

# Pro VS 2005 Reporting Using SQL Server and Crystal Reports



Kevin S. Goff and Rod Paddock

## **Pro VS 2005 Reporting Using SQL Server and Crystal Reports**

**Copyright © 2006 by Kevin S. Goff and Rod Paddock**

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN-13 (pbk): 978-1-59059-688-3

ISBN-10 (pbk): 1-59059-688-9

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Lead Editor: Matt Moodie

Technical Reviewer: Jason Lefebvre

Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Gennick, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Keir Thomas, Matt Wade

Project Manager: Sofia Marchant

Copy Edit Manager: Nicole Flores

Copy Editor: Ami Knox

Assistant Production Director: Kari Brooks-Copony

Production Editor: Katie Stence

Compositor: Molly Sharp

Proofreader: Nancy Riddiough

Indexer: Broccoli Information Management

Artist: April Milne

Cover Designer: Kurt Krames

Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail [orders-ny@springer-sbm.com](mailto:orders-ny@springer-sbm.com), or visit <http://www.springeronline.com>.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail [info@apress.com](mailto:info@apress.com), or visit <http://www.apress.com>.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at <http://www.apress.com> in the Source Code/Download section.



# More T-SQL 2005 Stored Procedures

In the previous chapter, you built the complete SQL code for the timesheet report. You also constructed several new user-defined functions (UDFs) and utilized some new SQL Server 2005 language features. In this chapter, you'll do the following:

- Build the result sets for the **construction job summary** report, which will make further use of the APPLY capability.
- Build the stored procedure for the **aging receivables** report, which will further utilize the PIVOT capability.
- Learn about a wide variety of different language features in T-SQL, and how they can be used both for your reporting requirements as well as general database tasks that developers typically face. Here is a list of what we'll cover:
  - How to query for the top N rows in a table, where N is a variable number, by using the new TOP N capability in SQL Server 2005. (In previous versions of SQL Server, the N was a literal.)
  - Building stored procedures to recursively query against hierarchical data, using the new **common table expression** capabilities in SQL Server 2005.
  - How to perform audit trail logging using Update triggers.
  - How to get immediate feedback on updates using the new OUTPUT and OUTPUT INTO keywords in SQL Server 2005.
  - How to build queries dynamically using dynamic SQL, and how to build alternative solutions by using the CASE statement and the COALESCE function.
  - Ranking result sets using the new built-in **ranking** functions in SQL Server 2005.
  - Implementing TRY...CATCH error-handling (including nested error handling) in SQL Server 2005.
  - Using the LIKE keyword to perform partial text searches.
  - Working with XML data.

- Using Dateparts to work with Datetime data
- A cleaner alternative to IN and NOT IN
- One final look at the APPLY operator in SQL Server 2005 for correlated subqueries
- Get a brief look at SQL Server 2000 code for functional equivalents of certain features we've covered in the last two chapters.

## Building the Construction Job Summary Report

Back in Chapter 1, we identified a job summary and profit report that displays aggregate totals (labor, materials, profit, etc.) for each construction job. Figure A-2 in Appendix A shows an example of the report. The report allows an end user to sort on any aggregate column, so it is an easy way to see which jobs were the most profitable, had the least amount of labor, etc. Fortunately, you're already part of the way there: you can leverage some of the UDFs you wrote in Chapter 2.

The construction job summary report contains one line per construction job (i.e., one job in the JobMaster table). Table 3-1 lists the measures for the report, along with a brief description that the client has provided.

**Table 3-1.** *Measures for the Construction Job Summary Report*

Column	Description
Total Labor \$	Sum of hours worked on the job, multiplied by labor and overhead rates
Total Material \$	Sum of material purchased for the job
Labor Profit \$	Sum of hours worked on the job, multiplied by an additional labor profit rate
Material Profit \$	Total Material \$, multiplied by a material markup percentage
Addtl Labor Profit \$	Additional lump-sum labor profit dollars, passed on to the client
*Total Profit \$	Sum of the three profit figures
*Total Contracted Amt	Total Labor \$ + Total Material \$ + Total Profit \$
Total Amount Billed	Amount actually invoiced to date
Total Amount Received	Total amount received to date
*Balance	Difference between total billed and total received
*Total Profit %	Total Profit % divided by Total Contracted Amt, expressed as a %

Of course, any SQL code should derive from a specification that defines these calculations in more technical detail. The specification may also reveal opportunities for reusable functions. So let's take a second look at the report's measures (excluding the ones marked with an asterisk, which are just simple calculations of the calculated columns).

- *Total Labor \$*: Must retrieve all workers who worked on the job and the hours they worked (from the TimeSheets table), determine their labor rates and overhead rates, and summarize this data.
- *Total Material \$*: Must retrieve all material purchases for the job from Materials.

- *Labor Profit \$*: Must sum the hours worked on the job (from TimeSheets) and multiply it by JobMaster.AdditionalHourlyRate.
- *Material Profit \$*: Take the data from Total Material \$ and multiply it by JobMaster.MaterialMarkupPct.
- *Addtl Labor Profit \$*: This is simply the column JobMaster.Additionallabor.
- *Total Amount Billed*: The sum of Invoices.InvoiceAmount for all invoices related to that construction job.
- *Total Amount Received*: The sum of Receipts.AmountPaid for that construction job (must join Receipts to Invoices).

If you want to continue with the practice of using user-defined functions, you are looking at **three** new functions to retrieve data at the construction job level. The first must calculate the labor statistics (and can utilize some of the existing UDFs you built in Chapter 2), the second must calculate the material purchase costs, and the third must retrieve the invoice information.

## Determining the Total Labor Data

First, let's take a look at the UDF to determine total labor data at the job level. In a nutshell, doing so requires the following steps:

- Determining the employees who worked on a job
- Determining the number of hours they worked each day on the job
- Determining the rates (GetWorkerRates from Listing 2-5 in Chapter 2)
- Bringing that data together, rolling up (summarizing) those figures, and applying any additional labor profit (markups) defined for the job

Listing 3-1 displays the UDF to determine the number of hours a worker works on a job. The query is fairly simple: it retrieves hours from TimeSheets and performs the PIVOT operations to convert rows for regular hours and OT hours into columns. Note the use of the COALESCE function: this handy function allows you to write code to handle instances where you want to query on either one job or all. If the calling procedure passes a NULL value for the JobNumber parameter, the query will simply join the JobNumber column to itself—essentially retrieving every job. (Later in this chapter, we'll cover COALESCE in a little more detail.)

