SAP Data Services 4.x Cookbook

Want to cost-effectively deliver trusted information to all of your crucial business functions? SAP Data Services delivers one enterprise-class solution for data integration, data quality, data profiling, and text data processing.

This book will lead you through the SAP Data Services environment to efficiently develop ETL processes. To begin with, you’ll learn to install, configure, and prepare the ETL development environment. You will become familiar with the concepts behind developing ETL processes with SAP Data Services. Starting from the smallest unit of work—the dataflow—you’ll be able to lead you to the highest organizational unit—the Data Services job—revealing advanced techniques of ETL design. It will then guide you through ETL-development patterns that enable the most effective performance when extracting, transforming, and loading data. You will also find out how to create validation functions and transforms.

Finally, the book will show you the benefits of data-quality management with the help of another SAP solution—Information Steward.

Who this book is written for

This book is for IT engineers who want to get familiar with the EIM solutions provided by SAP for ETL development and data quality management. The book requires familiarity with basic programming concepts and basic knowledge of the SQL language.

What you will learn from this book

- Install, configure, and administer SAP Data Services components
- Run through ETL design basics
- Maximize the performance of your ETL with advanced patterns for Data Services
- Extract methods from various databases and systems
- Become familiar with the transformation methods available in SAP Data Services
- Load methods into various databases and systems
- Code with the Data Services scripting language
- Validate and cleanse your data, applying the data quality methods of Information Steward

Delve into the SAP Data Services environment to efficiently prepare, implement, and develop ETL processes.
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 8 'Optimizing ETL performance'
- A synopsis of the book’s content
- More information on SAP Data Services 4.x Cookbook
About the Author

Ivan Shomnikov is an SAP analytics consultant specializing in the area of Extract, Transform, and Load (ETL). He has in-depth knowledge of the data warehouse life cycle processes (DWH design and ETL development) and extensive hands-on experience with both the SAP Enterprise Information Management (Data Services) technology stack and the SAP BusinessObjects reporting products stack (Web Intelligence, Designer, Dashboards).

Ivan has been involved in the implementation of complex BI solutions on the SAP BusinessObjects Enterprise platform in major New Zealand companies across different industries. He also has a strong background as an Oracle database administrator and developer.
SAP Data Services delivers an enterprise-class solution to build data integration processes as well as perform data quality and data profiling tasks, allowing you to govern your data in a highly-efficient way.

Some of the tasks that Data Services helps accomplish include: migration of the data between databases or applications, extracting data from various source systems into flat files, data cleansing, data transformation using either common database-like functions or complex custom-built functions that are created using an internal scripting language, and of course, loading data into your data warehouse or external systems. SAP Data Services has an intuitive user-friendly graphical interface, allowing you to access all its powerful Extract, Transform, and Load (ETL) capabilities from the single Designer tool. However, getting started with SAP Data Services can be difficult, especially for people who have little or no experience in ETL development. The goal of this book is to guide you through easy-to-understand examples of building your own ETL architecture. The book can also be used as a reference to perform specific tasks as it provides real-world examples of using the tool to solve data integration problems.

**What this book covers**

*Chapter 1, Introduction to ETL Development*, explains what Extract, Transform, and Load (ETL) processes are, and what role Data Services plays in ETL development. It includes the steps to configure the database environment used in recipes of the book.

*Chapter 2, Configuring the Data Services Environment*, explains how to install and configure all Data Services components and applications. It introduces the Data Services development GUI—the Designer tool—with the simple example of "Hello World" ETL code.

*Chapter 3, Data Services Basics – Data Types, Scripting Language, and Functions*, introduces the reader to Data Services internal scripting language. It explains various categories of functions that are available in Data Services, and gives the reader an example of how scripting language can be used to create custom functions.
Preface

Chapter 4, Dataflow – Extract, Transform, and Load, introduces the most important processing unit in Data Service, dataflow object, and the most useful types of transformations that can be performed inside a dataflow. It gives the reader examples of extracting data from source systems and loading data into target data structures.

Chapter 5, Workflow – Controlling Execution Order, introduces another Data Services object, workflow, which is used to group other workflows, dataflows, and script objects into execution units. It explains the conditional and loop structures available in Data Services.

Chapter 6, Job – Building the ETL Architecture, brings the reader to the job object level and reviews the steps used in the development process to make a successful and robust ETL solution. It covers the monitoring and debugging functionality available in Data Services and embedded audit features.

Chapter 7, Validating and Cleansing Data, introduces the concepts of validating methods, which can be applied to the data passing through the ETL processes in order to cleanse and conform it according to the defined Data Quality standards.

Chapter 8, Optimizing ETL Performance, is one of the first advanced chapters, which starts explaining complex ETL development techniques. This particular chapter helps the user understand how the existing processes can be optimized further in Data Services in order to make sure that they run quickly and efficiently, consuming as less computer resources as possible with the least amount of execution time.

Chapter 9, Advanced Design Techniques, guides the reader through advanced data transformation techniques. It introduces concepts of Change Data Capture methods that are available in Data Services, pivoting transformations, and automatic recovery concepts.

Chapter 10, Developing Real-time Jobs, introduces the concept of nested structures and the transforms that work with nested structures. It covers the mains aspects of how they can be created and used in Data Services real-time jobs. It also introduces new a Data Services component—Access Server.

Chapter 11, Working with SAP Applications, is dedicated to the topic of reading and loading data from SAP systems with the example of the SAP ERP system. It presents the real-life use case of loading data into the SAP ERP system module.

Chapter 12, Introduction to Information Steward, covers another SAP product, Information Steward, which accompanies Data Services and provides a comprehensive view of the organization's data, and helps validate and cleanse it by applying Data Quality methods.
If you tried all the previous recipes from the book, you can consider yourself familiar with the basic design techniques available in Data Services and can perform pretty much any ETL development task. Starting from this chapter, we will begin using advance development techniques available in Data Services. This particular chapter will help you to understand how the existing ETL processes can be optimized further to make sure that they run quickly and efficiently, consuming as less computer resources as possible with the least amount of execution time.

