap into this service. Although this is fine for operational

purposes where the data is a trickle feed (in other words, a small amount), the web service approach cannot be used for initial bulk loading where we need to load large volumes of data. The performance would be poor. For example, if each transaction takes tens or hundreds of milliseconds, it would take weeks or months to load millions of rows of initial data. We need to use another mechanism for initial data loading.

A message queue is a system that provides an asynchronous communication protocol, meaning that the sender does not have to wait until the receiver gets the message. System A sends messages to the queue, and system B reads the messages from the queue. A queue manager starts and stops the queue, cleans up unprocessed messages, sets threshold limits (maximum number of messages in the queue), logs the events, performs security functions such as authentications, deals with errors and warnings such as when the queue is full, governs the order of the queue, manages multicasting, and so on. To put a message queue into ETL context and to understand the implementation of message queues for ETL systems, let's go through an example from a project.

A source system captures the changes using insert, update, and delete triggers. The system has several event log tables to record the changes that happened in the system. At certain periods (typically between one and four hours, which is determined based on the business requirement, that is, how recent we need the DW data to be), the ETL routines read the event logs incrementally and retrieve the changed data from the source database using web services that return the data in XML format. The ETL routines then wrap the XML in outer envelopes (also in XML format) and send them as XML messages to message queues over a VPN. These message queues are located in the global data warehouse, so other subsidiary companies also use the same message queues for sending their data. The global data warehouse then reads the messages in the message queues in a multicast mode; that is, there were two redundant ETL routines reading each message from the same queue into two redundant relational stores within the data warehouse. In this case, the global data warehouse reads the MQ as an ETL source.

E-mails are stored in e-mail servers such as Microsoft Exchange and Java Email Server (JES). E-mails are accessed using application programming interfaces (APIs), Collaboration Data Objects (CDO), ADO.NET, or an OLEDB provider. For example, OLE DB for Exchange Server enables us to set up an Exchange Server as a linked server in SQL Server so that the e-mails stored in Exchange Server are exposed in a tabular form and so we can use a normal SQL *select* statement within SSIS to retrieve the data.

Extracting Data Using SSIS

Now let's try to extract data from Jade into the stage database using SSIS. Before we start, let's go through an overview of what we are going to do. To extract data from Jade into the stage database, in the next few pages we will be doing the following steps:

1. Create the source system database and a user login.
2. Create a new SSIS project.
3. Create data sources for Jade and the stage.
4. Create a Data Flow task.

5. Probe the source system data.
6. Create a data flow source.
7. Create a data flow destination.
8. Map the source columns to the destination.
9. Execute the SSIS package.

Before we open Business Intelligence Development Studio and start creating SSIS packages to extract data from Jade, we need to create a Jade database first. To simulate a Jade database, which in the Amadeus Entertainment case study is an Informix database, we will create a database called Jade in SQL Server. To create this database, we will restore it from a backup available on the Apress web site using the following steps:

1. Download the file called *Jade.zip* (about 6MB) from this book's page on the Apress web site at <http://www.apress.com/>.
2. Uncompress it to get *Jade.bak* from that file.
3. Put *Jade.bak* in your development SQL Server, say on c:\. This file is a backup of a SQL Server database named Jade, which we will use to simulate Jade as the source system.
4. Open SQL Server Management Studio, and restore the Jade database from *Jade.bak*, either by using SQL Server Management Studio or by typing the following in a query window, replacing *[sqldir]* with the SQL Server data directory:

```
restore database Jade from disk = 'c:\Jade.bak'
with move 'Jade_fg1' to '[sqldir]\Jade_fg1.mdf',
move 'Jade_log' to '[sqldir]\Jade_log.ldf'
```

We need a stage database for the target, and we created a stage database in Chapter 6. We also need to create a login to access Jade and the stage databases. To create a login called *ETL* with *db_owner* permission in both the stage and Jade databases, and with the default database set to the stage, do the following:

```
create login ETL with password = '[pwd]', default_database = stage
go
use Jade
go
create user ETL for login ETL
go
sp_addrolemember 'db_owner', 'ETL'
go
use Stage
go
create user ETL for login ETL
go
sp_addrolemember 'db_owner', 'ETL'
```

```

go
use Meta
go
create user ETL for login ETL
go
sp_addrolemember 'db_owner', 'ETL'
go

```

Replace *[pwd]* with a complex password that meets SQL Server security standards, or use Windows integrated security (Windows authentication mode). In the previous script, we also create a user for that login in the metadata database. This is to enable the *ETL* login to retrieve and store the extraction timestamp in the metadata database, which we will do in the next section.

Open Business Intelligence Development Studio. Click File  New  Project, select Integrated Services Project, and name the project **Amadeus ETL**. Click OK to create the project. The screen will look like Figure 7-6.

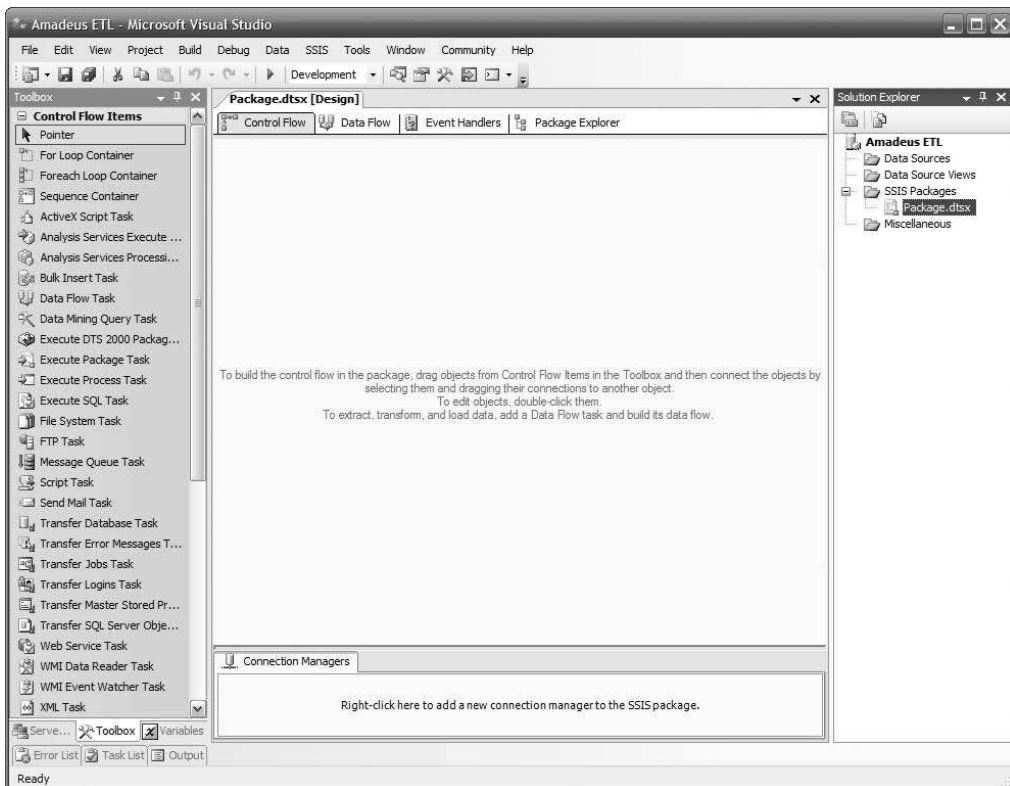


Figure 7-6. *Business Intelligence Development Studio initial project screen*

Now let's set up a data source for Jade:

1. In the top-right corner, right click Data Sources, and choose New Data Source. In the Data Source Wizard, click New.
2. In the Provider drop-down list, you would select IBM OLE DB Provider for Informix, but for this case study we will choose Microsoft OLE DB Provider for SQL Server in order to simulate the connection to Informix (in practice, if your source system is SQL Server, choose SQL Native Client).
3. Enter your development server name, select Use SQL Server Authentication, type **ETL** as the username along with the password, check "Save my password," and select Jade for the database name. If you use Windows integrated security, choose Use Windows Authentication.
4. The screen should look like Figure 7-7. Click Test Connection, and click OK.
5. Ensure that servername.Jade.ETL in the Data Connections box is selected, and click Next. Name it **Jade**, and click Finish. You will see now that under Data Sources we have *Jade.ds*.

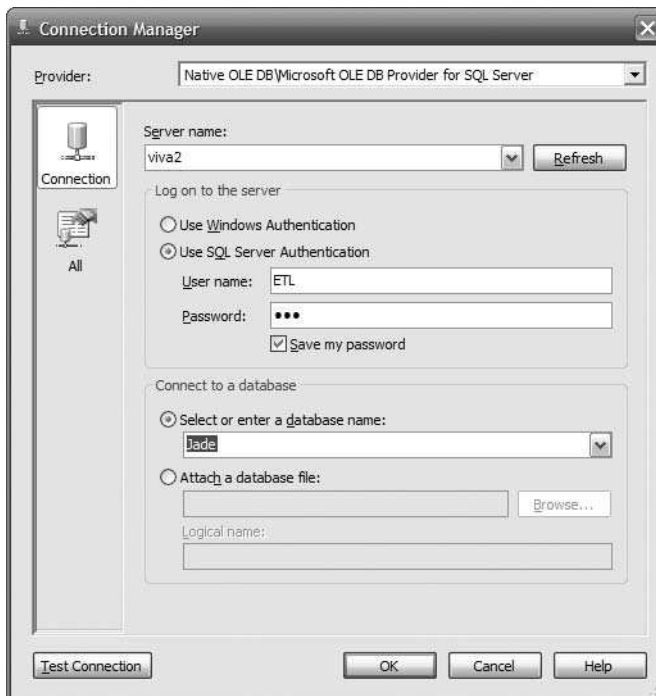


Figure 7-7. Using the Connection Manager dialog box to set up a connection to Jade

Repeat the same process to set up another data source for the stage, as follows:

1. Choose SQL Native Client as the provider.
2. Type the server name, username, and password the same as before.
3. Enter the database name as **Stage**.
4. Click Test Connection, and click OK if succeeds.
5. Make sure servername.Stage.ETL in the Data Connections box is selected, and click Next.
6. Name it **Stage**, and click Finish. Notice that you now have two data sources in the top-right corner.

Double-click the Data Flow task in the Toolbox on the left side. A Data Flow task will appear in the top-left corner of the design surface. Double-click that Data Flow task in the design surface, and the design surface will change from Control Flow to Data Flow. Double-click OLE DB Source under Data Flow Sources in the Toolbox on the left side. The OLE DB Source box will appear in the design surface, as shown in Figure 7-8.

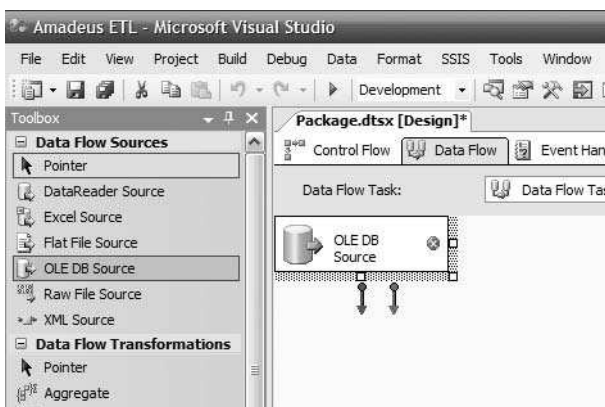


Figure 7-8. *Creating the OLE DB source in the Data Flow task*

When we are facing an unknown source system, we need to be careful. It could be a huge table containing 500 million rows. It could contain strange characters in Extended Binary Coded Decimal Interchange Code (EBCDIC). Perhaps we are not even familiar with the table names and column names. Perhaps we are even not familiar with the syntax of the SQL language in that platform. The last thing we want is to cripple that source system with our queries. Here I will illustrate how we can “probe” an unknown source system carefully.

First we need to know how many rows are in the source table. In practice, the source system may not be an SQL Server, so we can’t query it directly. Therefore, we use SSIS to count the rows as follows:

1. Click the OLE DB Source box on the design surface.
2. Hit F2, and rename it to **Jade order_header**.

3. Double-click the “Jade order_header” box.
4. In the OLE DB Connection Manager, click New.
5. Select servername.Jade.ETL, and click OK.
6. In the Data Access Mode drop-down list, choose SQL Command.
7. Click Build Query, and Query Builder window opens.
8. Click the rightmost icon, Add Table. This is so that we know what the tables are in the source system.
9. Click Close.
10. In the SELECT FROM section of the window, type **select count(*) from order_header**, and click OK.
11. Click Preview. If it takes a long time and it shows a big number (such as a few million or more), we need to be careful not to extract the whole table.
12. Click Close to close the table list.

Modify the SQL Command Text box to say *select top 10 * from order_header*, and click Preview. Every RDBMS has a different syntax, as follows:

*Informix: select first 10 * from order_header*

*Oracle: select * from order_header where rownum() <= 10*

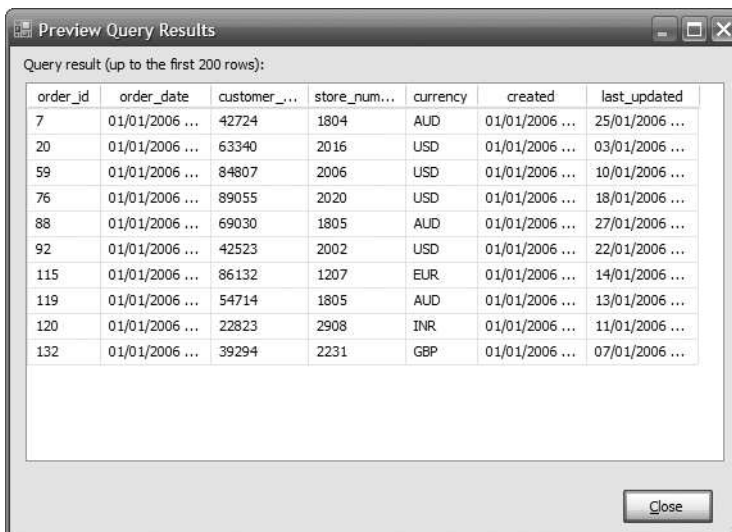
*DB/2: select * from order_header fetch first 10 rows only*

*Mysql and Postgres: select * from order_header limit 10*

*Sybase: set rowcount 10; select * from order_header;*

*Teradata: select * from order_header sample 10*

Figure 7-9 shows the result.