### Listing 3-1. UDF GetWorkerHours

```
CREATE FUNCTION [dbo].[GetWorkerHours]
( @nWorkerFK int, @dStartDate DATETIME, @dEndDate DATETIME, @nJobNumber int )
RETURNS
@tWorkerHours TABLE
( WorkerFK int, WorkDate DateTime, RegularHours decimal(10,2), OTHours
  decimal(10,2) )
```

```

AS
BEGIN
    -- You may want hours for just one job, or for all jobs,
    -- so use COALESCE

    -- Again, use PIVOT to place hours into Regular and OT

    INSERT INTO @tWorkerHours
        SELECT * FROM (
            SELECT WorkerFK, Workdate, LKRateTypeFK, HoursWorked
            FROM TimeSheets
            WHERE WorkerFK = @nWorkerFK
            AND JobMasterFK = COALESCE(@nJobNumber, JobMasterFK))
            AS Temp
        PIVOT ( SUM(HoursWorked) FOR LKRateTypeFK In ( [1],[2])) As X
    RETURN
END

```

Now that you have functions to retrieve both hours and rates, you can put it all together in yet another function, `GetJobLaborTotals` (see Listing 3-2). Note the `CROSS APPLY` references to `dbo.GetWorkerHours` and `dbo.GetWorkerRates`. Also, note the code in step 1: the UDF creates a derived table (`WorkerList`) to determine each worker and the minimum/maximum date worked for the job.

The UDF joins the results from `WorkerList` with the results of the `GetWorkerHours` and `GetWorkerRates` (`WorkerHours` and `WorkerRates`).

**Listing 3-2.** *GetJobLaborTotals for a Single Job*

```

CREATE FUNCTION [dbo].[GetJobLaborTotals]
(
    @nJobMasterFK int
)
RETURNS
    @tTotalJobCosts TABLE
(
    JobMasterFK int, TotJobHours decimal(14,2), Labor decimal(14,2), LaborProfit
        decimal(14,2)
)
AS
BEGIN

    INSERT INTO @tTotalJobCosts
        SELECT JobMaster.JobMasterPK, TotJobHours, Labor,
            (TotJobHours * AdditionalHourlyRate) +
                ISNULL(JOBMASTER.AdditionalLabor,0) AS LaborProfit
        FROM JobMaster
        JOIN
            (SELECT JobMasterPK, SUM(ISNULL(RegularHours,0) +

```

```

        ISNULL(OTHours,0)) AS TotJobHours ,
        SUM(ISNULL((RegularHours * RegularRate),0) +
        ISNULL((OTHours * OTRate),0) +
        (( ISNULL(RegularHours,0) + ISNULL(OTHours,0)) *
        OverHeadRate)) AS Labor
FROM JobMaster

-- Step 1, From TimeSheets, get workers on the job,
-- and for each one, their minimum and maximum Work Date
-- (to use for calls to GetWorkerHours and GetWorkerRates)
-- Derived table called WorkerList
JOIN (SELECT JobMasterFK, WorkerFK, MIN(WorkDate) AS MinWorkDate,
        MAX(WorkDate) AS MaxWorkDate
        FROM TimeSheets
        GROUP BY JobmasterFK, workerfk) AS WorkerList
ON WorkerList.JobMasterFK = JobMaster.JobmasterPK

-- Step 2, take each WorkerFK/MinWorkDate/MaxWorkDate on the job,
-- get their hours worked and daily rates (as hours and rates)
CROSS APPLY dbo.GetWorkerHours
        (WorkerList.Workerfk, MinWorkDate, MaxWorkDate, JobMasterPK)
        AS WorkerHours
CROSS APPLY dbo.GetWorkerRates
        (WorkerList.Workerfk, MinWorkdate, MaxWorkdate)
        AS WorkerRates
WHERE JobMasterPK = @nJobMasterFK AND
        Workerlist.WorkerFK = WorkerRates.WorkerFK AND
        WorkerRates.WorkerFK = WorkerHours.WorkerFK AND
        WorkerRates.Workdate = WorkerHours.Workdate
        GROUP BY JobMasterPK) AS TempJob
ON TempJob.JobMasterPK = JobMaster.JobMasterPK

RETURN
END

```

## Retrieving Material Purchases

Second, you need a simple UDF to retrieve material purchases associated with the job. Fortunately, this is much easier to create than the previous UDFs. Listing 3-3 displays the UDF for `GetJobMaterials`. The UDF simply queries `Materials` on the `JobNumber` to summarize the purchases, and also applies any markup percentage to determine material profit.

### Listing 3-3. *GetJobMaterials*

```

CREATE FUNCTION [dbo].[GetJobMaterials]
    (@nJobNumberFK int )
RETURNS

```

```

@tMaterials TABLE
(
    JobMasterFK int, PurchaseAmount decimal(14,2), MaterialProfit decimal(14,2)
)

AS
BEGIN
    INSERT INTO @tMaterials
        SELECT JobMasterFK, SUM(PurchaseAmount) AS PurchaseAmount,
               SUM(PurchaseAmount * MaterialMarkupPct) AS MaterialProfit
        FROM JobMaster
             LEFT JOIN Materials ON JobMasterPK = JobMasterFK
        WHERE JobMasterPK = @nJobNumberFK
        GROUP BY JobMasterFK

    RETURN
END

```

## Retrieving Invoiced Amounts and Payments

Third, you need a simple UDF to retrieve any invoiced amounts and any payments associated with the job. Again, this is a simple UDF. Listing 3-4 displays the UDF for `GetJobInvoiceTotals`. The UDF simply queries the `Invoices` table on `JobNumber` (and the `Receipts` table on `Invoice number`) to retrieve any invoices amounts and payments.

### Listing 3-4. *GetJobInvoiceTotals*

```

ALTER FUNCTION [dbo].[GetJobInvoiceTotals]
( @nJobNumberFK int )
RETURNS
    @tInvoices TABLE
    (JobMasterFK int, InvoiceAmount decimal(14,2), AmountPaid decimal(14,2),
     Balance decimal(14,2) )
AS
BEGIN
    DECLARE @InvoiceAmount decimal(14,2), @AmountReceived Decimal(14,2)
    SET @InvoiceAmount = (SELECT SUM(InvoiceAmount) FROM InvoiceJob
                          WHERE JobMasterFK = @nJobNumberFK)
    SET @AmountReceived = (SELECT SUM(AmountReceived) FROM Receipts
                           WHERE JobMasterFK = @nJobNumberFK)

    INSERT INTO @tInvoices
        VALUES (@nJobNumberFK, @InvoiceAmount, @AmountReceived,
               @InvoiceAmount - @AmountReceived)

    RETURN
END

```



## Putting It All Together

Finally you can put it all together. Listing 3-5 lists the final stored procedure (GetJobSummaries) to produce the result set for the report. Once again, the stored procedure utilizes many topics we've previously covered: use of the APPLY operator, use of the CommaListToTable function, and use of the CASE statement to either query on all jobs (if the JobList parameter is blank) or on the contents of the JobList parameter.