- Optimizing dataflow execution – push-down techniques
- Optimizing dataflow execution – the SQL transform
- Optimizing dataflow execution – the Data_Transfer transform
- Optimizing dataflow readers – lookup methods
- Optimizing dataflow loaders – bulk-loading methods
- Optimizing dataflow execution – performance options

**Introduction**

Data Services is a powerful development tool. It supports a lot of different source and target environments, all of which work differently with regard to loading and extracting data from them. This is why it is required of you, as an ETL developer, to be able to apply different design methods, depending on the requirements of your data migration processes and the environment that you are working with.
In this chapter, we will review the methods and techniques that you can use to develop data migration processes in order to perform transformations and migrate data from the source to target more effectively. The techniques described in this chapter are often considered as best practices, but do keep in mind that their usage has to be justified. They allow you to move and transform your data faster, consuming fewer processing resources on the ETL engine's server side.

**Optimizing dataflow execution – push-down techniques**

The Extract, Transform, and Load sequence can be modified to Extract, Load, and Transform by delegating the power of processing and transforming data to the database itself where the data is being loaded to.

We know that to apply transformation logic to a specific dataset we have to first extract it from the database, then pass it through transform objects, and finally load it back to the database. Data Services can (and most of the time, should, if possible) delegate some transformation logic to the database itself from which it performs the extract. The simplest example is when you are using multiple source tables in your dataflow joined with a single Query transform. Instead of extracting each table’s contents separately onto an ETL box by sending multiple `SELECT * FROM <table>` requests, Data Services can send the generated single `SELECT` statement with proper SQL join conditions defined in the Query transform’s `FROM` and `WHERE` tabs. As you can probably understand, this can be very efficient: instead of pulling millions of records into the ETL box, you might end up with getting only a few, depending on the nature of your Query joins. Sometimes this process shortens to a complete zero processing on the Data Services side. Then, Data Services does not even have to extract the data to perform transformations. What happens in this scenario is that Data Services simply sends the SQL statement instructions in the form of `INSERT INTO … SELECT` or `UPDATE … FROM` statements to a database when all the transformations are hardcoded in those SQL statements directly.

The scenarios when Data Services delegates the parts of or all the processing logic to the underlying database are called push-down operations.

In this recipe, we will take a look at different kinds of push-down operations, what rules you have to follow to make push-down work from your designed ETL processes, and what prevents push-downs from happening.

**Getting ready**

As a starting example, let's use the dataflow developed in the *Loading data from table to table – lookups and joins* recipe in Chapter 4, Dataflow – Extract, Transform, and Load. Please refer to this recipe to rebuild the dataflow if, for some reason, you do not have it in your local repository any more.
Push-down operations can be of two different types:

- **Partial push-downs**: A partial push-down is when Optimizer sends the `SELECT` query joining multiple source tables used in a dataflow or sends one `SELECT` statement to extract data from a particular table with mapping instructions and filtering conditions from the Query transform hardcoded in this `SELECT` statement.

- **Full push-downs**: A full push-down is when all dataflow logic is reformed by Optimizer in a single SQL statement and sent to the database. The most common statements generated in these cases are complex `INSERT/UPDATE` and `MERGE` statements, which include all source tables from the dataflow joined together and transformations in the form of database functions applied to the table columns.

**How to do it…**

1. To be able to see what SQL queries have been pushed down to the database, open the dataflow in the workspace window and select **Validation | Display Optimized SQL**.

2. The **Optimized SQL** window shows all queries generated by Data Services Optimizer and pushed down to the database level. In the following screenshot, you can see the `SELECT` query and part of the dataflow logic which this statement represents:

3. Let's try to push down logic from the rest of the Query transforms. Ideally, we would like to perform a full push-down to the database level.
4. The **Lookup_Phone** Query transform contains a function call which extracts the `PHONENUMBER` column from another table. This logic cannot be included as is because Optimizer cannot translate internal function calls into SQL construction, which could be included in the push-down statement.

5. Let’s temporarily remove this function call by specifying a hardcoded `NULL` value for the `PHONENUMBER` column. Just delete a function call and create a new output column instead of the `varchar(25)` data type.

6. Validate and save the dataflow and open the **Optimized SQL** window again to see the result of the changes. Straight away, you can see how logic from both the **Lookup_Phone** and **Distinct** Query transforms were included in the `SELECT` statement: the default `NULL` value for a new column and `DISTINCT` operator at the beginning of the statement:
7. What remains for the full push-down is the loading part when all transformations and selected datasets are inserted into the target table **PERSON_DETAILS**. The reason why this does not happen in this particular example is because the source tables and target tables reside in different datastores which connect to the different databases: OLTP (AdventureWorks_OLTP) and STAGE.

8. Substitute the **PERSON_DETAILS** target table from the DS_STAGE datastore with a new template table, **PERSON_DETAILS**, created in the DBO schema of OLTP.

9. As a change, you can see that Optimizer now fully transforms dataflow logic into a pushed-down SQL statement.
Optimizing ETL Performance

**How it works...**

Data Services Optimizer wants to perform push-down operations whenever possible. The most common reasons, as we demonstrated during the preceding steps, for push-down operations not working are as follows:

- **Functions**: When functions used in mappings cannot be converted by Optimizer to similar database functions in generated SQL statements. In our example, the `lookup_ext()` function prevents push-down from happening. One of the workarounds for this is to substitute the `lookup_ext()` function with an imported source table object joined to the main data set with the help of the Query transform (see the following screenshot):

- **Transform objects**: When transform objects used in a dataflow cannot be converted by Optimizer to relative SQL statements. Some transforms are simply not supported for push-down.