Query result (up to the first 200 rows):

order_id	order_date	customer_...	store_num...	currency	created	last_updated
7	01/01/2006 ...	42724	1804	AUD	01/01/2006 ...	25/01/2006 ...
20	01/01/2006 ...	63340	2016	USD	01/01/2006 ...	03/01/2006 ...
59	01/01/2006 ...	84807	2006	USD	01/01/2006 ...	10/01/2006 ...
76	01/01/2006 ...	89055	2020	USD	01/01/2006 ...	18/01/2006 ...
88	01/01/2006 ...	69030	1805	AUD	01/01/2006 ...	27/01/2006 ...
92	01/01/2006 ...	42523	2002	USD	01/01/2006 ...	22/01/2006 ...
115	01/01/2006 ...	86132	1207	EUR	01/01/2006 ...	14/01/2006 ...
119	01/01/2006 ...	54714	1805	AUD	01/01/2006 ...	13/01/2006 ...
120	01/01/2006 ...	22823	2908	INR	01/01/2006 ...	11/01/2006 ...
132	01/01/2006 ...	39294	2231	GBP	01/01/2006 ...	07/01/2006 ...

Figure 7-9. Previewing the first ten rows

This selection of the first ten rows has three purposes:

- To know the column names. We need these names for further queries. If the source system DBA provides us with the data dictionary, that's great. If not, here the names are. If you are familiar with the system tables of your source system and your user account has administrative privileges, you can query the source system metadata to get the column names. For example, in SQL Server it's catalog views, in Teradata it's DBC, and in Oracle it's system tables.
- To understand the content of the data for each column, scroll the window to the right. In particular, look for columns containing nonstandard characters (usually shown as little blank rectangles) and the date format.
- If *count(*)* is large, this “ten rows” query enables us to understand whether the source system has good response time or is very slow. If the table is huge but it is partitioned and indexed well, we should expect the first “ten rows” query to return in less than two seconds. Bear in mind that the response time is also affected by other factors such as system load, hardware and software configuration, throttling, and so on.

Then select the changed data by typing this in the SQL Command Text window:

```
select * from order_header
where (created > '2007-12-01 03:00:00'
and created <= '2007-12-02 03:00:00')
or (last_updated > '2007-12-01 03:00:00'
and last_updated <= '2007-12-02 03:00:00')
```

Change the dates to yesterday's date and today's date, and click Preview. Don't worry about the dates being hard-coded; later in this chapter you'll learn how to make this dynamic by retrieving and storing into the metadata database. Don't worry about the risk that it will return 400 million rows in 4 hours; SSIS will return only the first 200 rows because SSIS limits the output when previewing data. Scroll down and scroll to the right to examine the data. If everything looks all right, change the data access mode to Table or View, choose *order_header* as the name of table, click Preview, click Close, and then click OK.

Scroll down in the Toolbox. Under Data Flow Destination, double-click SQL Server Destination. The SQL Server Destination box will appear in the design surface. Click that box, press F2, and rename it to **Stage order header**. Resize the box if necessary. Click the “Jade order header” box, and pull the green arrow to the “Stage order header” box. Double-click the “Stage order header” box. In the “OLE DB connection manager” drop-down list, click New. In the Data Connection box, select *servername.Stage.ETL*, and click OK.

You can create the destination table in the stage manually, based on the source system column data types. You can also create it on the fly using the New button to the right of the table or the View drop-down list. I prefer to create it manually so I can be sure of the data types, table name, column name, constraints, and physical location/filegroup, as follows (of course you can use SQL Server Management Studio rather than typing the SQL statement):

```

create table order_header
( order_id          int
, order_date        datetime
, customer_number   int
, store_number      int
, created           datetime
, last_updated      datetime
) on stage_fg2
go

```

Click OK in Create Table window to create the *order_header* table in the stage database. In the table or view, choose *order_header* that we just created. Click Preview, and it shows an empty table. Click Close, and then click Mapping on the left side. Figure 7-10 shows the outcome.

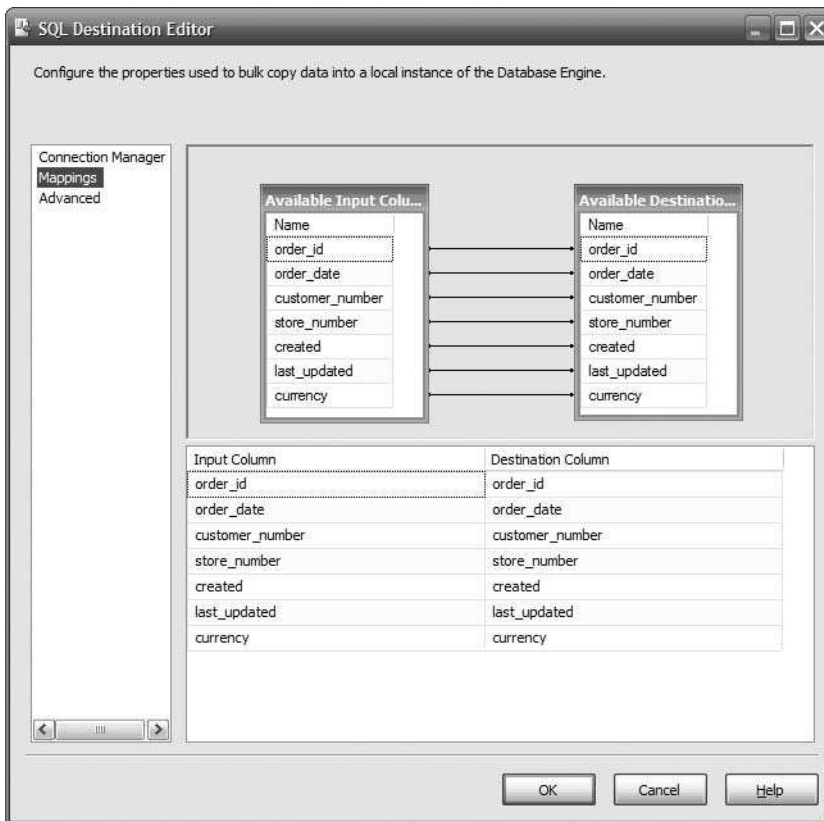


Figure 7-10. Mapping source columns to the stage table columns

If the column names are different, you need to map them manually. If the column names are the same, they are mapped automatically. Click OK to close this window. Press F5 or the green triangle icon in the toolbar to run the package. It will fail with a “Stage order_header” box marked in red, as shown in Figure 7-11.