**Listing 3-5.** *GetJobSummaries, Final Stored Procedure to Produce Report Result Set*

```
CREATE PROCEDURE [dbo].[GetJobSummaries]
    -- Add the parameters for the stored procedure here
    @cJobList VARCHAR(MAX)

AS
BEGIN
    -- SET NOCOUNT ON added to prevent extra result sets from
    -- interfering with SELECT statements.
    SET NOCOUNT ON;

    SELECT  JobMasterPK, ClientName, JM.Description, TotJobHours, Labor, LaborProfit,
            PurchaseAmount, MaterialProfit , MaterialProfit + (TotJobHours *
            AdditionalHourlyRate) + JM.Additionallabor AS TotalProfit,
            Labor + PurchaseAmount + MaterialProfit +
            (TotJobHours * AdditionalHourlyRate) + JM.Additionallabor
            AS ContractedAmount,
            InvoiceAmount, AmountPaid , Balance
    FROM JobMaster JM
        -- Table-valued UDF to create table of jobs
        CROSS APPLY dbo.CommaListToTable(@cJobList ) AS JobList
        -- Table-valued UDF - returns total hours, labor, & labor profit for a job
        CROSS APPLY dbo.GetJobLaborTotals (JM.JobMasterPK) AS JobLaborTotals
        -- Table-valued UDF - returns materials purchased/profit for a job
        CROSS APPLY dbo.GetJobMaterials ( JM.JobMasterPK) AS Materials
        -- Table-valued UDF to return amounts invoiced and amounts received
        CROSS APPLY dbo.GetJobInvoiceTotals(JM.JobMasterPK) AS JobInvoices

    WHERE JobLaborTotals.JobMasterFK = JM.JobMasterPK
        AND JobInvoices.JobmasterFK = JM.JobMasterPK
        AND Materials.JobMasterfk = JM.JobMasterPK
        -- Either join on the list of jobs selected, or run for ALL
        AND CASE
            WHEN @cJobList = '' THEN 1
            WHEN JobList.IntKey = JobMasterPK THEN 1
            ELSE 0 END = 1

END
```

Figure 3-1 shows an example of running `GetJobSummaries`.



EXEC `GetJobSummaries` '1,2'

	JobMasterPK	ClientName	Description	TotJobHours	Labor	LaborProfit	PurchaseAmount	MaterialProfit	TotalProfit	ContractedAmount
1	1	KINGSTON CONSTRUCTION	Job #1	18.00	548.30	36.00	1000.00	70.00	106.0000	1654.3000
2	2	GOLDGATE CONTRACTING	Job #2	2.00	78.70	0.00	2000.00	100.00	100.0000	2178.7000

**Figure 3-1.** Example result set for construction job summary report

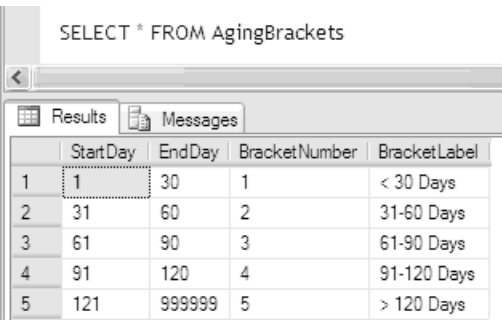
# Building the Result Sets for an Aging Receivables Report

Before we cover the queries for the aging receivables report, you need to define a new data-base table. The requirements for the aging receivables report call for configurable aging brackets. While aging brackets normally run in 30-day increments (1–30 days, 31–60 days, etc.), the client may want to run for custom ranges. Table 3-2 lists the structure for the table `AgingBrackets`.

**Table 3-2.** Table for Data-Driven Aging Brackets (`AgingBrackets`)

Column Name	Type	Description
StartDay	int	First day of the bracket range (e.g., day 31)
EndDay	int	End day of the bracket range (e.g., day 60)
BracketNumber	int	Bracket number for the aging columns (e.g., 2)
BracketLabel	char(50)	“31-60 Days”

Figure 3-2 displays the default entries.



SELECT \* FROM `AgingBrackets`

	StartDay	EndDay	BracketNumber	BracketLabel
1	1	30	1	< 30 Days
2	31	60	2	31-60 Days
3	61	90	3	61-90 Days
4	91	120	4	91-120 Days
5	121	999999	5	> 120 Days

**Figure 3-2.** Default values for table `AgingBrackets`

Listing 3-6 presents the stored procedure `GetAgingReceivables`. The code produces the result sets for the aging receivables report. Of interest are the following:

- The code uses the `CommaListToTable` UDF for a variable number of customers.
- The code joins the `Invoices` table against the `AgingBrackets` table and calculates the number of days each invoice is aged (with respect to the aging date) by using `dateparts` (the parameter in the `DateDiff` function).
- The code determines the corresponding aging bracket for the number of days aged and PIVOTs on that bracket to produce the aging bracket columns.
- In addition to the detailed result sets, the stored procedure also produces summary result sets for the aging bracket and the list of customers. These will also be used on the final report output.

**Listing 3-6.** *Complete Stored Procedure for Retrieving Aging Invoices*

```
CREATE PROCEDURE [dbo].[GetAgingReceivables]
    @cCustomerList VARCHAR(MAX), @dAgingDate DateTime, @lShowDetails bit
AS
BEGIN
    SET NOCOUNT ON;

    -- Table variable to hold results
    DECLARE @tAgingDetails TABLE
        (ClientFK int, InvoiceNumber char(20), InvoiceDate DateTime,
        TotalAged decimal(14,2), Bracket1 decimal(14,2), Bracket2 decimal(14,2),
        Bracket3 decimal(14,2), Bracket4 decimal(14,2), Bracket5 decimal(14,2) )

    INSERT INTO @tAgingDetails
    SELECT * FROM
        -- Grab open invoices, determine balance owed
        (SELECT ClientFK, InvoiceNumber, InvoiceDate, 0 AS TotalAged,
            InvoiceAmount-SUM(ISNULL(AmountReceived,0)) AS AmountOwed,
            ABR.BracketNumber
        FROM Invoices Inv
            -- Determine days aged, find the corresponding bracket
            JOIN AgingBrackets ABR ON DateDiff(dd,InvoiceDate,@dAgingDate)
                BETWEEN ABR.StartDay and ABR.EndDay
            LEFT JOIN Receipts RCT ON InvoicePK = INvoiceFK
        -- If run for selected customers
        CROSS APPLY dbo.CommaListToTable(@cCustomerList) AS CustomerList
        WHERE InvoiceClosed = 0
        -- Check if customers selected....if not, run for all
        AND CASE
            WHEN @cCustomerList = '' THEN 1
            WHEN CustomerList.IntKey = Inv.ClientFK THEN 1
```

```

ELSE 0 END = 1

GROUP BY
    ClientFK, InvoiceNumber, InvoiceDate, InvoiceAmount, ABR.BracekNumber)
AS Temp
-- Pivot amount owed into the correct bracket
PIVOT ( SUM(AmountOwed) FOR BracketNumber In ( [1],[2],[3],[4],[5]))
As Brackets

-- Post query munging, set the total aged column
-- (helpful in summarizing)
UPDATE @tAgingDetails SET TotalAged = ISNULL(Bracek1,0) +
                                         ISNULL(Bracek2,0) +
                                         ISNULL(Bracek3,0) +
                                         ISNULL(Bracek4,0) +
                                         ISNULL(Bracek5,0)

-- Four result sets

-- 1) Return detail aging data...if detail option was not selected, just bring back
empty structure
IF @lShowDetails = 1
    SELECT * FROM @tAgingDetails
ELSE
    SELECT * FROM @tAgingDetails WHERE 1=0

-- 2) Summary aging data
SELECT ClientFK, SUM(Bracek1) AS Bracke1,
          SUM(Bracek2) AS Bracke2,
          SUM(Bracek3) AS Bracke3,
          SUM(Bracek4) AS Bracke4,
          SUM(Bracek5) AS Bracke5,
          SUM( TotalAged ) AS TotalAged
FROM @tAgingDetails GROUP BY ClientFK

-- 3) The aging brackets (to display descriptions on the report)
SELECT * FROM AgingBrackets

-- 4) Client names for clients in the report
SELECT ClientPK, ClientName FROM Client
JOIN @tAgingDetails ON ClientPK = ClientFk
GROUP BY ClientPK, ClientName

END

