2

Report Authoring – Best Practices

SQL Reporting Services provides a platform for developing and managing reports in an environment that includes multiple data sources of information. Obtaining the data from these disparate data stores is facilitated by standards such as OLE DB, LDAP, and ODBC. SQL reporting Services, henceforth referred to as SRS in the book, uses Visual Studio as the main report authoring environment to produce a Report Definition Language file or RDL. We will be working extensively in this environment throughout the book to build a reporting solution for our company. Before we introduce all of the many elements of the reporting environment, it is important to begin with the heart of any data-driven report, whether it is Crystal Reports, SRS, or Microsoft Access, and that is the query. With any report design application, the underlying data connection and query to produce the desired data is fundamental.   Developing a query that returns the desired data efficiently is the key to a successful report.

In this chapter we will begin by analyzing the query development process. We will create queries based on real-world applications, the kind that report writers and database administrators create everyday, in addition to other standard data sources, like Active Directory. The performance and value of the report will be defined by the initial query so it is important to have an understanding of the procedure as well as the tools required to quickly create the query. Later, we will use the queries produced in this chapter to build reports.

Query Design Basics

Creating a Simple Query Graphically

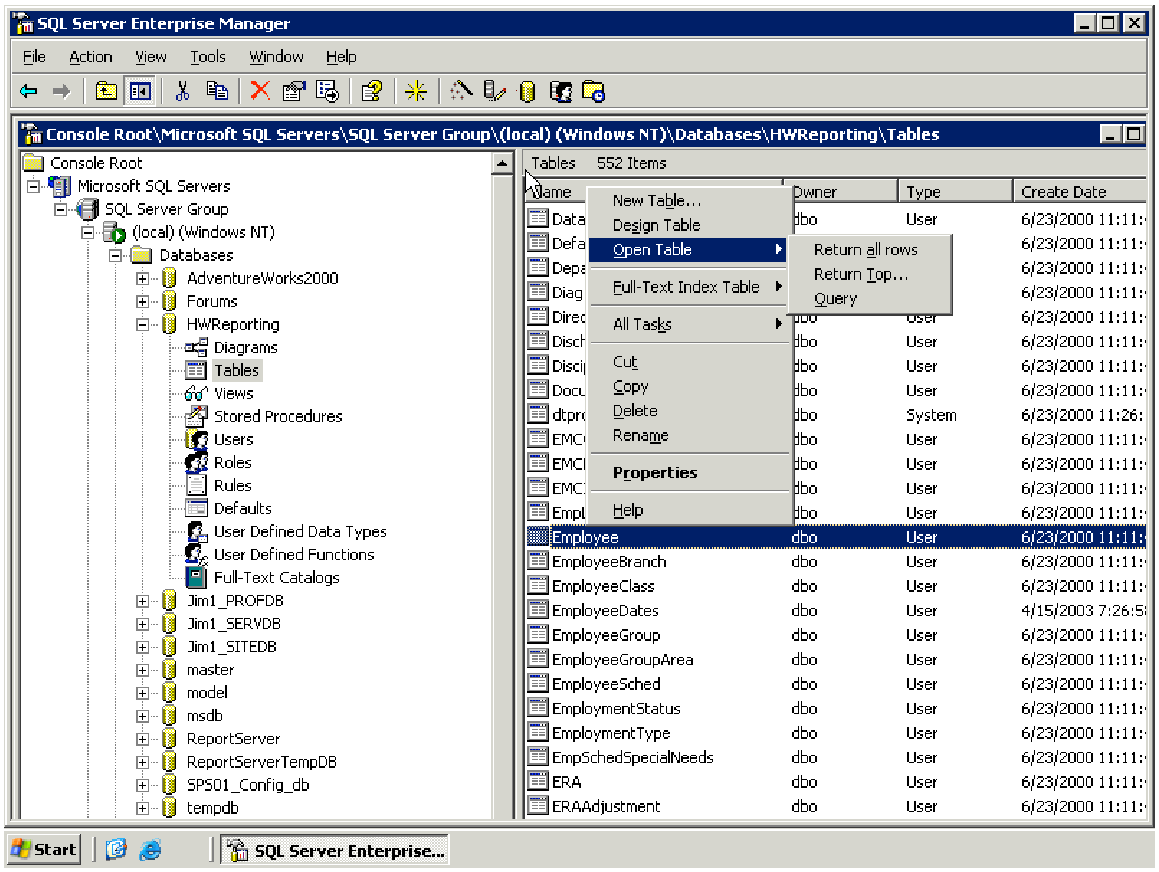
Query design begins typically with the request. As the report writer or DBA, it is common to be tasked with producing data that is otherwise unavailable through standard reports that are often delivered with third-party applications.

Lets begin with a hypothetical scenario that you receive an email that details out a report that will need to be created and deployed for an upcoming meeting, It has already been determined that the data is unavailable from any known reports, yet the data can be derived using a simple custom query.

> In the book we will be creating many of the examples using a real world database designed for the healthcare industry. It is an OLTP (Online Transactional Processing) database that captures billing and clinical information for home health and hospice patients. We will also be creating other examples that can be used by a wide range of professionals. The core concepts of query design can be easily migrated to any RDBMS database that uses SQL.

In the first example we will look at the following request for a healthcare organization: Deliver a report that shows the top 10 diagnosis for patients admitted in the past 120 days.

Assuming that the database administrator or data analyst that was given the task of delivering this report was familiar with the database, the query design process would begin either in Query Analyzer, or a graphical query design application, such as the one included within SQL Enterprise Manager or in Visual Studio. We will begin to design our query with the graphical tool from Enterprise Manager to demonstrate how the underlying SQL code is created. The graphical query designer can be accessed by right-clicking a table that will be used in your query and selecting Open Table\Query, (see Figure 2.1).