- **Automatic data type conversions**: These can sometimes prevent push-down from happening.
Different data sources: For push-down operations to work for the list of source or target objects, those objects must reside in the same database or must be imported into the same datastore. If they reside in different databases, dblink connectivity should be configured on the database level between those databases, and it should be enabled as a configuration option in the datastore object properties. All Data Services can do is send a SQL statement to one database source, so it is logical that if you want to join multiple tables from different databases in a single SQL statement, you have to make sure that connectivity is configured between databases, and then you can run SQL directly on the database level before even starting to develop the ETL code in Data Services.

What is also important to remember is that Data Services Optimizer capabilities depend on the type of underlying database that holds your source and target table objects. Of course, it has to be a database that supports the SQL standard language as Optimizer can send the push-down instructions only in the form of SQL statements.

Sometimes, you actually want to prevent push-downs from happening. This can be the case if:

- The database is busy to the extent that it would be quicker to do the processing on the ETL box side. This is a rare scenario, but still sometimes occurs in real life. If this is the case, you can use one of the methods we just discussed to artificially prevent the push-down from happening.
- You want to actually make rows go through the ETL box for auditing purposes or to apply special Data Services functions which do not exist at the database level. In these cases, the push-down will automatically be disabled and will not be used by Data Services anyway.

**Optimizing dataflow execution – the SQL transform**

Simply put, the SQL transform allows you to specify SQL statements directly inside the dataflow to extract source data instead of using imported source table objects. Technically, it has nothing to do with optimizing the performance of ETL as it is not a generally recommended practice to substitute the source table objects with the SQL transform containing hard-coded SELECT SQL statements.
Optimizing ETL Performance

How to do it...

1. Take the dataflow used in the previous recipe and select Validation | Display Optimized SQL... to see the query pushed down to the database level. We are going to use this query to configure our SQL transform object, which will substitute all source table objects on the left-hand side of the dataflow.

2. On the Optimized SQL window, click on Save As... to save this push-down query to the file.

3. Drag-and-drop the SQL transform from Local Object Library | Transforms | Platform into your dataflow.
4. Now you can remove all objects on the left-hand side of the dataflow prior to the **Lookup_Phone** Query transform.

5. Open the SQL transform for editing in a workspace window. Choose **OLTP** as a datastore and copy and paste the query saved previously from your file into the **SQL text** field. To complete the SQL transform configuration, create output schema fields of appropriate data types which match the fields returned by the **SELECT** statement.

6. Exit the SQL transform editor and link it to the next **Lookup_Phone** Query transform. Open **Lookup_Phone** and map the source columns to target.

7. Please note that the dataflow does not perform any native push-down queries anymore, and will give you the following warning message if you try to display optimized SQL:

   ![Warning Message]

   The optimized SQL cannot be generated for data flow <DF_SQL_Transform>. There are no SQL sources in this data flow. [801D-1170087]

8. Validate the job before executing it to make sure there are no errors.
How it works...

As you can see, the structure of the SQL transform is pretty simple. There are not many options available for configuration.

- **Datastore**: This option defines which database connection will be used to pass the `SELECT` query to
- **Database type**: This option pretty much duplicates the value defined for the specified datastore object
- **Cache**: This option defines whether the dataset returned by the query has to be cached on the ETL box
- **Array fetch size**: This option basically controls the amount of network traffic generated during dataset transfer from database to ETL box
- **Update schema**: This button allows you to quickly build the list of schema output columns from the SQL `SELECT` statement specified in the SQL text field

The two most common reasons why would you want to use SQL transform instead of defining source table objects are as follows:

- **Simplicity**: Sometimes, you do not care about anything else except getting things done as fast as possible. Sometimes you can get the extract requirements in the form of a `SELECT` statement, or if you want to use a tested `SELECT` query in your ETL code straight away.

- **To utilize database functionality which does not exist in Data Services**: This is usually a poor excuse as experienced ETL developers can do pretty much anything with standard Data Services objects. However, some databases can have internal non-standard SQL functions implemented which can perform complex transformations. For example, in Netezza you can have functions written in C++, which can be utilized in standard SQL statements and, most importantly, will be using the massive-parallel processing functionality of the Netezza engine. Of course, Data Services Optimizer is not aware of these functions and the only way to use them is to run direct `SELECT` SQL statements against the database. If you want to call a SQL statement like this from Data Services, the most convenient way to do it from within a dataflow is to use the SQL transform object inside the dataflow.

- **Performance reasons**: Once in a while, you can get a set of source tables joined to each in a dataflow for which Optimizer—for some reason or other—does not perform a push-down operation. You are very restricted in the ways you can create and utilize database objects in this particular database environment. In such cases, using a hard-coded `SELECT` SQL statement can help you to maintain an adequate level of ETL performance.
As a general practice, I would recommend that you avoid SQL transforms as much as possible. They can come in handy sometimes, but when using them, you not only lose the advantage of utilizing Data Services, the Information Steward reporting functionality, and ability to perform auditing operations, you also potentially create big problems for yourself in terms of ETL development process. Tables used in the SELECT statements cannot be traced with the View were used feature. They can be missing from your datastores, which means you do not have a comprehensive view of your environment and underlying database objects utilized by hiding source database tables inside the ETL code rather than having them on display in Local Object Library.

This obviously makes ETL code harder to maintain and support. Not to mention that migration to another database becomes a problem as you would most likely have to rewrite all the queries used in your SQL transforms.

The SQL transform prevents the full push-down from happening, so be careful. Only the SELECT query inside the SQL transform is pushed down to database level. The rest of the dataflow logic will be executed on the ETL box even if the full push-down was working before, when you had source table objects instead of the SQL transform.

In other words, the result data set for the SQL transform always transferred to the ETL box. That can affect the decisions around ETL design. From the performance perspective, it is preferable to spend more time building a dataflow based on the source object tables but for which Data Services performs the full push-down (producing the INSERT INTO ... SELECT statement), rather than quickly building the dataflow which will transfer datasets back and forth to the database, increasing the load time significantly.

### Optimizing dataflow execution – the Data_Transfer transform

The transform object Data_Transfer is a pure optimization tool helping you to push down resource-consuming operations and transformations like JOIN and GROUP BY to the database level.