```

Figures 3-3 and 3-4 show two different examples of running the aging receivables report: one for a single customer, and one for all customers.

```

DECLARE @dAgingDate DateTime, @cCustomerList varchar(100), @lShowDetails bit
SET @dAgingDate = GETDATE()
SET @cCustomerList = '2'
SET @lShowDetails = 1
EXEC GetAgingReceivables @cCustomerList, @dAgingDate, @lShowDetails

```

	ClientFK	InvoiceNumber	InvoiceDate	TotalAged	Bracket1	Bracket2	Bracket3	Bracket4	Bracket5
1	2	2005-1954	2005-05-01...	12000.00	0.00	0.00	0.00	0.00	12000.00

	ClientFK	TotalAged	Bracket1	Bracket2	Bracket3	Bracket4	Bracket5
1	2	12000.00	0.00	0.00	0.00	0.00	12000.00

	StartDay	Endday	BracketNumber	BracketLabel
1	1	30	1	< 30 Days
2	31	60	2	31-60 Days
3	61	90	3	61-90 Days
4	91	120	4	91-120 Days
5	121	999999	5	> 120 Days

	ClientPK	ClientName
1	2	GOLDGATE CONTRACTING

**Figure 3-3.** Running the aging receivables report for one customer, as of today

```

DECLARE @dAgingDate DateTime, @cCustomerList varchar(100), @lShowDetails bit
SET @dAgingDate = CAST('1-1-06' AS DateTime)
SET @cCustomerList = ''
SET @lShowDetails = 1
EXEC GetAgingReceivables @cCustomerList, @dAgingDate, @lShowDetails

```

	ClientFK	InvoiceNumber	InvoiceDate	TotalAged	Bracket1	Bracket2	Bracket3	Bracket4	Bracket5
1	1	2005-0662	2005-09-01...	0.00	0.00	0.00	0.00	0.00	0.00
2	1	2005-0778	2005-10-01...	2000.00	0.00	0.00	0.00	2000.00	0.00
3	1	2005-1209	2005-12-18...	3500.00	3500.00	0.00	0.00	0.00	0.00
4	1	2005-2182	2005-12-01...	3000.00	0.00	3000.00	0.00	0.00	0.00
5	2	2005-1954	2005-05-01...	12000.00	0.00	0.00	0.00	0.00	12000.00
6	3	2005-0944	2005-08-01...	13000.00	0.00	0.00	0.00	0.00	13000.00
7	3	2005-1001	2005-10-18...	17500.00	0.00	0.00	17500.00	0.00	0.00

	ClientFK	TotalAged	Bracket1	Bracket2	Bracket3	Bracket4	Bracket5
1	1	8500.00	3500.00	3000.00	0.00	2000.00	0.00
2	2	12000.00	0.00	0.00	0.00	0.00	12000.00
3	3	30500.00	0.00	0.00	17500.00	0.00	13000.00

	StartDay	Endday	BracketNumber	BracketLabel
1	1	30	1	< 30 Days
2	31	60	2	31-60 Days

	ClientPK	ClientName
1	1	KINGSTON CONSTRUCTION
2	2	GOLDGATE CONTRACTING
3	3	L & L EXCAVATING INC.

**Figure 3-4.** Running the aging receivables report for all customers, as of January 1

---

**Note** Chapters 2 and 3 have made heavy use of T-SQL User-Defined functions. Our goal is to demonstrate reusability. However, this is just one application. In your applications, you may choose to take some of the UDFs and incorporate them directly into the higher-level queries. There are always trade-offs between reusability and performance: use the knowledge of your specific situations to judge which approach is best.

---

## General SQL Server Programming Tasks

You can think of this next section as a miscellaneous section, albeit a large one. In different online forums, user groups, or client interactions, we've seen the same or similar questions asked over and over. So let's take some time and cover different database programming tasks that developers face: some might have relevance to reporting applications, though perhaps not directly related to the application in this book. Other tasks may be outside the scope of reporting applications, but are still important areas that SQL developers will want to examine. For that reason, we'll simply use the Northwind database that comes with SQL Server 2000 for this next section.

### TOP N Reporting

Let's take a look at the new TOP N capability in T-SQL 2005. As a basic explanation, TOP N instructs SQL Server to only return the first N number of rows in the result set, based on whatever sort order you specify. In SQL Server 2000, the N had to be a literal: to execute TOP N queries in SQL Server (where N was variable), the developer had to construct a SQL string and then execute it dynamically. While this worked, it resulted in code that is arguably a bit more difficult to maintain—not to mention that it probably discouraged less-experienced developers who wanted to avoid dynamic SQL altogether. Alternatively, the developer could set the ROWCOUNT system variable to limit the result set: this had limitations in more complex SQL statements and also required you to restore it afterwards.

Fortunately, SQL Server 2005 allows developers to specify a variable. This should encourage developers to place code for TOP N reporting in the database layer, as opposed to implementing a portion of it outside the database. Listing 3-7 demonstrates a simple but meaningful example of retrieving the TOP N jobs based on labor profit—by using one of the UDFs you previously built.

#### Listing 3-7. Code to Demonstrate TOP N

```
DECLARE @nTop int
SET @nTop = 5
SELECT TOP(@nTop) JobMasterPK, LaborProfit
    FROM JobMaster
        CROSS APPLY dbo.GetJobLaborTotals(JobMaster.JobMasterPK) JobLabor
    WHERE JobLabor.JobMasterFK = JobMaster.JobMasterPK
    ORDER BY LaborProfit DESC
```

By default, using TOP N will return the number of rows specified by N. If you want to return the top N% rows, you can use the PERCENT keyword:

```
SELECT TOP(@nTop) PERCENT...
```

The TOP variable can even be the result of a UDF or any expression that returns an integer, such as the following query from the Northwind Orders database:

```
SELECT TOP( SELECT COUNT(*) FROM SHIPPERS ) * FROM ORDERS
```

In SQL Server 2005, you can also use the TOP N capability with INSERT, UPDATE, and DELETE statements, to limit the scope of the command.

```
-- Create a table variable with three rows, but only update the first two
DECLARE @tTest1 TABLE ( Amount decimal(10,2))
INSERT INTO @ttest1 VALUES ( 100)
INSERT INTO @ttest1 VALUES ( 200)
INSERT INTO @ttest1 VALUES ( 300)
UPDATE TOP(@nTop) @tTest1 SET Amount = Amount * 10

-- Now create a second table variable,
-- and insert the first two rows from the first one
DECLARE @tTest2 TABLE ( Amount decimal(10,2))
INSERT TOP(2) @tTest2 SELECT * FROM @tTest1 order by amount
```

## Common Table Expressions/Recursive Queries

Prior to SQL Server 2005, performing a recursive query (i.e., a query that calls another query) meant creating temporary tables and writing multiple SQL statements. A common example is querying hierarchical data. While not specifically part of the examples, you want to take a minute to look at a powerful new way to query hierarchical data in SQL Server 2005.