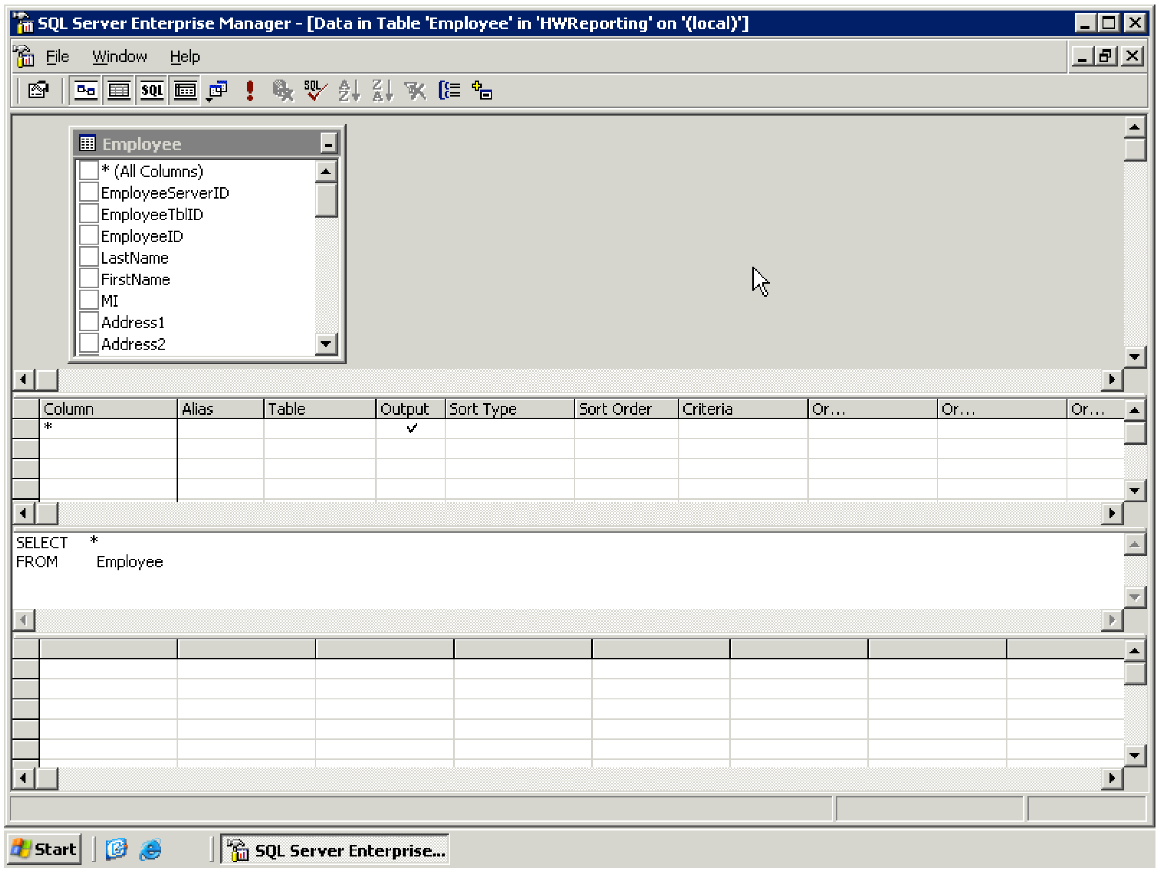
Tables   552 Items

Name	Owner	Type	Create Date
Data	dbo	User	6/23/2000 11:11:
Defa			6/23/2000 11:11:
Depa			6/23/2000 11:11:
Diag			6/23/2000 11:11:
Direc	dbo	User	6/23/2000 11:11:
Disch	dbo	User	6/23/2000 11:11:
Disci	dbo	User	6/23/2000 11:11:
Docu	dbo	User	6/23/2000 11:11:
dtpro	dbo	System	6/23/2000 11:26:
EMC	dbo	User	6/23/2000 11:11:
EMCI	dbo	User	6/23/2000 11:11:
EMC	dbo	User	6/23/2000 11:11:
Empl	dbo	User	6/23/2000 11:11:
Employee	dbo	User	6/23/2000 11:11:
EmployeeBranch	dbo	User	6/23/2000 11:11:
EmployeeClass	dbo	User	6/23/2000 11:11:
EmployeeDates	dbo	User	4/15/2003 7:26:5
EmployeeGroup	dbo	User	6/23/2000 11:11:
EmployeeGroupArea	dbo	User	6/23/2000 11:11:
EmployeeSched	dbo	User	6/23/2000 11:11:
EmploymentStatus	dbo	User	6/23/2000 11:11:
EmploymentType	dbo	User	6/23/2000 11:11:
EmpSchedSpecialNeeds	dbo	User	6/23/2000 11:11:
ERA	dbo	User	6/23/2000 11:11:
ERAAdjustment	dbo	User	6/23/2000 11:11:

Insert 0341f0201.tif

Accessing the graphical query design tool within SQL Enterprise Manager.

Once you have opened the query designer, tasks such as adding and joining additional tables, sorting, grouping and selecting criteria can be performed using the task panes, (see Figure 2-2).



- * (All Columns)
- EmployeeServerID
- EmployeeTblID
- EmployeeID
- LastName
- FirstName
- MI
- Address1
- Address2

```
SELECT    *
FROM      Employee
```

Insert 0341f0202.tif

SQL Enterprise Manager includes a graphical query designer with drag and drop functionality simplifying the process of creating complex queries.

This initial query is a relatively simple one, using 4 tables joined on relational columns. Through the graphical query designer we have added basic criteria and sorting and have selected only two fields for the report, a count of the patients with a specific medical diagnosis. We order the count descending so that we can see the trend for the most common diagnosis. The SQL query that was produced can be directly transported to a report, which we will do in the Chapter 4.

```
SELECT     TOP 10 COUNT(DISTINCT Patient.PatID) AS [Patient Count], Diag.Dscr AS
Diagnosis
FROM       Admissions INNER JOIN
             Patient ON Admissions.PatID = Patient.PatID INNER JOIN
             PatDiag ON Admissions.PatProgramID = PatDiag.PatProgramID INNER JOIN
             Diag ON PatDiag.DiagTblID = Diag.DiagTblID
WHERE     (Admissions.StartOfCare > GETDATE() - 120)
GROUP BY Diag.Dscr
ORDER BY COUNT(DISTINCT Patient.PatID) DESC
```

The SQL query produced using the graphical query designer to return the top 10 patient diagnosis in the past 120 days.

Output from the top 10 diagnosis query.

```
Patient Count    Diagnosis
71               PHYSICAL THERAPY NEC
67               ABNORMALITY OF GAIT
26               HYPERTENSION NOS
21               BENIGN HYP HRT DIS W CHF
20               CONGESTIVE HEART FAILURE
20               DMI UNSPF UNCNTRLD
19               BENIGN HYPERTENSION
19               DMII UNSPF NT ST UNCNTRL
18               CVA
16               URINARY INCONTINENCE NOS
```

This particular query has a very small result set, and even though it is potentially working with tens of thousands of records to produce the resulting 10 records, it runs in under a second so there is no worry that it will impact performance.