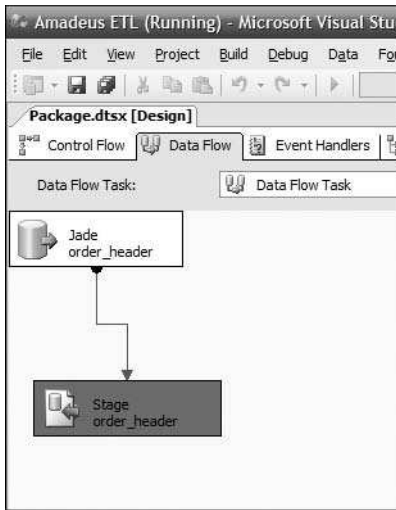


Figure 7-11. Running the SSIS package

Click the Progress tab. You will see that it says “You do not have permission to use the bulk load statement.” Press Shift+F5 or the blue rectangle to stop running the package. SSIS shows the Execution Results tab, displaying the same error message. This is a common error. The cause was that the user *ETL* does not have the bulk insert.

Click the Data Flow tab, and then click the “Stage_order_header” box. Scroll down the properties on the right side, as shown in Figure 7-12.

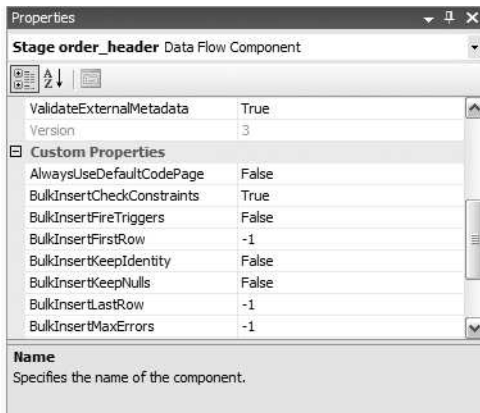



Figure 7-12. Bulk insert properties

The SQL Server destination uses the bulk insert to load data into SQL tables. Therefore, the login that we use needs to have a bulkadmin server role. You can use SQL Server Management Studio or SQL statement to assign this role. The SQL statement is `sp_addsrvrolemember 'ETL', 'bulkadmin'`.

Rerun the package. Now both boxes in the Data Flow tab should be green, meaning that SSIS has successfully imported those rows from the source table to the stage table. Click Stop, or press Shift+F5. Click the blue disk button (or select File  Save) to save your SSIS package. Go to SQL Server Management Studio, and query the *order_header* table in the stage database to verify that you have the rows in the stage.

Memorizing the Last Extraction Timestamp

Memorizing the last extraction timestamp means that we memorize the last timestamp of the data we extracted so that on the next extraction we can start from that point. We have just successfully extracted the order header records from Jade into the stage. But we used fixed values for the date range. Now let's store the last extraction timestamp in the metadata database so that every time the ETL process runs it will extract different records. The ETL process will extract only those records that were added after the last extract.

Before we begin, let's go through an overview of what we are going to do. In the next few pages, we will use the last extraction timestamp to control the next extraction. This enables us to extract the source table incrementally. To do that, in the next few pages we will be doing the following steps:

1. Create a table in the metadata database to store the timestamps.
2. Create a data source for the metadata database.
3. Modify the data flow as follows:
 - a. Store the current time.
 - b. Get the last extraction timestamp from the table.
 - c. Add timestamps to the extraction query as parameters.
 - d. Update the timestamps in the table.
4. Clear the target table in the stage database.
5. Set the initial timestamps in the metadata table.
6. Execute the package to populate the target table.

So first, let's create the *data_flow* table as follows:

```
use meta
go

if exists
( select * from sys.tables
  where name = 'data_flow' )
drop table data_flow
go
```



```

create table data_flow
( id          int          not null identity(1,1)
, name        varchar(20) not null
, LSET        datetime
, CET         datetime
, constraint pk_data_flow
    primary key clustered (id)
)
go

create index data_flow_name
on data_flow(name)
go

declare @LSET datetime, @CET datetime
set @LSET = '2007-12-01 03:00:00'
set @CET = '2007-12-02 03:00:00'
insert into data_flow (name, status, LSET, CET)
    values ('order_header', 0, @LSET, @CET)
insert into data_flow (name, status, LSET, CET)
    values ('order_detail', 0, @LSET, @CET)
insert into data_flow (name, status, LSET, CET)
    values ('customer', 0, @LSET, @CET)
insert into data_flow (name, status, LSET, CET)
    values ('product', 0, @LSET, @CET)
go

```

Note LSET stands for Last Successful Extraction Timestamp, and CET means Current Extraction Timestamp.

Verify that we have that *order_header* row in the *data_flow* table: *select * from data_flow*. You should see four rows, as shown in Table 7-6.

Table 7-6. *data_flow* Table

id	name	LSET	CET
1	<i>order_header</i>	2007-12-01 03:00:00.000	2007-12-02 03:00:00.000
2	<i>order_detail</i>	2007-12-01 03:00:00.000	2007-12-02 03:00:00.000
3	<i>customer</i>	2007-12-01 03:00:00.000	2007-12-02 03:00:00.000
4	<i>product</i>	2007-12-01 03:00:00.000	2007-12-02 03:00:00.000

OK, now let's modify our *order_header* Data Flow task in our Amadeus ETL package so that it first sets the CET to the current time and then reads the LSET. If the data flow is successfully executed, we update the LSET and status for that data flow. If it fails, we set the status to fail, and we don't update the LSET. This way, the next time the data flow is run, it will pick up the records from the same LSET.

So, let's create a new data source called Meta that connects to the metadata database using the *ETL* user. The procedure is the same as the last time. This is why in the previous section, when we created the Jade database, we also assigned the user *ETL* as *db_owner* in the metadata database. Now we will set the CET to the current time for all rows.

1. In the Toolbox on the left side, double-click the Execute SQL task. There will be a new box on the design surface labeled Execute SQL Task. Right-click, and rename this box as **Set CET**.
2. Double-click this box. In the SQL Statement section, click the Connection drop-down list, and select <New connection...>, as shown in Figure 7-13.

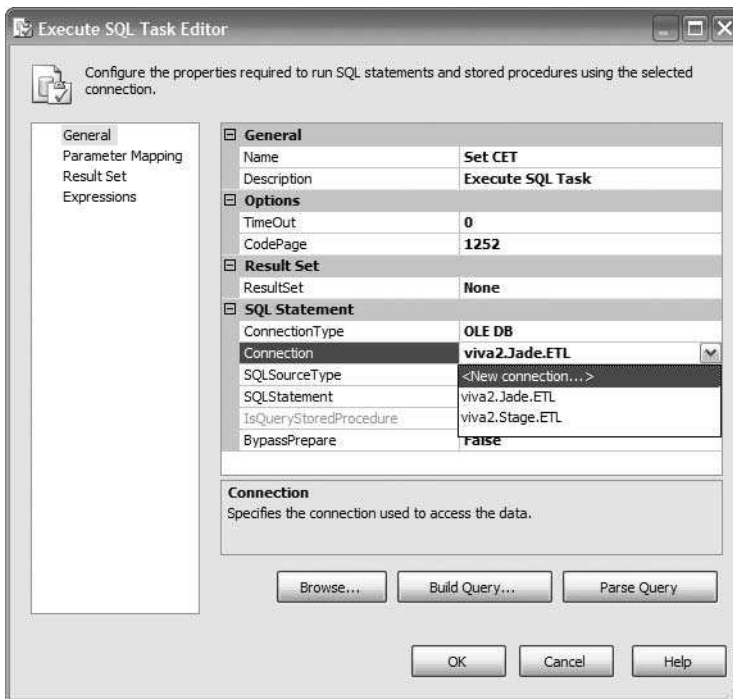


Figure 7-13. Configuring the connection for the Execute SQL Task

3. Choose the servername.Meta.ETL data connection, and click OK.
4. In the SQL Statement field, type *update data_flow set CET = getdate() where name = 'order_header'*. Click OK to close the Execute SQL Task Editor dialog box.