SQL Server 2005 allows developers to perform true recursive queries through common table expressions (CTEs). CTEs are similar to derived tables, and they allow you to write recursive queries in a single SQL statement.

As a basic example, take a look at the music hierarchy presented in Table 3-3.

**Table 3-3.** *Music Hierarchy Stored in MusicData*

<b>Data</b>	<b>Primary Key</b>	<b>Parent Key</b>
Musicians	1	NULL
Jazz	2	1
Rock	3	1
Classical	4	1
Saxophone	5	2
Trumpet	6	2
Guitar	7	3
Piano	8	4
Charlie Parker	9	5
John Coltrane	10	5
Miles Davis	11	6
Eddie Van Halen	12	7
Franz Liszt	13	8

For any given search name (might be an instrument, a musician, etc.), you want to determine that name's parent row, and then its parent, and so on, all the way up to the top of the hierarchy. While this may seem a meaningless example, imagine if this were a sales hierarchy, or even a product bill-of-material structure.

Listing 3-8 contains a query to retrieve a specific row and all parent rows. There are three elements to this query. First, you declare a CTE (*MusicTree*) with column names. You can think of this as similar to a derived table. Second, the inside query (or anchor query) retrieves the row based on the search criteria (in this case, Charlie Parker). Finally, the recursive query performs a join between the results in *MusicTree* and each parent, until the search is exhausted (i.e., reaches the top level).

**Listing 3-8.** *Example of Recursive Queries and CTEs*

```
DECLARE @cSearch char(50)
SET @cSearch = 'Charlie Parker'
;
WITH MusicTree (ResultName, PKValue)
AS (SELECT Name, ParentPK
    FROM MusicData
    WHERE Name = @cSearch
    UNION ALL
    SELECT Name, parentPK
    FROM MusicData
    INNER JOIN MusicTree ON PKValue = MainPK )
SELECT * FROM MusicTree
```

The results are displayed in Table 3-4.

**Table 3-4.** *Results of Recursive Query*

Result Name	PKValue
Charlie Parker	5
Saxophone	2
Jazz	1
Musicians	NULL

By default, SQL Server 2005 supports 100 recursion levels. For more information and examples on this powerful new feature, check for either CTEs or Recursive CTEs in the SQL Server 2005 help system.

While this example may seem a bit simple, there are many serious applications for common table expressions. For example, you may be building an e-commerce solution with a standard set of database tables. However, one client may have three levels of product/item hierarchy, another client may have four, etc. You can store the hierarchies in a fixed database schema and use recursive queries to deal with many different parent-child relationships. Another example might be the storage of menu options in a database: at any one time, you may need to retrieve and display available menu options, based on the current menu selection.



## Implementing Audit Trail Functionality

Storing database changes is a common requirement in database systems. For certain key pieces of data in an application, a client will want to know every time the data was changed: who changed it and when, the value of the data before the change, and the value of the data after the change. A frequent question on technical forums is how to implement audit trail functionality. Sometimes the person asking the question will know part of the answer but is missing some piece of the puzzle. Perhaps the person knows that Update triggers are part of the answer but isn't quite sure of the syntax. Other times, that person believes (erroneously) that the UPDATED function can be used to determine whether columns have changed.

Update triggers are indeed the area where audit trail processing occurs. An Update trigger is a specific type of stored procedure that fires every time an UPDATE statement executes against the table. Update triggers provide access to two critical system tables that contain the state of the row before it was updated (Deleted) and after it was updated (Inserted). Before we take you into full-blown audit trail functionality, we'll first show you a simpler use of the Inserted system table to get a taste of what you can do with it.

For example, suppose you have a column in a Client table called LastUpdated: you want to always store the server date and time on any updated row(s). Even if someone manually changes a row in the Client table outside the application (in Query Analyzer, SQL Management Studio, etc.), you always want LastUpdated to reflect the date and time the changes occurred. You can implement the following code in an Update trigger:

```
CREATE TRIGGER Upd_Client ON dbo.Client
FOR UPDATE
AS
    UPDATE Client
        SET LastUpdated = GETDATE()
    FROM Client C
        JOIN Inserted I
            ON I.PrimaryKey = C.PrimaryKey
```

Again, the Inserted system table contains one row for every row that was updated. So if someone manually executes an Update system that affects five rows, the trigger will fire once, and the Inserted table will contain the five rows. You can join against the Inserted table on the table's primary key column to update the LastUpdated column.

---

**Note** In SQL Server, an Update trigger fires once per UPDATE statement. Never try to assign the value of a column from the Inserted or Deleted system tables to a variable under the assumption that the Inserted/Deleted tables will only contain one row. This is a common beginner's mistake that usually leads to a runtime error.

---

Now that you see how you can use the Inserted system table, let's take a look at detecting database changes and then logging them. Table 3-5 shows the structure for an audit trail log that will serve as your history for every database change: you want to insert into this table every time a change occurs.

**Table 3-5.** *Structure for an Audit Trail History (AuditTrail)*

ColumnName	Type	Description
TableName	char	The name of the table where the change occurred
PrimaryKey	int	The primary key value of the row that was changed
LastUpdated	datetime	The date/time of the change
ColName	char	The name of the column that was changed
OldValue	varchar	The value of the column before the change
NewValue	varchar	The value of the column after the change

Listing 3-9 shows the code for an Update trigger on the Product Master table to track changes, by querying the Inserted and Deleted tables. For this example, you'll check the PRICE and DESCRIPTION columns for changes, and write the values from Inserted and Deleted into your AuditTrail log. Take notice that when you write out the old/new value of Price to the audit log, you must convert the price column from a decimal column to a character column.