This type of query is designed to deliver data for quick review by professionals who will be making business decisions from the results of the data. In this example, a healthcare administrator will notice that there is a demand for physical therapy and may review the staffing level for physical therapists in the company. Because physical therapists are in high demand, the cost of caring for physical therapy patients may need to be investigated. The next query that we develop will look at producing a report that looks at the cost of the care for these patients. We also want to make the query and subsequent report flexible enough that other types of medical services can be analyzed as well, not only physical therapy. This query will require more data for analysis than the previous query for top 10 diagnoses. Because we will be processing many more records the performance impact will need to be assessed.

Creating an Advanced Query

The next query will build upon the initial query and show the cost of caring for patients with specific diagnosis.

The design process will be the same. We will begin by adding the necessary tables to the graphical query designer and selecting the fields that we would like to include in the report (see Figure 2-3). The required data output for the report will need to include the following fields of information:

* Patient Name and ID Number

* Employee Name, Specialty and Branch

* Total Service Count for Patient by Specialty
* Diagnosis of the Patient
* Estimated Cost
* Dates of Services

The query to produce this desired output from our healthcare application looks like the following:

```
SELECT
        Trx.PatID,
        RTRIM(RTRIM(Patient.LastName) + ',' + RTRIM(Patient.FirstName)) AS [Patient Name],
        Employee.EmployeeID,
        RTRIM(RTRIM(Employee.LastName) + ',' + RTRIM(Employee.FirstName)) AS [Employee Name],
        ServicesLogCtgry.Service AS [Service Type],
        SUM(ChargeInfo.Cost) AS [Estimated Cost],
        COUNT(Trx.ServicesTblID) AS Visit_Count,
        Diag.Dscr AS Diagnosis, DATENAME(mm, Trx.ChargeServiceStartDate) AS [Month],
        DATEPART(yy, Trx.ChargeServiceStartDate) AS [Year],
        Branch.BranchName AS Branch
FROM
        Trx INNER JOIN
        ChargeInfo ON Trx.ChargeInfoID = ChargeInfo.ChargeInfoID
        INNER JOIN  Patient ON Trx.PatID = Patient.PatID INNER JOIN
        Services ON Trx.ServicesTblID = Services.ServicesTblID          INNER JOIN
        ServicesLogCtgry ON
        Services.ServicesLogCtgryID = ServicesLogCtgry.ServicesLogCtgryID INNER JOIN
        Employee ON ChargeInfo.EmployeeTblID = Employee.EmployeeTblID INNER JOIN
        Diag ON ChargeInfo.DiagTblID = Diag.DiagTblID INNER JOIN
        Branch on TRX.BranchID = Branch.BranchID
WHERE
        (Trx.TrxTypeID = 1) AND (Services.ServiceTypeID = 'v')
GROUP BY
        ServicesLogCtgry.Service,
        Diag.Dscr,
        Trx.PatID,
        RTRIM(RTRIM(Patient.LastName) + ',' + RTRIM(Patient.FirstName)),
        RTRIM(RTRIM(Employee.LastName)   + ',' + RTRIM(Employee.FirstName)),
        Employee.EmployeeID,
        DATENAME(mm, Trx.ChargeServiceStartDate),
        DATEPART(yy, Trx.ChargeServiceStartDate),
        Branch.BranchName
ORDER BY
```