**Getting ready**

1. Take the dataflow from the Loading data from a flat file recipe in Chapter 4, Dataflow – Extract, Transform, and Load. This dataflow loads the Friends_*.txt file into a STAGE.FRIENDS table.
2. Modify the Friends_30052015.txt file and remove all lines except the ones about Jane and Dave.
3. In the dataflow, add another source table, `OLTP.PERSON`, and join it to a source file object in the Query transform by the first-name field. Propagate the `PERSONTYPE` and `LASTNAME` columns from the source `OLTP.PERSON` table into the output Query transform schema, as shown here:

![Diagram showing dataflow and Query transform]

**How to do it...**

Our goal will be to configure this new dataflow to push down the insert of the joined dataset of data coming from the file and data coming from the `OLTP.PERSON` table to a database level.

By checking the Optimized SQL window, you will see that the only query sent to a database from this dataflow is the `SELECT` statement pulling all records from the database table `OLTP.PERSON` to the ETL box, where Data Services will perform an in-memory join of this data with data coming from the file. It’s easy to see that this type of processing may be extremely inefficient if the `PERSON` table has millions of records and the `FRIENDS` table has only a couple of them. That is why we do not want to pull all records from the `PERSON` table for the join and want to push down this join to the database level.
Looking at the dataflow, we already know that for the logic to be pushed down, the database should be aware of all the source data sets and should be able to access them by running a single SQL statement. The **Data_Transfer** transform will help us to make sure that the Friends file is presented to a database as a table. Follow these steps to see how it can be done:

1. Add the **Data_Transfer** object from Local Object Library | Transforms | Data Integrator into your dataflow, putting it between the source file object and the Query transform.

2. Edit the Data_Transfer object by opening it in a workspace window. Set Transfer type to **Table** and specify the new transfer table in the Table options section with STAGE.DBO.FRIENDS_FILE.

![Data_Transfer object settings](image)
3. Close the Data Transfer transform editor and select Validation | Display Optimized SQL... to see the queries pushed down to a database. You can see that there are now two SELECT statements generated to pull data from the OLTP.PERSON and STAGE.FRIENDS_FILE tables.

- The join between these two datasets happens on the ETL box. Then the merged data set is sent back to the database to be inserted into the DS_STAGE.FRIENDS table.

4. Add another Data_Transfer transformation between the source table PERSON and the Query transform. In the Data_Transfer configuration window, set Transfer type to Table and specify DS_STAGE.DBO.DT_PERSON as the data transfer table.

5. Validate and save the dataflow and display the Optimized SQL window.
Now you can see that we successfully implemented a **full push-down** of dataflow logic, inserting merged data from two source objects (one of which is a flat file) into a staging table. In the preceding screenshot, logic in the section marked as red is represented by a SQL statement `INSERT` pushed down to the database level.

**How it works...**

Under the hood, **Data_Transfer** transform creates a subprocess that transfers the data to the specified location (file or table). Simply put, **Data_Transfer** is a target dataflow object in the middle of a dataflow. It has a lot of options similar to what other target table objects have; in other words, you can set up a bulk-loading mechanism, run **Pre-Load Commands** and **Post-Load Commands**, and so on.

The reason why I called **Data_Transfer** a pure optimization tool is because you can redesign any dataflow to do the same thing that **Data_Transfer** does without using it. All you have to do is to simply split your dataflow in two (or three, for the dataflow in our example). Instead of forwarding your data into a **Data_Transfer** transform, you forward it to a normal target object and then, in the next dataflow, you use this object as a source.

What **Data_Transfer** still does, which cannot be done easily when you are splitting dataflows, is automatically clean up temporary data transfer tables.

It is critical to understand how push-down mechanisms work in Data Services to be able to effectively use the **Data_Transfer** transform. Putting it to use at the wrong place in a dataflow can decrease performance drastically.
**Why we used a second Data_Transfer transform object**

Our goal was to modify the dataflow in such a way as to get a full push-down SQL statement to be generated: `INSERT INTO STAGE.FRIENDS SELECT <joined PERSON and FRIENDS data sets>`.

As we remember from the previous recipe, there could be multiple reasons why full push-down does not work. One of these reasons, which is causing trouble in our current example, is that the PERSON table resides in a different database, while our data transfer table, FRIENDS_FILE, and target table, FRIENDS, reside in the same STAGE database.

To make the full push-down work, we had to use a second Data_Transfer transform object to transfer data from the OLTP.PERSON table into a temporary table located in a STAGE database.

**When to use Data_Transfer transform**

Whenever you encounter a situation where a dataflow has to perform a very "heavy" transformation (say the GROUP BY operation, for example) or join two very big data sets and this operation is happening on an ETL box. In these cases, it is much quicker to transfer the required data sets to the database level so that the resource-intensive operation can be completed there by the database.

**There's more...**

One of the good examples of a use case for the Data_Transfer transform is when you have to perform the GROUP BY operation in a Query transform right before inserting data into a target table object. By placing Data_Transfer right before the Query transform at the end of the dataflow, you can quickly insert the dataset processed by dataflow logic before the Query transform with the GROUP BY operation and then push down the INSERT and GROUP BY operations in a single SQL statement to a database level.

When you perform the transformations on datasets which include millions of records, using the Data_Transfer transform can save you minutes, and sometimes hours, depending on your environment and the number of processed records.

**Optimizing dataflow readers – lookup methods**

There are different ways in which to perform the lookup of a record from another table in Data Services. The three most popular ones are: a table join with a Query transform, using the lookup_ext() function, and using the sql() function.

In this recipe, we will take a look at all these methods and discuss how they affect the performance of ETL code execution and their impact on a database used to source data from.
Getting ready

We will be using the same dataflow as in the first recipe, the one which populates the PERSON DETAILS stage table from multiple OLTP tables.

How to do it...

We will perform a lookup for the PHONENUMBER column of a person from the OLTP table PERSONPHONE in three different ways.