**Listing 3-9.** *Update Trigger to Detect and Log Price Changes*

```

CREATE TRIGGER Upd_Price ON dbo.Price
FOR UPDATE
AS
    DECLARE @dLastUpdate DATETIME
    SET @dLastUpdate = GETDATE()
    -- Set the last Update for rows updated
    UPDATE Price
        SET LastUpdate = @dLastUpdate
    FROM Price P
        JOIN Inserted I
            ON I.PrimaryKey = P.PrimaryKey

    -- Write out to the audit Log, for rows where price changed

    INSERT INTO AuditTrail (TableName, PrimaryKey, LastUpdate, ColName,
        OldValue, NewValue, UserKey)
        SELECT 'PRICE' AS TableName, I.PrimaryKey, @dLastUpdate AS LastUpdate,
            'Price' AS ColName, CONVERT(CHAR(20),D.Price) AS OldValue,
            CONVERT(CHAR(20),I.Price) AS NewValue,
            I.UserKey
    FROM Inserted I
        JOIN Deleted D ON D.PrimaryKey = I.PrimaryKey
    WHERE I.Price <> D.Price

```

---

**Note** Another common “beginner’s mistake” is to use the `UPDATED` function (or `COLUMNS_UPDATED`) to detect which column(s) have changed. These functions return a value of `true` for columns that were referenced in an `UPDATE` statement. Because an `UPDATE` statement may do nothing more than update a column value with the same value before the update, you would be logging a change that didn’t occur. Audit trail logging needs to join the `Inserted` and `Deleted` tables to examine before/after values.

---

## Getting Immediate Feedback with `OUTPUT` and `OUTPUT INTO` in SQL Server 2005

In your audit trail solution in the previous section, you must query against the `AuditTrail` log to view the list of changes. In most instances, that will be fine, but there may be cases where you want immediate feedback on what has been changed. SQL Server 2005 allows you to specify an `OUTPUT` clause on an `UPDATE` statement so that you can immediately output the before/after changes.

In SQL Server 2000, the `Inserted` and `Deleted` system tables (that we mentioned in the previous section) were only available inside an `Update` trigger. SQL Server 2005 expands the visibility of `Inserted/Deleted` by allowing you to specify them in an `OUTPUT` clause of an `UPDATE` statement. The following code snippet creates a table variable with three rows and updates the amount column of the first two. In your `UPDATE` statement, you can `OUTPUT` rows from `Inserted/Deleted` to immediately see what was changed.

```
DECLARE @tTest1 TABLE ( RowNum int, Amount decimal(10,2))
INSERT INTO @ttest1 VALUES (1, 100)
INSERT INTO @ttest1 VALUES (2, 200)
INSERT INTO @ttest1 VALUES (3, 300)
UPDATE TOP(2) @tTest1 SET Amount = Amount * 10
      OUTPUT Inserted.RowNum,Deleted.Amount,Inserted.Amount
```

-- You can also `OUTPUT` into a temporary table for later use in the procedure

---

**Note** While the `OUTPUT` and `OUTPUT INTO` statements provide considerable convenience, do not use them in place of audit trail functionality in `Update` triggers. `Update` triggers give you the protection of firing for **any** update, even ones beyond the control of your application. Think of `OUTPUT` and `OUTPUT INTO` as shortcuts if you need immediate feedback on what was changed, without having to make another callback to the server to call the `AuditTrail` log.

---

## Using Dynamic SQL (and Finding Alternatives)

A common (and sometimes controversial) subject is using dynamic SQL to programmatically construct SQL syntax on the fly, and then executing it using the SQL Server system stored procedure `sp_executesql`. Dynamic SQL is most often used when some key piece of information

on a SQL query (such as a table name, column name, or result set structure) is not known until runtime. While some purists may argue that dynamic SQL is a symptom of a faulty overall design, the reality is that some developers will face situations where some significant aspect of a query is not known until runtime.

Dynamic SQL syntax can be a bit daunting, so Listing 3-10 shows two examples of dynamic SQL.

**Listing 3-10.** *Examples of Dynamic SQL*

```
USE northwind
-- Will return orders where EmployeeID = 5
DECLARE @cSQLSyntax nvarchar(2000)
DECLARE @cTableName varchar(20)
SET @cTableName = 'Orders'
SET @cSQLSyntax = N'SELECT * FROM ' + @cTableName + ' WHERE EmployeeID = 5'
EXEC sp_executesql @cSQLSyntax

-- Return a column into an output variable.
-- Query must only contain one row, else an error will occur
-- (query could also be a SUM that returns a scalar value).

DECLARE @cValue nVarChar(30)
DECLARE @cColumnName varchar(20)
SET @cColumnName = 'OrderID'
SET @cSQLSyntax = N' SELECT @cValue = ' + @cColumnName + ' FROM'
SET @cSQLSyntax = @cSQLSyntax + ' Orders where employeeid = 1'
EXECUTE SP_EXECUTESQL @cSQLSyntax, N'@cValue nVarChar(30) OUTPUT', @cValue OUTPUT
SELECT @cValue
```

However, there are instances where other approaches offer benefits over dynamic SQL. Suppose you have a form (or web page) that allows users to retrieve customers based on several input criteria (first and last name, address, city, zip, etc.) The user may enter one field, two fields, or many fields. You need to write a stored procedure to examine all possible parameters, but only query on those that the user entered.

You could write a stored procedure that examines each parameter, constructs a SQL SELECT string based on parameters the user filled out, and executes the string using dynamic SQL. Many SQL developers opt for this approach.

Alternatively, you can use the SQL Server COALESCE function (see Listing 3-11). COALESCE, available both in SQL Server 2000 and SQL Server 2005, provides you with an alternative approach that arguably leads to cleaner T-SQL code. For each search condition, you pass COALESCE two values: the search variable, and a value to use if the search variable is NULL. So for any search values that the user did not specify, the search defaults to the column being equal to itself. This approach is still very fast, even when querying against millions of rows.

**Listing 3-11.** *Using COALESCE to Deal with Many Parameters, Where Some Could Be NULL*

```

DECLARE @city varchar(50), @state varchar(50), @zip varchar(50),
        @FirstName varchar(50), @LastName varchar(50),
        @Address varchar(50)

SET @FirstName = 'Kevin'
SET @State = 'NY'

SELECT * FROM CUSTOMERS WHERE
        FirstName = COALESCE(@FirstName,FirstName) AND
        LastName = COALESCE(@LastName,LastName) AND
        Address = COALESCE(@Address,Address) AND
        City = COALESCE(@City,City) AND
        State = COALESCE(@State,State) AND
        Zip = COALESCE(@Zip,Zip)

```

Another instance where dynamic SQL is often used is when an SQL statement may have a conditional ORDER BY or GROUP BY clause. As a somewhat contrived and simple (but meaningful) example, suppose you want to sum the freight in the Northwind Orders database, and group it either by EmployeeID or ShipVia method. Some developers might either use dynamic SQL to construct the query (including the group by) at runtime or duplicate the code and change the GROUP BY. You can utilize a CASE statement to build a simpler solution:

```

DECLARE @nGroupOpt int
SET @nGroupOpt = 1 -- 1 for employee, 2 for ship method

SELECT CASE @nGroupOpt WHEN 1 THEN EmployeeID ELSE ShipVia END AS GroupColumn,
        SUM(Freight) AS TotFreight
FROM ORDERS
GROUP BY CASE @nGroupOpt WHEN 1 THEN EmployeeID ELSE ShipVia END
ORDER BY TotFreight DESC

```

## New Functions in SQL Server 2005 to Assign Ranking Values

Suppose you want to rank the top orders by customer, in descending sequence (for orders greater than \$500). You may want to use the ranking order number as a display column on a report. As part of the result set, you want to return a ranking number, as demonstrated here:

Customer	Rank	Description
Customer A	1	Highest order
Customer A	2	Second highest order
Customer A	3	Third highest order
Customer B	1	Highest order

And so on.