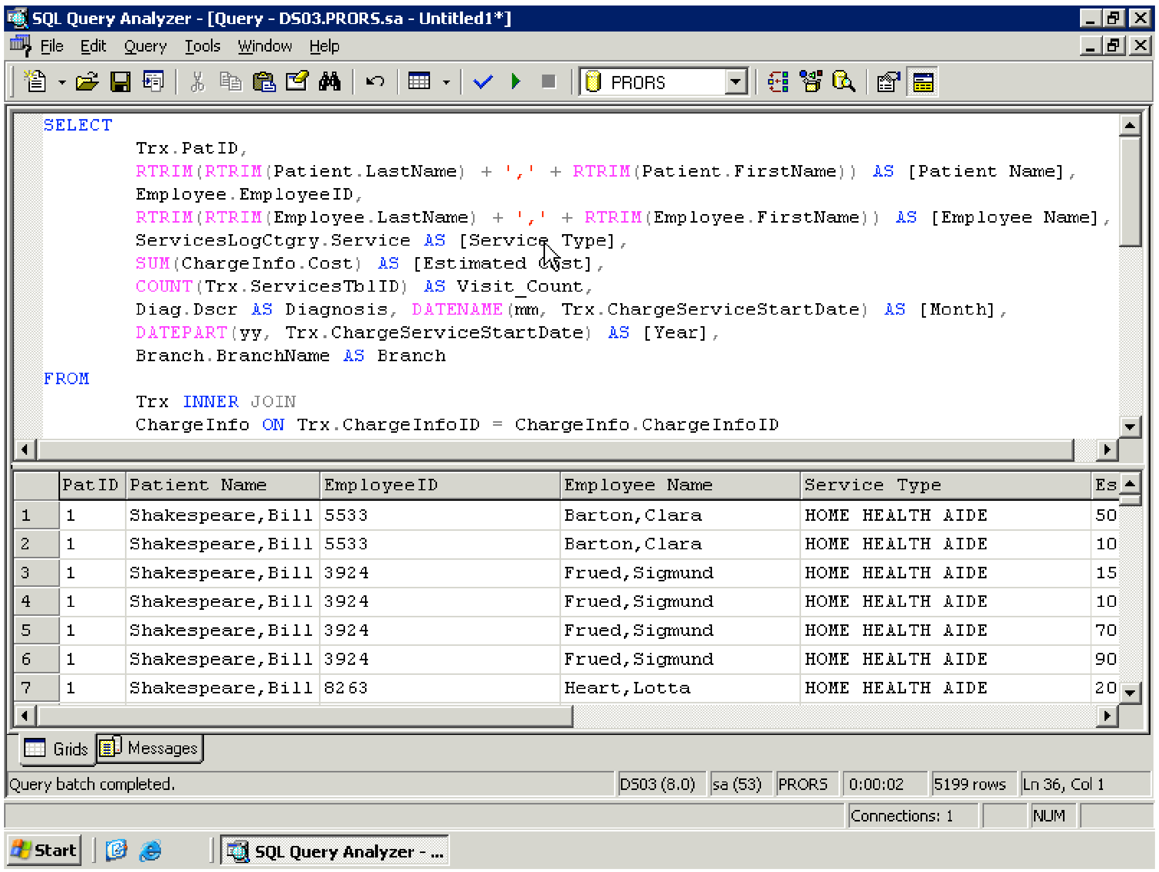
```
Trx.PatID
```

The alias names identified with "AS" in the select clause of the query should serve as pointers to the data that answers the requirements of the report request. Knowing the schema of the database that you will be working with to produce queries, again, is important, but for the sake of the example, the joined tables are typical of a normalized database where detailed transactional data is stored in a separate table than the descriptive information and therefore must be joined together. In this sample query, the Trx table is where the transactional patient service information is stored and the descriptive information of the specialty services like "Physical Therapy" is stored in the Services table. Other tables are also joined, such as the Patient and Employee to retrieve their respective names. SQL functions, COUNT and SUM are used to provide aggregated calculations on cost and service information and RTRIM is used to remove any trailing spaces in the concatenated patient and employee names. The ODER BY PATID clause was used for testing the query to insure that it was returning multiple rows per patient as expected. It is not necessary to add the additional burden of sorting to the query as you will see in the next chapters, sorting will be handled within the report.

Tip: Because SRS and T-SQL share many data formatting and manipulation functions, it is sometimes necessary to decide in which process, query or report, that these functions should be used. By dividing the load between the query and report, you can limit the number of rows that the report will have to work with, making report rendering much faster. In turn, having the report perform additional grouping and calculations, allows the query or stored procedure to execute faster limiting the impact on the SQL Server.

Testing Performance with Query Analyzer

Now that we have our query developed, lets take a quick look at the output to make sure it is going to be all that we need before we move on to the next phase, (see Figure 2-3). The output was created by pasting the SQL query into SQL Query Analyzer, another tool for query design that has much more functionality than its graphical counterpart. The query can be further modified directly in Query Analyzer if desired; however, one of the best features of Query Analyzer that we will be using is the ability to view performance. Once that is done, our next step will be to create the stored procedure.


SQL Query Analyzer - [Query - DS03.PRORS.sa - Untitled1*]
File Edit Query Tools Window Help
PRORS
Grids
Messages
Start
SQL Query Analyzer - ...

```
SELECT
        Trx.PatID,
        RTRIM(RTRIM(Patient.LastName) + ',' + RTRIM(Patient.FirstName)) AS [Patient Name],
        Employee.EmployeeID,
        RTRIM(RTRIM(Employee.LastName) + ',' + RTRIM(Employee.FirstName)) AS [Employee Name],
        ServicesLogCtgry.Service AS [Service Type],
        SUM(ChargeInfo.Cost) AS [Estimated Cost],
        COUNT(Trx.ServicesTblID) AS Visit_Count,
        Diag.Dscr AS Diagnosis, DATENAME(mm, Trx.ChargeServiceStartDate) AS [Month],
        DATEPART(yy, Trx.ChargeServiceStartDate) AS [Year],
        Branch.BranchName AS Branch
FROM
        Trx INNER JOIN
        ChargeInfo ON Trx.ChargeInfoID = ChargeInfo.ChargeInfoID
```

```
   PatID Patient Name      EmployeeID  Employee Name   Service Type       Es
1  1     Shakespeare,Bill  5533        Barton,Clara    HOME HEALTH AIDE   50
2  1     Shakespeare,Bill  5533        Barton,Clara    HOME HEALTH AIDE   10
3  1     Shakespeare,Bill  3924        Frued,Sigmund   HOME HEALTH AIDE   15
4  1     Shakespeare,Bill  3924        Frued,Sigmund   HOME HEALTH AIDE   10
5  1     Shakespeare,Bill  3924        Frued,Sigmund   HOME HEALTH AIDE   70
6  1     Shakespeare,Bill  3924        Frued,Sigmund   HOME HEALTH AIDE   90
7  1     Shakespeare,Bill  8263        Heart,Lotta     HOME HEALTH AIDE   20
```

```
Query batch completed.   DS03 (8.0)   sa (53)   PRORS   0:00:02   5199 rows   Ln 36, Col 1
Connections: 1   NUM
```