Lookup with the Query transform join

1. Import the lookup table into a datastore and add the table object as a source in the dataflow where you need to perform the lookup.

2. Use the Query transform to join your main dataset with the lookup table using the BUSINESSENTITYID reference key column, which resides in both tables.
Optimizing ETL Performance

Lookup with the lookup_ext() function

1. Remove the PERSONPHONE source table from your dataflow and clear out the join conditions in the Lookup_Phone Query transform.

2. As you have seen in the recipes in previous chapters, the lookup_ext() function can be executed as a function call in the Query transform output columns list. The other option is to call the lookup_ext() function in the column mapping section. For example, say that we want to put an extra condition on when we want to perform a lookup for specific value.

Instead of creating a new function call for looking up the PHONENUMBER column for all migrated records, let's put in the condition that we want to execute the lookup_ext() function only when the row has nonempty ADDRESSLINE1, CITY, and COUNTRY columns; otherwise, we want to use the default value UNKNOWN LOCATION.

3. Insert the following lines in the Mapping section of the PHONENUMBER column inside the Lookup_Phone Query transform:

   ifthenelse(
       (Get_Country.ADDRESSLINE1 IS NULL) OR
       (Get_Country.CITY IS NULL) OR
       (Get_Country.COUNTRY IS NULL), 'UNKNOWN LOCATION',
       lookup_ext()
   )

4. Now double-click on the lookup_ext() text to highlight only the lookup_ext function and right-click on the highlighted area for the context menu.

5. From this context menu, select Modify Function Call to open the Lookup_ext parameter configuration window. Configure it to perform a lookup for a PHONENUMBER field value from the PERSONPHONE table.
After closing the function configuration window, you can see the full code generated by Data Services for the `lookup_ext()` function in the **Mapping** section.

When selecting the output field, you can see all source fields used in its **Mapping** section highlighted in the **Schema In** section on the left-hand side.

**Lookup with the sql() function**

1. Open the **Lookup_Phone** Query transform for editing in the workspace and clear out all code from the **PHONENUMBER** mapping section.
2. Put the following code in the **Mapping** section:

   ```sql
   sql('OLTP','select PHONENUMBER from Person.PERSONPHONE where BUSINESSENTITYID = [Get_Country.BUSINESSENTITYID]
   ```

**How it works...**

**Query transform joins**

The advantages of this method are:

- **Code readability**: It is very clear which source tables are used in transformation when you open the dataflow in a workspace.

- **Push-down lookup to the database level**: This can be achieved by including a lookup table in the same SELECT statement. Yes, as soon as you have placed the source table object in the dataflow and joined it properly with other data sources using the Query transform, there is a chance that it will be pushed down as a single SQL SELECT statement, allowing the joining of source tables at the database level.
DS metadata report functionality and impact analysis: The main disadvantage of this method comes naturally from its advantage. If a record from the main dataset references multiple records in the lookup table by the key column used, the output data set will include multiple records with all these values. That is how standard SQL query joins work, and the Data Services Query transform works in the same way. This could potentially lead to duplicated records inserted into a target table (duplicated by key columns but with different values in the lookup field, for example).

lookup_ext()  
The opposite of a Query transform, this function hides the source lookup table object from the developer and from some of the Data Services reporting functionality. As you have seen, it can be executed as a function call or used in the mapping logic for a specific column.

This function’s main advantage is that it will always return a single value from the lookup table. You can even specify the return policy, which will be used to determine the single value to return—\text{MAX} or \text{MIN}—with the ability to order the lookup table dataset by any column.

sql()  
Similar to the \text{lookup\_ext()} function in the presented example, it is rarely used that way as \text{lookup\_ext()} fetches rows from the lookup table more efficiently, if all you want to do is to extract values from the lookup table referencing key columns.

At the same time, the \text{sql()} function makes possible the implementation of very complex and flexible solutions as it allows you to pass any SQL statement that can be executed on the database side. This can be the execution of stored procedures, the generation of the sequence numbers, running analytical queries, and so on.

As a general rule, though, the usage of the \text{sql()} function in the dataflow column mappings is not recommended. The main reason for this is performance, as you will see further on. Data Services has a rich set of instruments to perform the same task but with a proper set of objects and ETL code design.

Performance review  
Let’s quickly review dataflow execution times for each of the explained methods.

- The first method: The lookup with the Query transform took 6.4 seconds.

<table>
<thead>
<tr>
<th>Path Name</th>
<th>State</th>
<th>Row Count</th>
<th>Elapsed Time (sec)</th>
<th>Absolute Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCT_TABLE_TO_TABLE_Distinct_PERSON_DETAILS</td>
<td>STOP</td>
<td>58798</td>
<td>0.000</td>
<td>6.410</td>
</tr>
<tr>
<td>DCT_TABLE_TO_TABLE_Distinct</td>
<td>STOP</td>
<td>58798</td>
<td>0.000</td>
<td>6.410</td>
</tr>
</tbody>
</table>
The second method: The lookup with the `lookup_ext()` function took 6.6 seconds.

<table>
<thead>
<tr>
<th>Path Name</th>
<th>State</th>
<th>Row Count</th>
<th>Elapsed Time (sec)</th>
<th>Absolute Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>`IF_TABLE_TO_TABLE</td>
<td>Distinct_PERSON_DETAILS`</td>
<td>STOP</td>
<td>18708</td>
<td>0.512</td>
</tr>
<tr>
<td>`IF_TABLE_TO_TABLE</td>
<td>Round_Split`</td>
<td>STOP</td>
<td>18708</td>
<td>0.308</td>
</tr>
<tr>
<td>`IF_TABLE_TO_TABLE</td>
<td>Distinct_Mapping2`</td>
<td>STOP</td>
<td>0400</td>
<td>0.300</td>
</tr>
<tr>
<td>`IF_TABLE_TO_TABLE</td>
<td>Distinct`</td>
<td>STOP</td>
<td>18708</td>
<td>0.147</td>
</tr>
<tr>
<td>`IF_TABLE_TO_TABLE</td>
<td>Distinct_Mapping3`</td>
<td>STOP</td>
<td>0508</td>
<td>0.300</td>
</tr>
<tr>
<td>`IF_TABLE_TO_TABLE</td>
<td>PERSON+ADDRESS+IDENTITY+...`</td>
<td>STOP</td>
<td>18708</td>
<td>0.308</td>
</tr>
<tr>
<td>`IF_TABLE_TO_TABLE</td>
<td>Merge`</td>
<td>STOP</td>
<td>18708</td>
<td>0.147</td>
</tr>
</tbody>
</table>

The third method: This used the `sql()` function and took 73.3 seconds.