The following query reads the Northwind Orders database for orders greater than \$500. It uses the new ROW\_NUMBER OVER function to assign row numbers based on a particular order and the new PARTITION keyword to group the ranking.

use northwind

```
SELECT CustomerID, OH.OrderID, OrderDate,
(UnitPrice * Quantity) as Orderamount,
    ROW_NUMBER() OVER (PARTITION BY CUSTOMERID
    ORDER BY (UnitPrice * Quantity) DESC)
    AS OrderRank
FROM Orders OH
    JOIN [dbo].[Order Details] OD
        ON OH.OrderID = OD.OrderID
WHERE (UnitPrice * Quantity) > 500
    ORDER BY CUSTOMERID, OrderAmount DESC
```

## Implementing Better Error Handling with TRY...CATCH

SQL Server 2005 implements TRY...CATCH functionality to trap runtime errors. You can use syntax similar to the syntax that .NET developers have been able to use for years. This allows you to gracefully check for (and manage) runtime errors and exceptions.

Consider the following code:

```
DELETE from Orders where orderid = 10336
```

This will generate a runtime error because of the constraint between the Order header and Order detail tables in the Northwind database. However, you can wrap this in a basic TRY...CATCH block as follows:

```
BEGIN TRY
    DELETE from Orders where orderid = 10336
END TRY
BEGIN CATCH
    SELECT ERROR_NUMBER() AS ErrNum, ERROR_SEVERITY() AS ErrSev,
        ERROR_STATE() as ErrState, ERROR_MESSAGE() as ErrMsg;
END CATCH
```

This code will return the error information in a result set, and will continue on. You can even build nested TRY...CATCH blocks for more complicated stored procedure requirements!

---

**Note** If you plan to return error conditions as a result set, the data access layer that reads the result set must be prepared to handle a result set that contains this structure.

---

## Using LIKE to Perform Text Searches

The T-SQL LIKE command allows developers to search for patterns within strings. A common use would be searching for the phrase “XP” within a text/character column that contains “skills include Windows XP, FrontPage, etc.”. Developers can use the LIKE command and the wildcard percentage character (%) to perform pattern matching.

You can use LIKE in multiple ways, depending on the type of search you want to perform. Most searches need to check for a pattern that exists anywhere within a column. However, some searches only need to return rows where a column begins with a search pattern. Additionally, some searches may be interested in rows where a column ends with a search pattern. Here are some examples:

```
Search anywhere in the column
SELECT * FROM Applicants WHERE Skills LIKE '%XP%'

-- Search where skills begins with XP
SELECT * FROM Applicants WHERE Skills LIKE 'XP%'

-- Search where skills ends with XP
SELECT * FROM Applicants WHERE Skills LIKE '%XP'
```

Additionally, developers can use the single wildcard underscore character (Name LIKE '\_EVIN'),

## Working with XML Data

SQL Server 2000 provided many great capabilities for working with XML data, and SQL Server 2005 adds even more new functions through the new XML data type and XML querying capabilities. Let's look at two different examples for getting XML data into tables (and also querying out of XML-stored data).

Listing 3-12 shows two different examples of reading XML-supplied data into a simple address table. The first uses attribute-centric mapping. The second uses element-centric mapping and also demonstrates nested columns.

### Listing 3-12. Reading XML Data into Database Columns

```
-- First example
DECLARE @cXMLDoc XML
declare @hdoc int
SET @cXMLDoc = '<AddressType >
  <AddressRecord
    AccountID = "1"
    Street="31 Main Dr" City="Philly" State="PA" Zip="12345"/>
  <AddressRecord
    AccountID = "2"
    Street="1 Wilson Dr" City="Newark" State="NJ" Zip="22222"/>
</AddressType>'
```

```

EXEC sp_xml_preparedocument @hdoc OUTPUT, @cXMLDoc

SELECT * FROM OPENXML (@hdoc, '/AddressType/AddressRecord',1)
    WITH (AccountID int, Street varchar(100), City varchar(100),
        State varchar(10), Zip varchar(13))

GO

-- Second example, uses an Address tag to specify nested columns

DECLARE @cXMLDoc XML
declare @hdoc int
SET @cxmldoc =
'<customer>
    <Customernum>48456</Customernum>
    <Firstname>Kevin</Firstname>
    <Lastname>Goff</Lastname>
    <Address>
        <city>Allentown</city>
        <state>PA</state>
    </Address>
</customer>'

EXEC sp_xml_preparedocument @hdoc OUTPUT, @cxmldoc
DECLARE @tTemp TABLE (Customernum int, Firstname char(50),
    Lastname char(50), City char(50), State Char(10))

insert into @ttemp
    SELECT * FROM OPENXML (@hdoc, '/customer',2)
        WITH (Customernum int, Firstname varchar(50), Lastname varchar(50),
            city varchar(50) './Address/city',
            state varchar(50) './Address/state')

```

To reverse the examples, let's assume you're storing address information in the new XML column `DataType` and need to query based on one of the XML columns. To make things even more interesting, let's suppose you need to query on a subset of a column. Listing 3-13 provides an example of this.

### **Listing 3-13.** *Querying from XML Data*

```

declare @tTest table (address xml)
insert into @ttest values ('<Address >
    <AddrRecord
        AccountID = "1"
        Street="31 Main Dr" City="Newark" State="NJ" Zip="11111" />
    </Address>' )

```



```

insert into @ttest values ('<Address >
    <AddrRecord
        AccountID = "2"
        Street="1 Wilson Rd" City="Philly" State="PA" Zip="22222"/>
    </Address>' )

SELECT * FROM @ttest

WHERE Address.exist('/Address/AddrRecord [contains(@City,"hil")]') = 1

```

## Using Date Parts

Suppose you need to retrieve daily order information and summarize it by a week-ending date (Saturday). You want to write a UDF that converts any date during the week to a Saturday date for that week. You can use SQL Server's DATEPART functions to solve this requirement. As the name implies, DATEPART returns a specific part of a date: day of week, month, year, quarter, etc. (Check SQL Server Books Online for all possible DATEPART options.)

In the following example, you call the DATEPART function and ask it to return the value of the weekday for the date in question, subtract that value from 7, and then add that result to the initial date. That will give you the Saturday date for the week!

```

CREATE FUNCTION dbo.GetEndOfWeek
    (@dDate DateTime)
-- Converts date to the Saturday date for the week
RETURNS DateTime AS
BEGIN
    DECLARE @dRetDate DateTime
    SET @dRetDate = @dDate + ( 7-DATEPART(weekday,@dDate))

    RETURN @dRetDate
END

```

You can then use this UDF as part of a query against orders, to summarize the data by the week-ending Saturday date.

```

SELECT dbo.GetEndOfWeek(OrderDate) AS WeekEnding,
    SUM(Amount) AS WeekAmount
FROM OrderHdr
GROUP BY dbo.GetEndOfWeek(OrderDate)

```

---

**Note** By default, SQL Server's setting for the first day of the week is Sunday. To set the first day of the week to a different day (e.g., summarize sales from Monday to Sunday instead of Sunday to Saturday), use the SET DATEFIRST <NumberVar> command: check the help system for details on the DATEFIRST option.