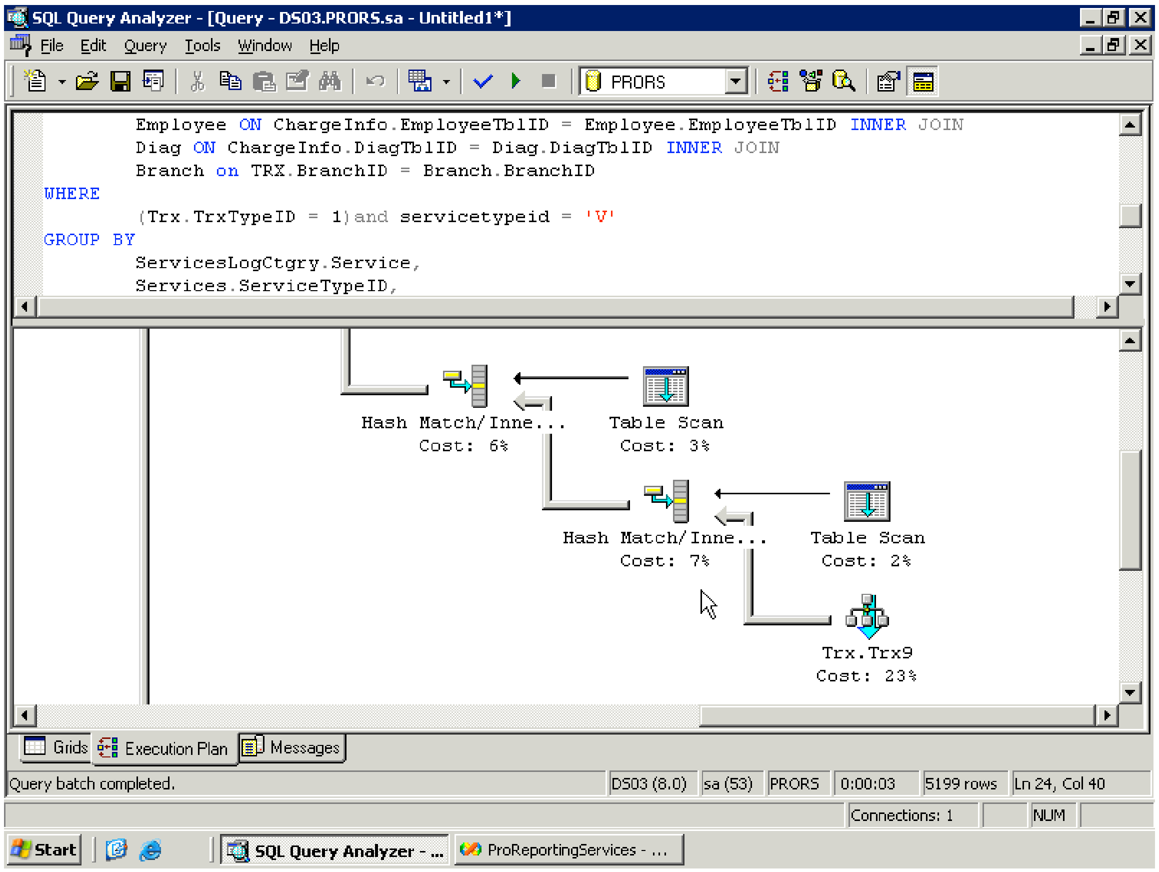
Insert 0341f0203.tif

Shows the output of the query that returns the patient's and employee's names as well as the service types and diagnosis for the patient with visit counts and dates.

We now have our data the way we want and at first appearance it is executing within acceptable timeframes as evidenced by the fact that it has returned 5199 records under 3 seconds. Let's look at the Execution Plan to gain a better understanding of what is happening when we execute the query. In Query Analyzer select Query on the menu bar and then select Show Execution Plan. When the query is executed, another tab will appear in the Results pane called Execution Plan.

The Execution Plan in Query Analyzer is a graphical representation of how the SQL query optimizer chose the most efficient method for executing the report, based on the different elements of the query; for example a clustered index may have been chosen instead of a table scan. Each execution step has an associated cost. When we take a closer look at our query, which ran in three seconds, it is easy to see which section of the query had the highest cost percentage. The WHERE clause in the query had a 23% cost when determining the TrxTypeID and the Service Type. (For reference, the TrxTypeID is a integer field that specifies the type of financial transactions, as charge, payment or adjustments. We are concerned only with TrxTypeID of 1, which are charges. For Service Type, we are only interested in "V", which are visits, and not other types of billable services, such as medical supplies.) Take a look at Figure 02-04 to see the

Execution Plan for our query. Notice that the query also took three seconds to execute. If we could get the cost of the WHERE clause down to a lower number the query may improve in overall performance.



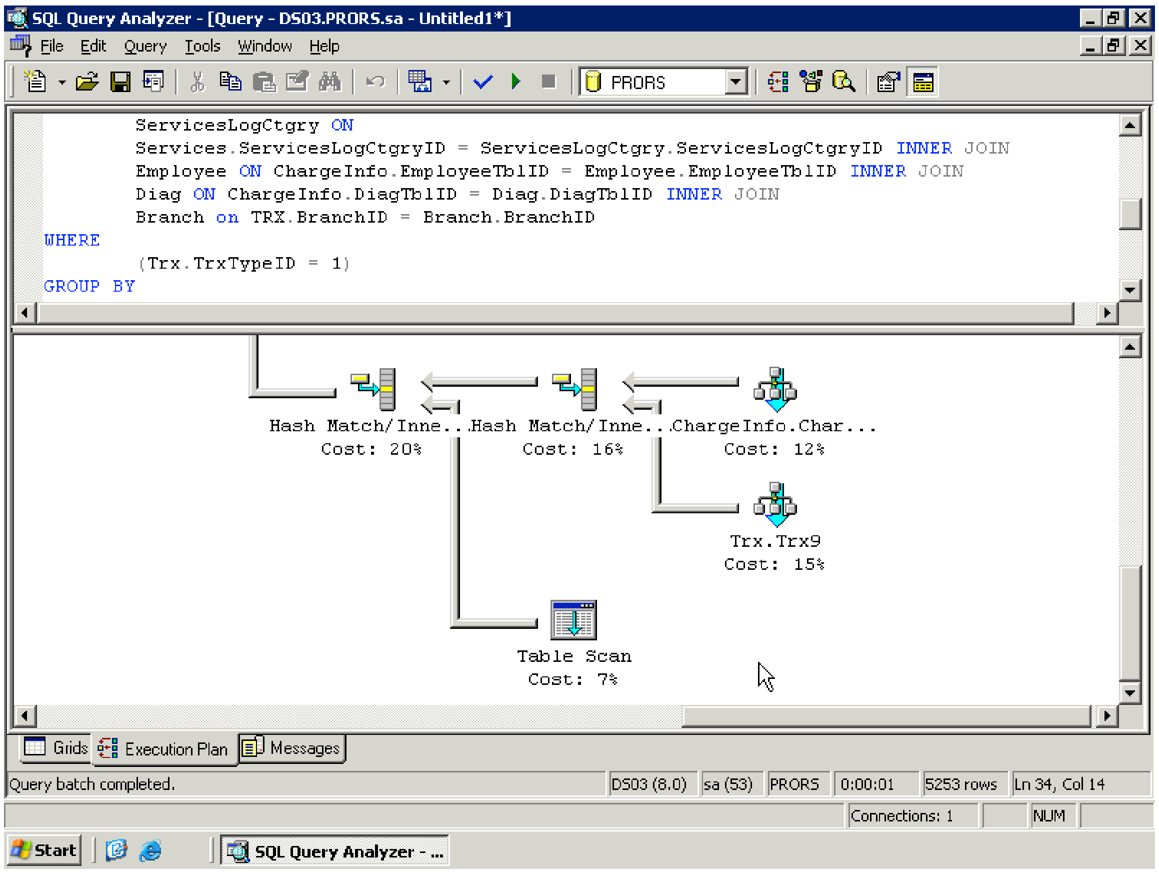
```
        Employee ON ChargeInfo.EmployeeTblID = Employee.EmployeeTblID INNER JOIN
        Diag ON ChargeInfo.DiagTblID = Diag.DiagTblID INNER JOIN
        Branch on TRX.BranchID = Branch.BranchID
WHERE
        (Trx.TrxTypeID = 1) and servicetypeid = 'V'
GROUP BY
        ServicesLogCtgry.Service,
        Services.ServiceTypeID,
```

```
Query batch completed.    DS03 (8.0)  sa (53)  PRORS  0:00:03  5199 rows  Ln 24, Col 40
```

Insert Figure 0341f0204.tif

The Execution Plan for our query indicates that there is a 23% cost associated with WHERE clause.