<table>
<thead>
<tr>
<th>Path Name</th>
<th>State</th>
<th>Row Count</th>
<th>Elapsed Time (sec)</th>
<th>Absolute Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>`IF_TABLE_TO_TABLE</td>
<td>Distinct_PERSON_DETAILS`</td>
<td>STOP</td>
<td>18708</td>
<td>1.964</td>
</tr>
<tr>
<td>`IF_TABLE_TO_TABLE</td>
<td>Get_Country`</td>
<td>STOP</td>
<td>18708</td>
<td>63.742</td>
</tr>
<tr>
<td>`IF_TABLE_TO_TABLE</td>
<td>Distinct`</td>
<td>STOP</td>
<td>18708</td>
<td>62.798</td>
</tr>
<tr>
<td>`IF_TABLE_TO_TABLE</td>
<td>Lookup_Phone`</td>
<td>STOP</td>
<td>18708</td>
<td>56.193</td>
</tr>
</tbody>
</table>

The first two methods look like the methods with similar effectiveness, but that is only because the number of rows and the size of the dataset used is very small. The `lookup_ext()` function allows the usage of the different cache methods for the lookup dataset, which makes it possible to tune and configure it depending on the nature of your main data and that of the lookup data. It can also be executed as a separate OS process, increasing the effectiveness of fetching the lookup data from the database.

The third figure for the `sql()` function, on the contrary, shows the perfect example of extremely poor performance when the `sql()` function is used in the column mappings.

Optimizing dataflow loaders – bulk-loading methods

By default, all records inside a dataflow coming to a target table object are sent as separate `INSERT` commands to a target table at the database level. If millions of records pass the dataflow and transformation happens on the ETL box without push-downs, the performance of sending millions of `INSERT` commands over the network back to a database for insertion could be extremely slow. That is why it is possible to configure the alternative load methods on the target table object inside a dataflow. These types of loads are called bulk-load loads. Bulk-load methods are different in nature, but all of them have the main principle and achieve the same goal—they avoid the execution of millions of `INSERT` statements for each migrated record, providing alternative ways of inserting data.
Optimizing ETL Performance

Bulk-load methods executed by Data Services for inserting data into a target table are completely dependent on the type of target database. For example, Oracle Database Data Services can implement bulk-loading through the files or through the Oracle API.

Bulk-loading mechanisms for inserting data into Netezza or Teradata are completely different. You will notice this straightaway if you create different datastores connecting to different types of databases and compare the target table Bulk Loader Options tab to the target table object from each of these datastores.

For detailed information about each bulk-load method available for each database, please refer to official SAP documentation.

How to do it...

To see the difference between loading data in normal mode—row by row—and bulk loading, we have to generate quite a significant number of rows. To do this, take the dataflow from a previous recipe, Optimizing dataflow execution – the SQL transform, and replicate it to create another copy for using in this recipe. Name it DF_Bulk_Load.

Open the dataflow in the workspace window for editing.

1. Add a new Row_Generation transform from Local Object Library | Transforms | Platform as a source object and configure it to generate 50 rows, starting with row number 1.
2. The Row_Generation transform is used to multiply the number of rows currently being transformed by the dataflow logic. Previously, the number of rows returned by the Person OLTP SQL transform was approximately 19,000. By performing a Cartesian join of these records to 50 artificially generated records, we can get almost 1 million records inserted in a target PERSON_DETAILS table. To implement Cartesian join, use the Query transform but without specifying any join conditions and leaving the section empty.

<table>
<thead>
<tr>
<th>Input schema(s)</th>
<th>SELECT FROM WHERE GROUP BY ORDER BY Advanced Find</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generate 50_rows</td>
<td>Count 0 Automatic</td>
</tr>
<tr>
<td>Distinct</td>
<td>Count 0 Automatic</td>
</tr>
</tbody>
</table>

Join pairs:

<table>
<thead>
<tr>
<th>Left</th>
<th>Join Type</th>
<th>Right</th>
<th>Join Condition</th>
</tr>
</thead>
</table>
3. Your dataflow should look like this:

![Dataflow Diagram]

4. To test the current dataflow execution time, save and run the job, which includes this dataflow. Your target table’s **Bulk Loader Option** tab should be disabled, and on the **Options** tab, the **Delete data from table before loading** flag should be selected.

5. The execution time of the dataflow is 49 seconds, and as you can see, it took 42 seconds for Data Services to insert 9,39,900 records into the target table.

6. To enable bulk loading, open the target table configuration in the workspace for editing, go to the **Bulk Loader Options** tab, and check **Bulk load**. After that, set **Mode** to **truncate** and leave other options at their default values.
Optimizing ETL Performance

7. Save and execute the job again.

8. The following screenshot shows that total dataflow execution time was 27 seconds, and it took 20 seconds for Data Services to load the same number of records. That is two times faster than loading records in normal mode into the SQL Server database. Your time could be slightly different depending on the hardware you are using for your Data Services and database environments.