---

## A Cleaner Alternative to IN: INTERSECT and EXCEPT

Most SQL developers have written queries that filtered result sets based on values simply existing (or not existing) in other tables. In SQL 2000, developers used the IN keyword. For example, if you wanted to query against construction jobs that, among other things, had invoices (or didn't have invoices), you'd write the following code:

```
SELECT JobMasterPK
FROM JobMaster
WHERE JobMasterPK IN
      (SELECT JobMasterFK FROM Invoices)
```

```
SELECT JobMasterPK
FROM JobMaster
WHERE JobMasterPK NOT IN
      (SELECT ISNULL(JobMasterFK,0) FROM Invoices)
```

SQL Server 2005 implements the INTERSECT and EXCEPT operators, which are part of the SQL-99 standard. You can write these same two queries as follows:

```
SELECT JobMasterPK
FROM JobMaster
INTERSECT SELECT JobMasterFK FROM Invoices
```

```
SELECT JobMasterPK
FROM JobMaster
EXCEPT SELECT JobMasterFK FROM Invoices
```

Note that in the first set of examples, you must check for NULL values when searching for jobs that do not have invoices: in the second set of examples, EXCEPT (and INTERSECT) are able to handle NULL values for you.

## APPLY Revisited: Using UDFs with Correlated Subqueries

Suppose, in the Northwind database, you want to know which customers have had at least two orders for more than \$5,000 (or five orders for more than \$1,000 dollars, etc.) variable.

Your first step is to build a table-valued UDF called GetCustOrders. The UDF contains two parameters (customer ID and dollar threshold), and returns a table variable of orders for that customer that exceed the threshold.

```
CREATE FUNCTION [dbo].[GetCustOrders]
(@CustomerID AS varchar(10), @nThreshold AS decimal(14,2))
RETURNS @tOrders TABLE (OrderID int, CustomerID varchar(10),
                        OrderDate datetime, OrderAmount decimal(14,2))
```

```
AS
BEGIN
```

```

INSERT INTO @tOrders
    SELECT OH.OrderID, CustomerID, OrderDate, (UnitPrice * Quantity)
    AS Orderamount
FROM Orders OH
JOIN [dbo].[Order Details] OD
    ON OH.OrderID = OD.OrderID
WHERE CustomerID = @CustomerID AND
    (UnitPrice * Quantity) > @Threshold
ORDER BY OrderAmount DESC
RETURN
END
Go

```

Your next step is to run that UDF against every customer in the database and determine which customers have at least two orders from the UDF. Ideally, you'd like to construct a subquery to pass each customer as a parameter to the UDF. Here's where the power of T-SQL Server 2005 comes in.

In SQL Server 2000, table-valued functions within a correlated subquery could not reference columns from the outer query. Fortunately, T-SQL 2005 removes this restriction and allows you to use a table-valued function in a subquery, where the arguments to the UDF come from columns in the outer query. You can now build a subquery that calls your UDF and passes columns from the outer query as arguments to the UDF:

```

DECLARE @nNumOrders int, @nMinAmt decimal(14,2)
SET @nNumOrders = 2
SET @nMinAmt = 5000.00
SELECT CustomerID FROM Customers
    WHERE (SELECT COUNT(*) FROM
        DBO.GetCustOrders_GT_X(CustomerID,@nMinAmt)) >=@nNumOrders

```

## Using SQL Server 2000

You've used three new functions in SQL2005: TOP N, APPLY, and PIVOT. Implementing similar or comparable capabilities in SQL Server 2000 is as follows:

- To achieve TOP N capability in SQL Server 2000, you must use dynamic SQL to construct and execute a SQL statement where the N value is unknown until runtime. Listing 3-14 shows a brief example. Note the use of the system stored procedure `sq_executesql`.
- To APPLY the results of a table-valued UDF in SQL Server 2000 to every row in the inner expression, you'll need to create a temporary table (or table variable, or derived table) and then perform a JOIN. Listing 2-3 back in Chapter 2 demonstrates a simple example of this.
- To convert source table rows into destination columns, you must hand-code a conversion process by using CASE statements. Listing 3-15 contains an equivalent code snippet for the aging receivables report that utilized a PIVOT back in Listing 3-6.

**Listing 3-14.** *Equivalent Code in SQL Server 2000 to Demonstrate TOP N*

```

DECLARE @nRows int
SET @nRows = 5
DECLARE @cSQL nvarchar(2000)

SET @cSQL = N'SELECT TOP ' + CAST(@nRows AS VARCHAR(5)) + ' * FROM Customer'
EXEC sp_executesql @cSQL

```

**Listing 3-15.** *Equivalent Code in SQL Server 2000 to Demonstrate PIVOT*

```

SUM(CASE WHEN DDate BETWEEN @dAgeDate-30 AND @dAgeDate
      THEN DBalance ELSE 0 END) AS Age30 ,

SUM(CASE WHEN DDate BETWEEN @dAgeDate-60 AND @dAgeDate-31
      THEN DBalance ELSE 0 END) AS Age60 ,

SUM(CASE WHEN DDate BETWEEN @dAgeDate-90 AND @dAgeDate-61
      THEN DBalance ELSE 0 END) AS Age90 ,

SUM(CASE WHEN DDate BETWEEN @dAgeDate-120 AND @dAgeDate-91
      THEN DBalance ELSE 0 END) AS Age120 ,

SUM(CASE WHEN DDate < @dAgeDate - 120
      THEN DBalance ELSE 0 END) AS AgeGT120,

SUM(CASE WHEN DDate > @dAgeDate
      THEN DBalance ELSE 0 END) AS NotAged,
SUM(DBalance) AS TotBalance

```

## Summary

In this chapter, you spent some more time looking at the new APPLY and PIVOT capabilities. You also looked at other new language features, such as variable TOP N, recursive queries with common table expressions, OUTPUT, and new XML features. You also took a look at other tasks, such as audit trail logging, and how you can build solutions to address them. Hopefully these last two chapters have provided valuable insight into how you can use T-SQL to generate result sets and solve many other database problems.

## Recommended Reading

Bob Beauchemin and Dan Sullivan. *A Developer's Guide to SQL Server 2005*. Boston, MA: Addison-Wesley, 2006.

Itzik Ben-Gan et al. *Inside Microsoft SQL Server 2005: T-SQL Querying*. Redmond, WA: Microsoft Press, 2006.

Shawn Wildermuth. "Making Sense of the XML DataType in SQL Server 2005." CoDe Magazine, May/June 2006.