In steps 5 to 7 we are going to retrieve the last successful extraction time and the current extraction time from the data flow metadata table.

5. In the Toolbox, double-click the Execute SQL task again to create a new task in the design surface. Rename this box to **Get LSET**.
6. Double-click the box to edit it. Set the connection to `servername.Meta.ETL`. Set Result-Set to Single Row.
7. In the SQL Statement field, type **select LSET, CET from data_flow where name = 'order_header'**. Click Parse Query to verify.

In steps 8 to 12, we are going to store the query result in variables. They will be used later to limit the order header query.

8. In the left column, click Result Set. Click Add. Change NewResultName to **LSET**.
9. Click the Variable Name cell, expand the drop-down list, and choose **<New variable...>**. Leave the container set to Package, which means that the scope of the variable is the SSIS package, not just the Get LSET task, which means we can use the variable in other tasks.
10. Change the name to **dtLSET**. Leave Namespace set to User.
11. Change Value Type to **DateTime**. Set the value as **2007-10-01 00:00:00** (or any date you like), and click OK.
12. Click Add again to add another variable. Set Result Name as **CET** and Variable Name as a new variable called **dtCET** of type DateTime. The result will look like Figure 7-14.

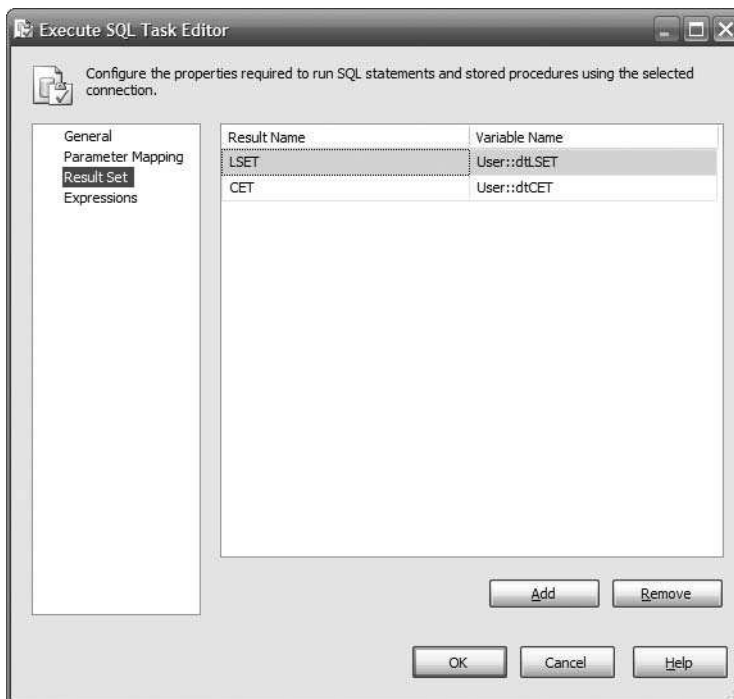


Figure 7-14. Storing the result set in variables

13. Click OK to close the Execute SQL Task Editor dialog box.

In steps 14 to 16 we will connect the tasks on the design surface and tidy up the layout.

14. Connect the green success arrow from the Set CET box to the Get LSET box. Rename the Data Flow task to **Extract Order Header**, and connect the green arrow from the Get LSET box to this box.
15. In the Toolbox, double-click the Execute SQL task to create a new box, and rename it to **Set LSET**. Connect the green arrow from Extract Order Header box to this Set LSET box.
16. Let's tidy up the layout a little bit. In the menu bar, select Format ☉ Auto Layout ☉ Diagram. The result looks like Figure 7-15.

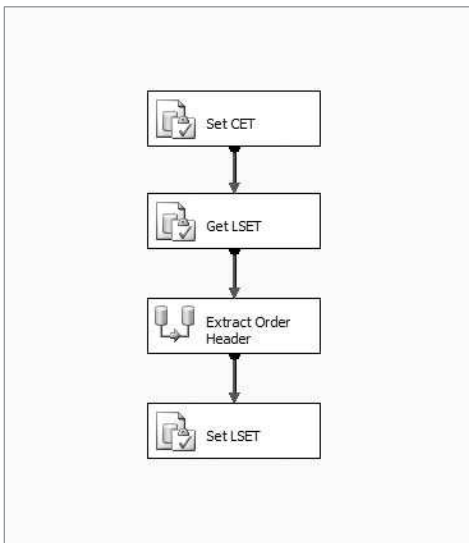


Figure 7-15. Arranging the layout after connecting the task boxes

In steps 17 to 20, we will modify to extract order header task to include the timestamp as parameters to limit the query.

17. Double-click the Extract Order Header box, and in the Data Flow design surface double-click the Jade Order Header box to edit it.
18. Update the SQL command text by changing all the fixed dates to question marks, as follows:

```

select * from order_header
where (created > ? and created <= ?)
or (last_updated > ? and last_updated <= ?)
  
```

19. Click the Parameter button, and set parameter 0 to User::dtLSET, parameter 1 to User::dtCET, parameter 2 to User::dtLSET, and parameter 3 to User::dtCET, as shown in Figure 7-16.



Figure 7-16. Making the date range in the Data Flow task dynamic

20. Click OK, and click OK again to close the windows and return to the data flow design surface.

In steps 21 to 22 we will modify the Set LSET task to update the timestamps stored in the *data_flow* table.

21. On top of the design surface, click the Control Flow tab. Double-click the Set LSET box, and set the connection to *servername.Meta.ETL*.
22. Set the SQL statement to *update data_flow set LSET = CET where name = 'order_header'*, and click OK. Save the package by pressing Ctrl+S.

In steps 23 to 25, we will prepare for the package execution by emptying the target table and setting the timestamp value.

23. Before we run the package, delete all rows from the *order_header* table in the stage, as follows: *truncate table stage.dbo.order_header*.
24. Set the CET column for *order_header* on the *data_flow* table in the metadata database to yesterday's date and the LSET column to the day before yesterday, as follows: *update meta.dbo.data_flow set LSET = getdate()-2, CET = getdate()-1 where name = 'order_header'*.
25. Verify the rows in Jade's *order_header* table that we are going to extract by using the following query:

```
select * from jade.dbo.order_header
where (created between getdate()-1 and getdate())
or (last_updated between getdate()-1 and getdate())
```

Note how many rows this query returns.