As stated earlier, the decision will need to be made on whether to allow the query or stored procedure to return more records and let the report perform additional filtering when necessary, or the opposite, which is to make the stored procedure handle the bulk of the load. In our query, based on initial benchmarking, we have determined that we will let the report filter out any non visits and we will remove the portion of the WHERE clause that returns only Service Types of “V”. When we remove the Service Type criteria from the query and re-execute it, we see that our overall execution time went from 3 seconds to 1 and the cost of the WHERE clause went from 23% to 15%. Also, it is important to note in the performance analysis that our record count went up by only 54 records, from 5199 to 5253 See (Figure 02-05).



```
        ServicesLogCtgry ON
        Services.ServicesLogCtgryID = ServicesLogCtgry.ServicesLogCtgryID INNER JOIN
        Employee ON ChargeInfo.EmployeeTblID = Employee.EmployeeTblID INNER JOIN
        Diag ON ChargeInfo.DiagTblID = Diag.DiagTblID INNER JOIN
        Branch on TRX.BranchID = Branch.BranchID
WHERE
        (Trx.TrxTypeID = 1)
GROUP BY
```

```
Query batch completed.    DS03 (8.0)  sa (53)  PRORS  0:00:01  5253 rows  Ln 34, Col 14
                                                     Connections: 1      NUM
```

Insert Figure 0341f0205.tif

By removing a portion of the WHERE clause, our performance increased significantly with only a minimal number of additional rows in the results.

In order to take advantage of a report filter to do the work of the WHERE clause that we have just altered, we will need to add an additional field to our query, called ServiceTypeID that we will use as the filter value. By proceeding in this fashion, even though we are returning more rows than we need for a particular report, we do also gain the benefit of using this same stored procedure for other reports, which may include service types other than visits. For example, there may be a need to investigate the cost or quantity of supplies (a Service Type of “S”) used by employees. This same query and stored procedure can be used for that report as well.

The query that we have outlined, now including the ServiceTypeID as a value in the SELECT clause and not as criteria, is ready to begin its life as a stored procedure. Queries serve many purposes and are good to develop reports with as we will do in chapter 4; however, for best performance, maintenance and parameterization, stored procedures are the preferred method for deployment. In the next section we will create a stored procedure based on our employee cost query.

Making a Parameterized Stored Procedure

There are many way to create a stored procedure from the query we have written. We could, for example, use a template from Query Analyzer and fill in the appropriate information for dropping and creating the stored procedure if it existed in the database. The simplest method for creating a stored procedure is in Enterprise Manager. In the database where the procedure will be created, we will simply right-click the stored procedures folder and select "New Stored Procedure". This opens a window that has a single "Create" command for the new stored procedure:

```
CREATE PROCEDURE [OWNER].[PROCEDURE NAME] AS
```

To complete our new stored procedure which we will name Emp_Svc_Cost, we simply need to past in our select statement. However, we do know that we will provide optional parameters with the stored procedure. These parameters will be used to limit the result set for the following criteria based on the service time (Year and Month), the branch where the employee works, the individual employee, and the type of service. To create parameters for a stored procedure, we will add in the variable names preceding by the "@" characters and provide the appropriate data types and initial value; the initial value for all of the parameters will be NULL.

```
CREATE PROCEDURE [dbo].[Emp_Svc_Cost]
@ServiceMonth  Int=NULL,
@ServiceYear Int=NULL,
@BranchID int=NULL,
@EmployeeTblID int=NULL,
@ServicesLogCtgryID char(5)=NULL
AS
SELECT
        Trx.PatID,
        RTRIM(RTRIM(Patient.LastName) + ',' + RTRIM(Patient.FirstName)) AS [Patient
Name],
        Branch.BranchName,
        Employee.EmployeeID,
        RTRIM(RTRIM(Employee.LastName) + ',' + RTRIM(Employee.FirstName)) AS
[Employee Name],
        Employee.EmployeeClassID,
        ServicesLogCtgry.Service AS [Service Type],
        SUM(ChargeInfo.Cost) AS [Estimated Cost],
        COUNT(Trx.ServicesTblID) AS Visit_Count,
        Diag.Dscr AS Diagnosis, DATENAME(mm, Trx.ChargeServiceStartDate) AS
[Month],
        DATEPART(yy, Trx.ChargeServiceStartDate) AS [Year] ,
        Services.ServiceTypeID
FROM
        Trx INNER JOIN
        Branch on Trx.Branchid = Branch.BranchID Inner Join
        ChargeInfo ON Trx.ChargeInfoID = ChargeInfo.ChargeInfoID
```

```
          INNER JOIN  Patient ON Trx.PatID = Patient.PatID INNER JOIN
          Services ON Trx.ServicesTblID = Services.ServicesTblID              INNER JOIN
          ServicesLogCtgry ON
          Services.ServicesLogCtgryID = ServicesLogCtgry.ServicesLogCtgryID INNER JOIN
          Employee ON ChargeInfo.EmployeeTblID = Employee.EmployeeTblID INNER JOIN
          Diag ON ChargeInfo.DiagTblID = Diag.DiagTblID
WHERE
          (Trx.TrxTypeID = 1) AND
          (isnull(Branch.BranchID,0) = isnull(@BranchID,isnull(Branch.BranchID,0))) AND
          (isnull(Services.ServicesLogCtgryID,0) = isnull(@ServicesLogCtgryID,
isnull(Services.ServicesLogCtgryID,0)))   AND
          (isnull(Employee.EmployeeTblID,0) = isnull(@EmployeeTblID,
isnull(Employee.EmployeeTblID,0))) AND

--Case to determine if Year and month was passed in

          1=Case
                    When ( @ServiceYear is  NULL) then 1
                    When ( @ServiceYear is  NOT NULL)
                    AND @ServiceYear = Cast(DatePart(YY,ChargeServiceStartDate) as int)
then 1
          ELSE 0
          End
AND
          1=Case
                    When (@ServiceMonth is NULL)  then 1
                    When (@ServiceMonth is NOT NULL)
                    AND @ServiceMonth = Cast(DatePart(MM,ChargeServiceStartDate) as
int) then 1
          ELSE 0
          END
GROUP BY
          ServicesLogCtgry.Service,
          Diag.Dscr,
          Trx.PatID,
          Branch.BranchName,
          RTRIM(RTRIM(Patient.LastName) + ',' + RTRIM(Patient.FirstName)),
          RTRIM(RTRIM(Employee.LastName)   + ',' + RTRIM(Employee.FirstName)),
          Employee.EmployeeClassid,
          Employee.EmployeeID,
          DATENAME(mm, Trx.ChargeServiceStartDate),
          DATEPART(yy, Trx.ChargeServiceStartDate) ,
          Services.ServiceTypeID
ORDER BY
          Trx.PatID
```

```
GO
```