<table>
<thead>
<tr>
<th>Path Name</th>
<th>State</th>
<th>Row Count</th>
<th>Elapsed Time (sec)</th>
<th>Absolute Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DF_SQL_Transform/Round_Rollen_Split</td>
<td>STOP</td>
<td>50</td>
<td>0.000</td>
<td>27.446</td>
</tr>
<tr>
<td>DF_SQL_Transform/Cartehan</td>
<td>STOP</td>
<td>18798</td>
<td>0.044</td>
<td>27.472</td>
</tr>
<tr>
<td>DF_SQL_Transform/Round_Rollen_Split[1]</td>
<td>STOP</td>
<td>18798</td>
<td>0.500</td>
<td>27.463</td>
</tr>
<tr>
<td>DF_SQL_Transform/District</td>
<td>STOP</td>
<td>18798</td>
<td>0.309</td>
<td>27.450</td>
</tr>
<tr>
<td>DF_SQL_Transform/Lookup_Phone</td>
<td>STOP</td>
<td>9400</td>
<td>0.352</td>
<td>27.455</td>
</tr>
<tr>
<td>DF_SQL_Transform/Cartehan_2</td>
<td>STOP</td>
<td>90</td>
<td>23.114</td>
<td>27.455</td>
</tr>
<tr>
<td>DF_SQL_Transform/Cartehan_2 PERSON DETAILS</td>
<td>STOP</td>
<td>902000</td>
<td>26.291</td>
<td>27.478</td>
</tr>
<tr>
<td>DF_SQL_Transform/Cartehan_2 Mapping1</td>
<td>STOP</td>
<td>93968</td>
<td>0.100</td>
<td>27.462</td>
</tr>
<tr>
<td>DF_SQL_Transform/Range1</td>
<td>STOP</td>
<td>18798</td>
<td>0.100</td>
<td>27.462</td>
</tr>
<tr>
<td>DF_SQL_Transform/Cartehan_2 Range1 Reader</td>
<td>STOP</td>
<td>93968</td>
<td>0.300</td>
<td>27.461</td>
</tr>
<tr>
<td>DF_SQL_Transform/Cartehan_2-BIN0</td>
<td>STOP</td>
<td>93968</td>
<td>0.599</td>
<td>27.451</td>
</tr>
<tr>
<td>DF_SQL_Transform/Range2</td>
<td>STOP</td>
<td>93960</td>
<td>21.135</td>
<td>27.465</td>
</tr>
<tr>
<td>DF_SQL_Transform/Cartehan_2-BIN2</td>
<td>STO</td>
<td>0</td>
<td>0.002</td>
<td>27.463</td>
</tr>
<tr>
<td>DF_SQL_Transform/Range</td>
<td>STOP</td>
<td>90</td>
<td>23.114</td>
<td>27.455</td>
</tr>
<tr>
<td>DF_SQL_Transform/Round_Rollen_Split[1]</td>
<td>STOP</td>
<td>50</td>
<td>0.249</td>
<td>27.480</td>
</tr>
</tbody>
</table>

How it works...

Availability of the bulk-load methods is totally dependent on which database you use as a target. Data Services does not perform any magic; it simply utilizes bulk-loading methods available in a database.

These methods are different for different databases, but the principle of bulk loading is usually as follows: Data Services sends the rows to the database host as quickly as possible, writing them into a local file. Then, Data Services uses the external table mechanism available in the database to present the file as a relational table. Finally, it executes a few UPDATE/INSERT commands to query this external table and insert data into a target table specified as a target object in a Data Services dataflow.

To run one INSERT ... SELECT FROM command is much faster than to execute 1 million INSERT commands.

Some databases perform these small insert operations quite effectively, while for others this could be a really big problem. In almost all cases, if we talk about a significant number of records, the bulk-loading method will always be the quicker way to insert data.
When to enable bulk loading?
You have probably noticed that as soon as you enable bulk loading in target table configuration, the Options tab becomes grayed out. Unfortunately, by enabling bulk loading, you lose all extra functionality available for loading data, such as autocorrect load, for example. This happens because of the nature of the bulk-load operation. Data Services simply passes the data to the database for insertion and cannot perform extra comparison operations, which are available for row-by-row inserts.

The other reason for not using bulk loading is that enabled bulk loading prevents full push-downs from occurring. Of course, in most of the cases push-down is the best possible option in terms of execution performance, so you would never think about enabling bulk loading if you have full push-down working. For partial push-downs, when you push down only SELECT queries to get data onto the ETL box for transformation, bulk loading is perfectly valid. You still want to send records back to the database for insertion and want to do it as quickly as possible.

Most of the time, bulk loading does a perfect job when you are passing a big number of rows for insertion from the ETL box and do not utilize any extra loading options available in Data Services.

The best advice in terms of making decisions to enable or not enable bulk loading on your target table is to experiment and try different ways of inserting data. This is a decision which should take into account all parameters, such as environment configuration, workload on a Data Services ETL box, workload on a database, and of course, the number of rows to be inserted into a target table.

Optimizing dataflow execution – performance options

We will review a few extra options available for different transforms and objects in Data Services which affect performance and, sometimes, the way ETL processes and transforms data.

Getting ready

For this recipe, use the dataflow from the recipe Optimizing dataflow readers – lookup methods in this chapter. Please refer to this recipe if you need to create or rebuild this dataflow.
How to do it...

Data Services performance-related configuration options can be put under the following categories:

- Dataflow performance options
- Source table performance options
- Query transform performance options
- Lookup functions performance options
- Target table performance options

In the following sections, we will review and explain all of them in details.

Dataflow performance options

To access dataflow performance options, right-click on a dataflow object and select Properties from the context menu.

The **Degree of parallelism** option replicates transform processes inside the dataflow according to the number specified. Data Services creates separate subdataflow processes and executes them in parallel. At the points in the dataflow where the processing cannot be parallelized, data is merged back together from different subdataflow processes in the main dataflow process. If the source table used in the dataflow is partitioned and the value in the **Degree of parallelism** option is higher than 1, Data Services can use multiple reader processes to read the data from the same table. Each reader reads data from corresponding partitions. Then, data is merged or continued to be processed in parallel if the next transform object allows parallelization.