In steps 26 to 29, we will execute the package and check the execution results.

26. Go back to Business Intelligence Development Studio, and press F5 (or click the green triangle button in the toolbar) to run the package. All four boxes become yellow and then green one by one, as shown in Figure 7-17.

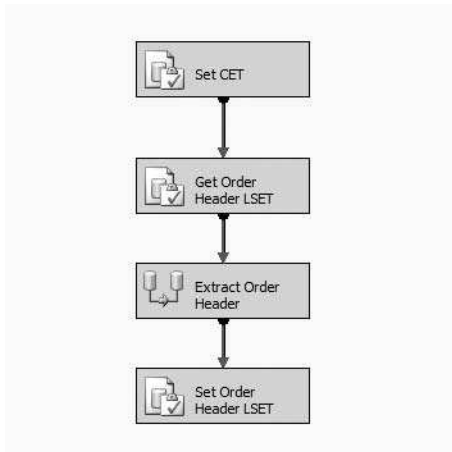


Figure 7-17. Running SSIS package that memorizes the last extraction time

27. If one of them becomes red, don't panic. Click the last tab (Progress) and see the error message. You can debug by right-clicking the boxes on the control flow and choosing Edit Breakpoints. The Set Breakpoints dialog box will pop up, as shown in Figure 7-18, and you can choose when you want the breakpoint to occur. You can also specify whether the breakpoint always happens or only at a certain count, that is, how many times the breakpoint is ignored before the execution is stopped.

You can also see the value of the variable by clicking the Watch 1 or Locals tabs in the bottom-left corner of the screen. Figure 7-19 shows that we can observe the value of the local variables when the execution is suspended at a breakpoint. For example, the value of user variable `dtCET_OH` is 06/10/2007 21:47:22 and the data type is DateTime.

28. Press Shift+F5 to stop the package running. Click the Execution Results tab, and verify that all tasks were executed successfully.
29. Open SQL Server Management Studio, and check that the LSET and CET columns for `order_header` in the `data_flow` table in the metadata database are updated correctly to current time. Also check that `order_header` in the stage database contains the correct number of rows from the Jade `order_header` table, as noted earlier.

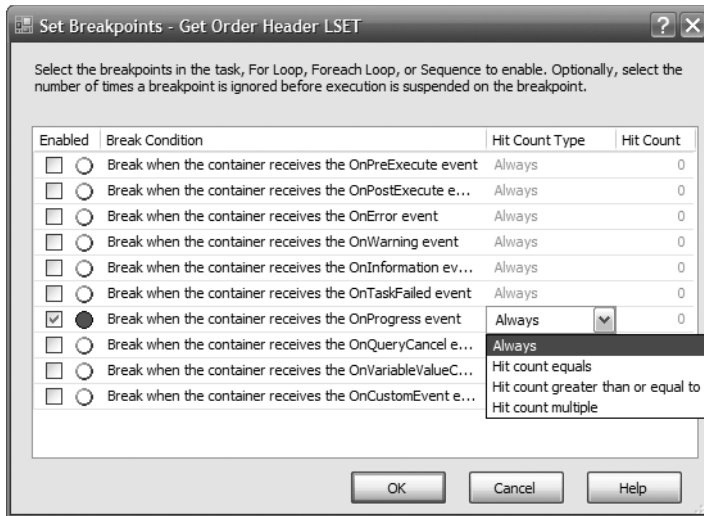


Figure 7-18. Setting up a breakpoint

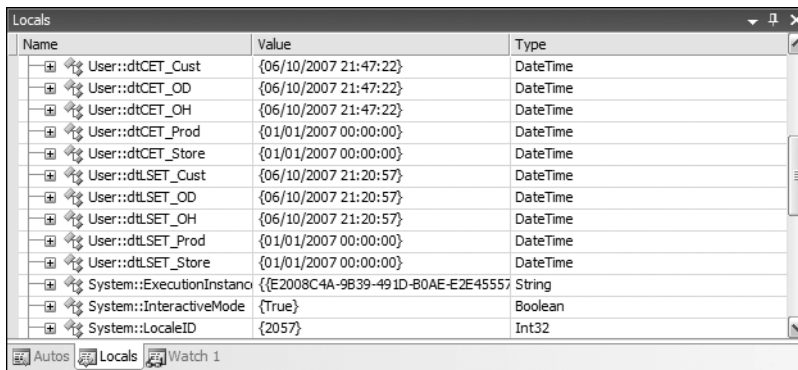


Figure 7-19. The Locals window showing the variable value and data type

30. At the moment, the name of the SSIS package in Solution Explorer is *Package.dtsx*. Rename it to *order_header.dtsx* by right-clicking it. When asked “Do you want to rename the package object as well,” click Yes. By default SSIS renames only the package file, but internally the object name is not changed (you can see the object name by clicking the View menu and choosing Properties Window). By answering yes, we rename the package object as well as the package file.
31. Click the File menu, and choose Save All to save the entire solution. Click the File menu, and choose Close Project to close the package.

With that, we’re finished going through the basics of extracting data incrementally from a relational database. Before we close this chapter, let’s do a data extract from a file because in real data warehouse projects this happens a lot.

Extracting from Files

Now that we have extracted data from a database, let's try to extract data from files. In the next few pages, we will extract data from a flat file. For the purpose of this exercise, we have ISO country data supplied to us in pipe-delimited files. We will import this file to the stage database. The file will be supplied once a week, from an external data provider.

In the following pages we will create an SSIS package to import the flat file into the stage database. After importing successfully to the stage, we will archive the file by moving it to another directory. We will accomplish this by doing the following steps:

1. Download the flat file containing the ISO country data.
2. Open the flat file to see the content.
3. Create the target table in the stage database.
4. Create a new SSIS package.
5. Create an Execute SQL task to truncate the target table.
6. Create a Data Flow task to load the flat file to the stage database.
7. Create a File System task to archive the file.
8. Execute the package.

The SSIS package that we are building looks like Figure 7-20. It truncates the target table, extracts data from the country flat file, and then archives the file.

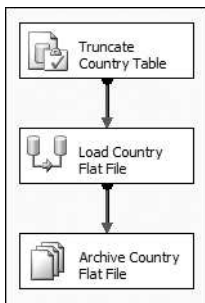


Figure 7-20. SSIS package to extract data from file

OK, so let's get started. First download the flat file from the book's web site at <http://www.apress.com/>. It is located in the folder `/data/external`. The file name is `country.dat` (8KB). Put this file in your local drive, for example: `c:\`.

Let's open the `country.dat` file using Notepad to see the content. It looks like this:



```
Code|Country
ad|Andorra
ae|United Arab Emirates
af|Afghanistan
ag|Antigua and Barbuda
ai|Anguilla
al|Albania
```

This consists of two columns, and it is pipe delimited. The first column is a two-character ISO country code, and the second column is the country name. The first line contains the column header: *Code* for the first column and *Country* for the second column.