Next we will grant execute privileges for the stored procedure for the appropriate roles or users. In this case, we will allow the public role to execute the procedure (I am sure the humor was not lost on the developer who knew someone would grant a public execution).

> Note: The test server we are developing the reports on is an isolated and secure system. Typically granting execution privileges to the public role is not recommended. We will lock down both the stored procedure and the report in the chapter on Security.

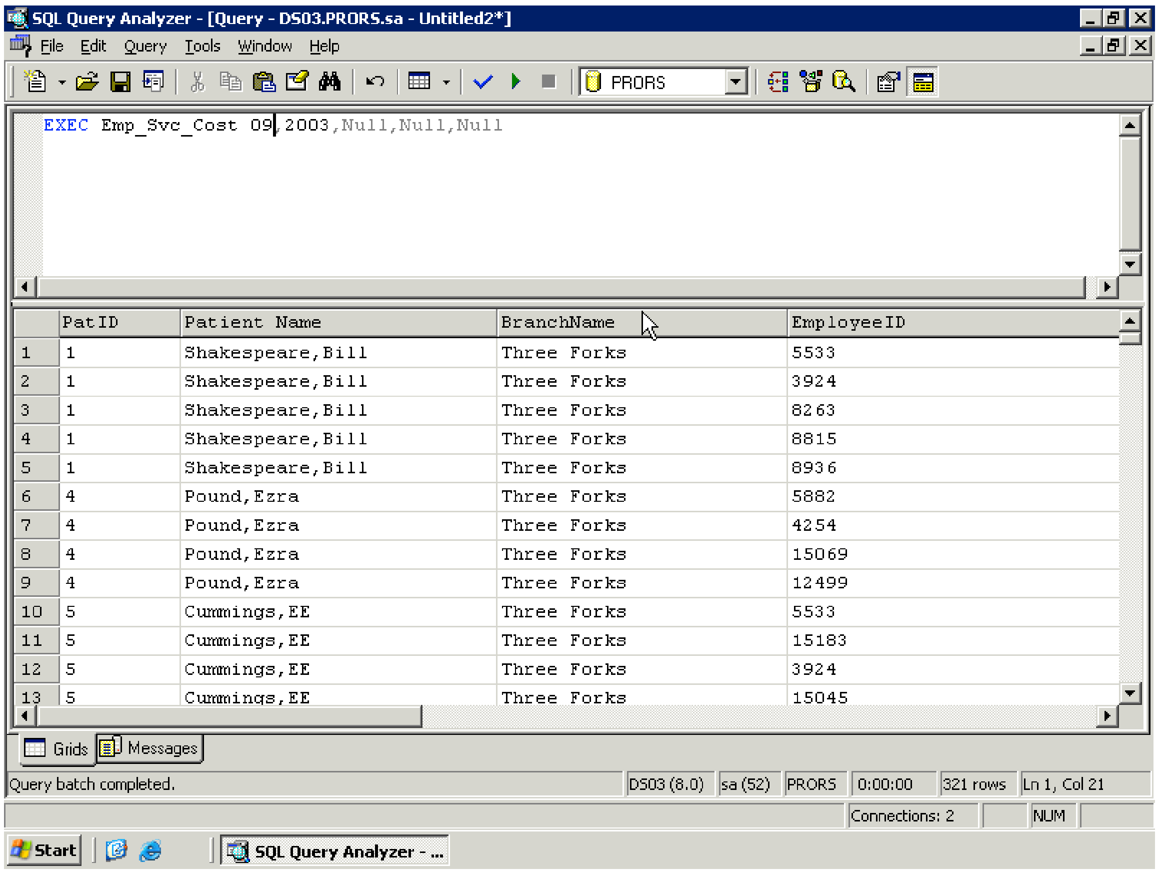
We can now test the procedure directly in query analyzer with the following command:

```
EXEC Emp_Svc_Cost
```

Because we have allowed NULL values for the parameters, we do no explicitly have to pass them in on the command line. However, to test the functionality of the stored procedure we can pass in the full command line with appropriate parameters, say for example, all services rendered in September of 2003:

```
EXEC Emp_Svc_Cost 09,2003,NULL,NULL,NULL
```

Executing the procedure in this way returns 321 records in under a second and the results verify that indeed only services in September of 2003 were returned (See Figure 02-05).


SQL Query Analyzer - [Query - DS03.PRORS.sa - Untitled2*]
File Edit Query Tools Window Help
PRORS
Grids
Messages
Start
SQL Query Analyzer - ...

```
EXEC Emp_Svc_Cost 09,2003,Null,Null,Null
```

	PatID	Patient Name	BranchName	EmployeeID
1	1	Shakespeare,Bill	Three Forks	5533
2	1	Shakespeare,Bill	Three Forks	3924
3	1	Shakespeare,Bill	Three Forks	8263
4	1	Shakespeare,Bill	Three Forks	8815
5	1	Shakespeare,Bill	Three Forks	8936
6	4	Pound,Ezra	Three Forks	5882
7	4	Pound,Ezra	Three Forks	4254
8	4	Pound,Ezra	Three Forks	15069
9	4	Pound,Ezra	Three Forks	12499
10	5	Cummings,EE	Three Forks	5533
11	5	Cummings,EE	Three Forks	15183
12	5	Cummings,EE	Three Forks	3924
13	5	Cummings,EE	Three Forks	15045

```
Query batch completed.    DS03 (8.0)  sa (52)  PRORS  0:00:00  321 rows  Ln 1, Col 21
                                                          Connections: 2     NUM
```

Insert Figure 0341f0206.tif

This screen shows execution times for the new stored procedure, in addition to dead poets brought back to life as ficticious and HIPAA compliant patients.