For detailed information on how the **Degree of Parallelism** option works, please refer to the official documentation, *SAP Data Services: Performance Optimization Guide*. You should be very careful with this parameter. The usage and value of **Degree of parallelism** should depend on the complexity of the dataflow and on the resources available on your Data Services ETL server, such as the number of CPUs and amount of memory used.
If the **Use database links** option is configured on both database and Data Services datastore levels, database links can help to produce push-down operations. Use this option to enable or disable database links usage inside a dataflow.

**Cache type** defines which type of cache will be used inside a dataflow for caching datasets. A **Pageable** cache is stored on the ETL server's physical disk and **In-Memory** keeps the cached dataset in memory. If the dataflow processes very large datasets, it is recommended that you use a pageable cache to not run out of memory.

**Source table performance options**

Open your dataflow in the workspace and double-click on any source table object to open the table configuration window.

<table>
<thead>
<tr>
<th>Source table performance options</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATEPROVINCE</td>
<td></td>
</tr>
<tr>
<td>STATEPROVINCENAME</td>
<td>int</td>
</tr>
<tr>
<td>COUNTRYREGIONCODE</td>
<td>varchar(3)</td>
</tr>
<tr>
<td>ISONLYSTATEPROVINCEFLAG</td>
<td>int</td>
</tr>
<tr>
<td>NAME</td>
<td>varchar(50)</td>
</tr>
<tr>
<td>TERRITORYID</td>
<td>int</td>
</tr>
<tr>
<td>ROWGUID</td>
<td>varchar(36)</td>
</tr>
<tr>
<td>MODIFIEDDATE</td>
<td>datetime</td>
</tr>
</tbody>
</table>

**Array fetch size** allows you to optimize the number of requests Data Services sends to fetch the source data set onto the ETL box. The higher the number used, the fewer the requests that Data Services has to send to fetch the data. This setting should be dependent on the speed of your network. The faster your network is, the higher the number you can specify to move the data in bigger chunks. By decreasing the number of requests, you can potentially also decrease the CPU usage consumption on your ETL box.
Optimizing ETL Performance

Join rank specifies the "weight" of the table used in Query transforms when you join multiple tables. The higher the rank, the earlier the table will be joined to the other tables. If you have ever optimized SQL statements, you know that specifying big tables in the join conditions earlier can potentially decrease the execution time. This is because the number of records after the first join pair can be decreased dramatically through inner joins, for example. This makes the join pairs further on produce smaller datasets and run quicker. The same principle applies here in Data Services but to specify the order of join pairs, you can use the rank option.

Cache can be set up if you want the source table to be cached on the ETL server. The type of cache used is determined by the dataflow cache type option.

Query transform performance options
Open the Query transform in the workspace window:

<table>
<thead>
<tr>
<th>Input scheme(s)</th>
<th>From</th>
<th>Join rank</th>
<th>Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUSINESSENTITYADDRESS</td>
<td>✔️</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td>PERSON</td>
<td>✔️</td>
<td>1</td>
<td>Automatic</td>
</tr>
<tr>
<td>ADDRESS</td>
<td>✔️</td>
<td>3</td>
<td>Automatic</td>
</tr>
</tbody>
</table>

Join rank offers the same options as described earlier and allows you to specify the order in which the tables are joined.

Cache is, again, the same as described earlier and defines whether the table will be cached on the ETL server.

lookup_ext() performance options
Right-click on the selected lookup_ext function in the column mapping section or on the function call in the output schema of the Query transform and select Modify Function Call in the context menu:

Available parameters:
- Lookup table
- Input Schema
- Variables

Condition:
- Column in lookup table = "{country=", BUSINESSENTITYID = Get_Country,BUSINESSENTITYID}"
**Cache spec** defines the type of cache method used for the lookup table. **NO_CACHE** means that, for every row in the main dataset, a separate SELECT lookup query is generated, extracting value from the database lookup table. When **PRE_LOAD_CACHE** is used, the lookup table first pulled to the ETL box and cached in memory or on the physical disk (depending on the dataflow cache type option). **DEMAND_LOAD_CACHE** is a more complex method best used when you are looking up repetitive values. Only then is it most efficient. Data Services caches only values already extracted from the lookup table. If it encounters a new key value that does not exist in the cached table, it makes another request to the lookup table in the database to find it and then caches it too.

**Run as a separate process** can be encountered in many other transforms and object configuration options. It is useful when the transform is performing high-intensive operations consuming a lot of CPU and memory resources. If this option is checked, Data Services creates separate subdataflow processes that perform this operation. Potentially, this option can help parallelize object execution within a dataflow and speed up processing and transformations significantly. By default, the OS creates a single process for a dataflow, and if not parallelized, all processing is done within this single OS process. Run as separate process help to create multiple processes helping main dataflow OS process to perform all extracts, join and calculations as fast as possible.

**Target table performance options**

Click on a target table to open its configuration options in the workspace window:

<table>
<thead>
<tr>
<th>Target Options</th>
<th>Bulk Loader Options</th>
<th>Load Triggers</th>
<th>Pre-Load Commands</th>
<th>Post-Load Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Rows per commit:</strong></td>
<td>1000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Delete data from table before loading:</strong></td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Advanced:</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>General:</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Column comparison:</strong></td>
<td>Compare by name</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Number of loaders:</strong></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Error handling:</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Use overflow file:</strong></td>
<td>No</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Row name:</strong></td>
<td>Not Applicable</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Row format:</strong></td>
<td>Not Applicable</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Update control:</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Rows per commit** is similar to **Array fetch size** but defines how many rows are sent to a database within the same network packet. Do decrease amounts of packets with rows for insert sent to a database you can increase this number.

**Number of loaders** helps to parallelize the loading processes. Enable partitions on the table objects on the **Datastores** tab if the tables are partitioned at the database level. If they are not partitioned, set the same number of loaders as **Degree of parallelism**.
Where to buy this book

You can buy SAP Data Services 4.x Cookbook from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.