Now that we know the content and format of the file, let's create the SSIS package to load this file into the stage. But before that, we need to create the stage table first. If we look at the data file, the longest country name is 34 characters (Saint Tome (Sao Tome) and Principe), so for this exercise we will set the width of the country name column to 50 characters. Let's open Management Studio and create the target table called *country* on the stage database as follows:

```
create table country
( country_code      char(2)
, country_name      varchar(50)
)
```

Now we can create the SSIS package:

1. Open Business Intelligence Development Studio, and open the Amadeus ETL project (select File  Open  Project, navigate to *Amadeus ETL.sln*, and double-click it).
2. In Solution Explorer, right-click SSIS Packages, and choose New SSIS Package. A blank design surface opens.
3. To be able to run the package many times, we need to truncate the target table before we load the data. Click the Control Flow tab, and double-click the Execute SQL task in the Toolbox.
4. Rename the box **Truncate Country Table**, and double-click the box to edit it. Set the connection to `servername.Stage.ETL`.
5. Set the SQL statement to *truncate table country*, and click OK. Click OK again to close the Execute SQL Task Editor dialog box.
6. Now let's create a Data Flow task to load the flat file to the stage database. Double-click Data Flow Task in the Toolbox, and rename the new box **Load Country Flat File**. Connect the green arrow from Truncate Country Table to Load Country Flat File.
7. Double-click Load Country Flat File to edit it. Let's add the country flat file as a source. Double-click Flat File Source in the Toolbox, and rename the new box **Country Flat File**.
8. Double-click the Country Flat File box, and click the New button.

9. The Flat File Connection Manager Editor dialog box opens, as shown in Figure 7-21. Set the name and description as Country Flat File, click Browse, and choose the *country.dat* file you downloaded earlier. Set the other properties as shown on Figure 7-21.

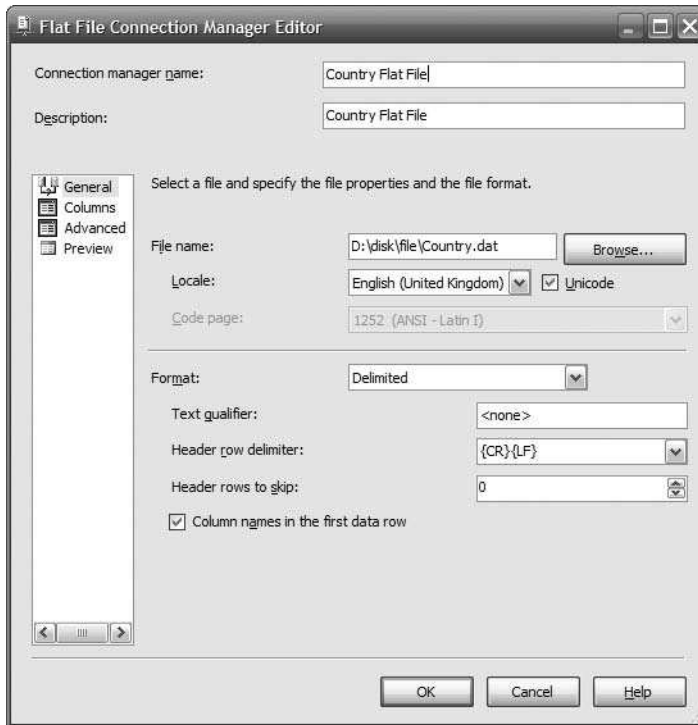


Figure 7-21. *Configuring the flat file connection properties*

10. In the pane on the left, click Columns, and specify the column delimiter as Vertical Bar {|}. Leave the row delimiter as {CR}{LF}.
11. Because the country code is two characters wide, we need to set the output column width accordingly. In the page on the left, click Advanced, and change the Output-ColumnWidth for the Code column from 50 to 2, as shown in Figure 7-22. Leave the Country column width as 50 because this column is 50 characters, as we discussed earlier.
12. Click OK to close the Flat File Connection Manager Editor and return to the Flat File Source Editor window. Click Preview to see the sample data, and then click Close. Click OK return to the design surface.
13. Now let's create a Data Conversion transformation to convert the flat file output, which is in Unicode, to suit the column on target table, which is an ANSI character string. Double-click the Data Conversion transformation in the Toolbox, and connect the green arrow from Country Flat File to Data Conversion.

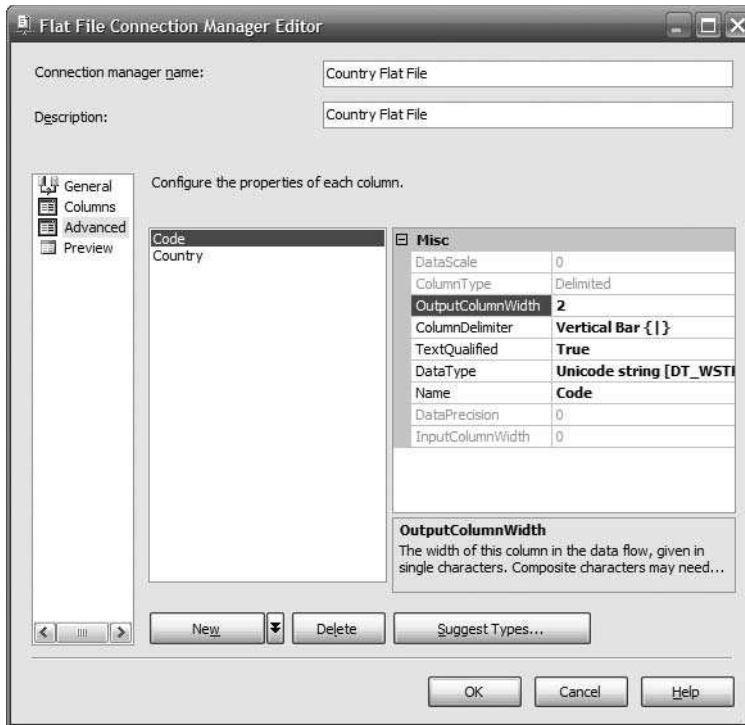


Figure 7-22. *Configuring the properties of each column*

14. Double-click Data Conversion to edit it, and check both the Code and Country columns under Available Input Columns. In the Data Type column, choose string [DT_STR] for both the Code and Country columns, as shown in Figure 7-23.