Evaluating the Parameters

You will notice that we have added several new criteria to the WHERE clause for evaluation of the parameters. We are using the “isnull” function and a “case” to evaluate the values of the database fields and parameters.

```
        (isnull(Branch.BranchID,0) = isnull(@BranchID,isnull(Branch.BranchID,0)))
  AND
        1=Case
        When ( @ServiceYear is  NULL) then 1
        When ( @ServiceYear is  NOT NULL)
        AND @ServiceYear = Cast(DatePart(YY,ChargeServiceStartDate) as int)  then 1
        ELSE 0
        End
```

The logic for these evaluations, at first, may seem confusing, but if you remember that as long as the criteria are equal, results will be returned. This is true through the entire

WHERE clause because it is evaluated with "AND". This is easier to understand with a sample statement:

```
SELECT * from Table1 WHERE 1 = 1
```

In this statement all rows will be returned because 1 will always equal 1. It does not matter that we are not comparing values from the table itself.

For the "isnull" function we look to see if the value of a database field, BranchID for example, contains a NULL value and if so isnull replaces NULL with zero. The right side of that equation looks to see if the @BranchID parameter was passed in as NULL; if so then the value for @BranchID is set to the value of BranchID in the database table and will equal every row. If the @BranchID parameter is passed to the stored procedure as a value, say 2 for the Branch "Nested Valley", then only BranchID 2 will be returned because BranchID = @BranchID = 2. This evaluation is performed when there may be NULL values in the field because NULL values can not be compared with standard operators like "=".

For fields that will always have non-NULL values such as service dates, we can evaluate those with a "case statement in the WHERE clause. For our two time values, Service Year and Month we are using very similar logic as we did with the isnull evaluations. If the Parameters @ServiceMonth and @ServiceYear are passed in as NULL to the stored procedure, it returns every record, the case statement sets the equation to 1 = 1. If the parameters contain legitimate values, such as 2004 as the year, the case statement is set to 1 = 1 only when the parameter equals the value from the database, otherwise it is set to zero and will skip the record.

Knowing Your Data – Quick Trick with a Small Procedure

For every report writer, familiarity with the location of the data in a given database can often come only with time. Of course, have a database diagram or schema provided by a vendor is a useful tool, but this is not always available. One day, faced with the dilemma of trying to find the right table for a specific piece of missing data, I decided to put together a stored procedure, which I named sp_FieldInfo, that returns a list of all of the tables in a specific database that contains the same field names, typically the primary or foreign key fields. In the healthcare database, for example, if you wanted a list of fields that contained the PatID (Patient's ID number that is used to join several tables); the command would look like this:

```
sp_fieldinfo Patid
```

The output would be similar to the following:

```
TableName      FieldName
PatMentalStat  PatID
PatMiscNotes   PatID
PatNoteSummary PatID
PatNurseNotes  PatID
PatNutrition   PatID
PatOccurrence  PatID
```

PatOrders	PatID
PatPhysician	PatID

Armed with this information, I could at least deduce that the patient's physician information, for example, would be stored in the PatPhysician table. Often, however, table and field names are not intuitively named. When I encounter a database like this, from time to time, I will run a profiler trace and perform some actions on the associated application to get a starting point with the captured data.

> Tip: SQL Server Profiler is an excellent tool for capturing not only the actual queries and stored procedures that are executing against the server, it can also be used to capture performance data, such as the duration of the execution time, CPU and I/O measurements, as well as the application that initiated the query. Because this data can be saved directly to a SQL table, it can be analyzed readily, and even makes good fodder for a source for a report in SRS.

The very simple stored procedure text for sp_FieldInfo:

```
CREATE PROCEDURE sp_FielInfo
(
 @column_name            nvarchar(384) = NULL
 )
AS
SELECT
        Object_Name(id) as "Table Name",
        rtrim(name) as "Field Name"
FROM
        syscolumns
WHERE
        Name like @column_name
```

Other Data Sources

One exciting aspect of SRS is its ability to query multiple data source types in addition to SQL Server. Any ODBC or OLE-DB provider can be a potential data source for SRS. We will be covering two of these additional providers while covering both developing reports in SRS as well as examining an overall Business Intelligence platform with Analysis Services in a later chapter.

For a simple example of using a data source other than a SQL Server database, let's look at the OLE-DB Provider for Directory Services. By using a direct LDAP query, it is possible to generate field information for use in SRS.

```
SELECT cn,sn,objectcategory,department
  FROM 'LDAP://DirectoryServerName/ OU=OuName ,DC=Company,DC=Com'
```

The query uses a standard SQlLdialect that will return the common name, surname, objectcategory (computer or person) and department from the Active Directory. The field names will be automatically created and can be used like any other data field for a report.

There are couple of caveats that must be taken into consideration when querying AD, as well as other data sources that do not support the graphical query designer in SRS.

* Query Parameters are not supported directly in the query, however, Report Parameters can be defined and utilized both in the query, referred to as dynamic query, or used to filter data.
* Because a graphical query designer is not available, the query will need to be developed in the generic query designer by typing the query directly and testing. This requires knowledge of Active Directory objects and names.

Tip: There are several tools available to assist in the development and management of Active Directory, such as ADAM (Active Directory Application Mode) and LDP, an Active Directory tool included in the Windows Support Tools.

Deciding the Best Route

In this chapter we have begun the process of designing the essential part of a report, the query and stored procedure. By utilizing stored procedures we gain the benefits of central administration and security and at the same time gain the ability to execute compiled code to return the data set instead of a stand alone query. Queries can be developed in conjunction with the report, in the built-in query tools within SRS, however, when the report is deployed, it is best to deploy it with a stored procedure.

A report request and target audience will be the deciding factor when determining the layout and default rendering of the report. However, it is important to consider that even though reports will often be designed to answer a specific need, if they are based on the same tried and true stored procedures, with similar parameters and groupings, the data will be accurate across all reports and the design time can be focused on the report itself and not re-writing queries.