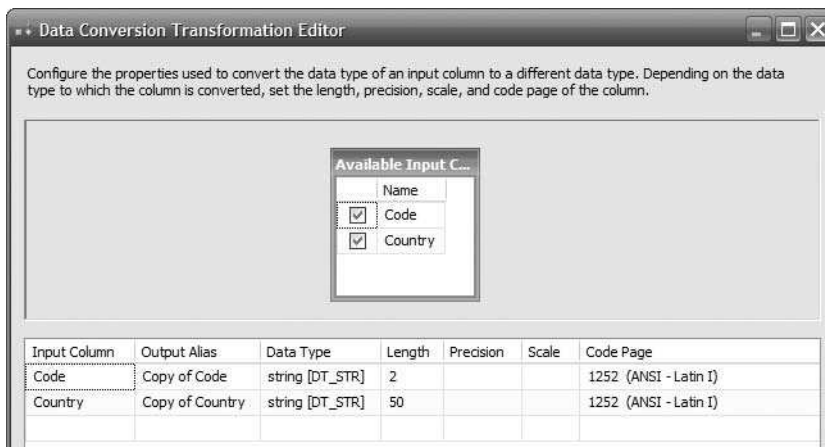


Figure 7-23. *Data Conversion Transformation Editor*

15. In the Toolbox, double-click SQL Server Destination, and rename the new box to **Country Stage**. Connect the green arrow on the Data Conversion box to the Country Stage box. Double-click the Country Stage box, and set the OLE DB connection manager to `servername.Stage.ETL`.
16. Set “Use table or view” to *country*, which we created earlier. In the panel on the left, click Mappings. Click and drag the Copy of Code column on the left to the *country_code* column on the right. Click and drag Copy of Country column on the left to the *country_name* column on the right, as shown in Figure 7-24.

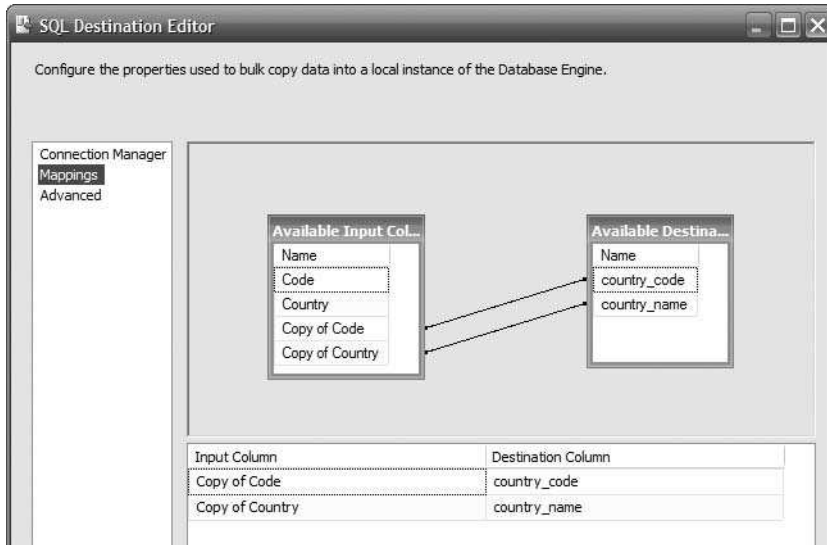


Figure 7-24. Mapping the source columns to the target columns

17. Click OK to return to the design surface. Click the Control Flow tab on the top of the design surface. We need to add one more task here, which is to archive the country flat file by moving it to another directory.
18. Double-click the File System task in the Toolbox, and rename the newly created box **Archive Country Flat File**.
19. Double-click the Archive Country Flat File box to edit it. Set Source Connection to point to the country flat file. Set Destination Connection to point to an empty folder. Set Operation to Move File, as shown on Figure 7-25.
20. Click OK to return to the design surface. Let's tidy up the layout a little bit. Click the Edit menu, and choose Select All. Click the Format menu, choose Auto Layout, and choose Diagram. The three boxes are now aligned and evenly spaced.
21. Click the File menu, and choose Save All to save the entire solution. Hit F5 to run the package. It should now run OK. Each box should turn yellow and then turn green, as shown in Figure 7-26.

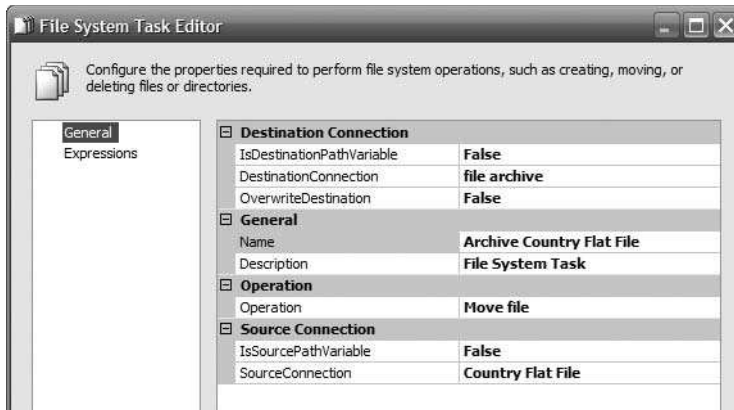


Figure 7-25. Configuring the File System task to archive the file

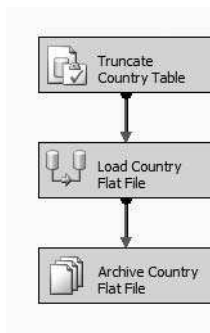


Figure 7-26. Importing country data from flat files into the stage

Even though we are not extracting incrementally here, we still have to add the metadata bit as we did previously when extracting the order header. This means we need to store the last successful run timestamp in the *data_flow* table. Recording the result of every single ETL data flow is important so that when the ETL fails and we want to restart from failure, we know where to restart from because we know which tasks were successful and which tasks failed to run. We do this by storing the execution result in the same *data_flow* metadata table as shown in Figure 7-27. (I will be discussing metadata in Chapter 10.)

key	name	status	LSET	CET
1	stg_order_header	2	2007-10-06 21:20:57.730	2007-10-06 21:47:22.750
2	stg_order_detail	2	2007-10-06 21:20:57.730	2007-10-06 21:47:22.750
3	stg_customer	2	2007-10-06 21:20:57.730	2007-10-06 21:47:22.750
4	stg_product	2	2007-10-06 21:20:57.730	2007-10-06 21:47:22.750
5	stg_product_status	1	2007-04-22 13:13:58.560	2007-04-22 13:13:58.560
6	stg_product_category	1	2007-04-22 13:13:58.560	2007-04-22 13:13:58.560
7	stg_product_type	1	2007-04-22 13:13:58.560	2007-04-22 13:13:58.560
27	stg_country	1	2007-10-07 16:43:19.733	2007-10-07 16:43:19.733

Figure 7-27. The *data_flow* metadata table stores execution status, the LSET, and the CET

Summary

This chapter started with the approaches and architectures for ETL systems, that is, ETL, ELT, using a dedicated ETL server or not, push vs. pull, choices of which server to place the ETL processes, and so on. Then it went through different kinds of source systems, such as relational databases, spreadsheets, flat files, XML files, web logs, database log files, images, web services, message queues, and e-mails. It also discussed how to extract incrementally a whole table every time using fixed ranges and how to detect data leaks.

Then I showed how to extract order header data from Jade to the stage using SSIS. We put some processes in place to memorize the last extract so we could extract incrementally. We also extracted the country flat file into the stage database using SSIS.

This chapter gave you a good foundation of data extraction principles, as well as a clear understanding of how to do it using SSIS. In the next chapter, we will bring the data that we have imported to the stage into the NDS and the DDS.