

SQL QUERIES IN PBCONNECT FOR MICROSOFT JET 4.0 OLE DB PROVIDER

By Luc Magnée

Introduction

Microsoft Jet 4.0 OLEDB Provider is used to retrieve data from Microsoft Access databases, Microsoft Excel workbooks or folders containing structured text files (*.csv, *.txt).

Table 1: Equivalence between Access, Excel and Structured Text Files

Access	Excel	Structured Text file
Database	Workbook	Directory
Table <i>tablename</i>	Worksheet <i>sheetname</i> \$ Named Range <i>namedrange</i>	File filename#csv or filename#txt
Field <i>fieldname</i>	Field <i>columnname</i>	Fields as defined in a schema.ini file <i>Columnname</i>
Record	Row	Line

ADO Query is one of the four methods used by pbconnect Retrieve Jobs that link to an OLE DB data source connection, making use of SQL statements.

Even though SQL is defined by both ANSI and ISO standards, it is far from being standardized and is implemented with sometimes significant variations between different database platforms. The information in this document focuses on the Microsoft Jet 4.0 SQL implementation.

General syntax rules

Table and Field names in SQL

General rule: the table and/or field names and/or aliases are written as is.

Exceptions:

If the table and/or field and/or alias name contains special characters like **\$** or a **white space**, or a reserved word (See App. A, Reserved words) the name is enclosed between square brackets. Examples: **[field name]**, **[sheet1\$]**, **[currency]**.

For Excel only: if the sheet name contains a white space, the sheet name is further enclosed in single quotes. Example: **['address list\$']**. Note: a named range cannot contain white spaces.

If a field name is composed of numbers only, it is enclosed between square brackets. Example: **[103205]**.

Data type

Out of the three possible data sources, only one is predictable in terms of data type: Microsoft Access. The data type is the one declared at design time for each field.

Microsoft Excel is a totally different issue as it “guesses” sometimes luckily and sometimes less luckily at the data type, based on the majority of data found in a single field (this in turn is based on the IMEX setting in the data source).

Finally, with the structured text file, it all depends on the schema.ini file: unless you write an extended schema.ini file, with description of all fields together with their data type, Microsoft Jet 4.0 OLEDB Provider also guesses at the data type, the same way it does with Excel files, and with the same results.

The best way to see Microsoft Jet 4.0 OLEDB Provider at work is to have a look at the table “address_list” in the “SQL Pocket Guide.mdb” Access database, the “address_list” sheet in the “SQL Pocket Guide.xls” Excel workbook and the “address_list.csv” file in the “..\SQL Pocket Guide” directory and pay special attention to the data in the field “postal_code”, row 6:

Access

Query: **SELECT * FROM [address list]**

Data Source: ..\SQL Pocket Guide.mdb

Result: A correct retrieval of all data, with the right data type.

Excel

Query: **SELECT * FROM ['address list\$']**

Data Source: ..\SQL Pocket Guide.xls;...;Extended Properties="Excel 8.0;hdr=yes;"... (Note the absence of IMEX setting!)

Result: An incorrect retrieval of all data, based on following “guesses” of the data type:

Fields **ID** : right guess with data type **integer**

Field [**postal code**]: wrong guess with data type **integer** (note the **(null)** data in **row 6**)

Field [**last earning**]: wrong guess with data type **double** (note the **(null)** data in **row 5**)

All other fields: right guess with data type **text**

Data Source: ..\SQL Pocket Guide.xls;...;Extended Properties="Excel 8.0;hdr=yes;IMEX=1;" ... (Note the IMEX setting!)

Result: An almost correct retrieval of the data, based on following luckier “guess” of the data type:

Fields **ID**: right guess with data type **integer**

Field [**last earning**]: still a wrong guess with data type **double** (we want **currency**!)

All other fields: right guess with data type **text** (note the **5431SH** data in field [**postal code**], **row 6**)

Structured text file

Query: **SELECT * FROM [address list#csv]**

Schema.ini (see Appendix B, Schema.ini file):

```
...
...
[Address_list.csv]
Format=CSVDelimited
ColNameHeader=True
MaxScanRows=7
CharacterSet=OEM
...
...
```

Result: An incorrect retrieval of all data, based on following unlucky “guess” (**MaxScanRow** has the same effect on a structured text file as **IMEX=1** on an Excel file) of the data type:

Fields **ID** : right guess with data type **integer**

Field [**postal code**]: wrong guess with data type **integer** (note the (null) data in row 6)

Field [**last earning**]: wrong guess with data type **double** (note the (null) data in row 5)

All other fields: right guess with data type **text**

Schema.ini (see Appendix B, Schema.ini file):

```
...
...
[Address_list.csv]
Format=CSVDelimited
ColNameHeader=True
Col1=ID Short
Col2= "first name" Text
Col3= "last name" Text
Col4=Address Text
Col5= "postal code" Text
Col6=city Text
Col7=phone Text
Col8=mobile Text
Col9= "last earning" Currency
CharacterSet=OEM
...
...
```

Result: A correct retrieval of all data.

Now, why is the data type that important? Aside of some obvious reasons like the ability to perform mathematical operations on the data like sum, average, ... (see Aggregate functions), the data type also dictates the choice of a comparison operator in a SQL clause like WHERE when testing an equality (see Comparison operators).

Finally, the terms “lucky” and “unlucky” used above are also relative: what about a bad formatted Excel sheet where data that should have a data type integer get sometimes a text data type? Luckily enough, there’s a trick to force the data type during the retrieval (Calculated fields and functions).

Summary: When starting a project with a customer where there is hesitation about using Excel or CSV files, always go for the last one. When using an extended schema.ini file, you will always be sure of the data type of the fields.

Table 2: Jet 4.0 Data type

Data type	Description
NUMBERS	
1. Integer numbers	
Bit	Used to represent data that can have one of two values like True/False, Yes/No, On/off,... Uses 1 bit of memory.
Byte	Used to represent positive integers ranging in value from 0 through 255. Uses 1 byte of memory (8 bits).
Short	Used to represent integers ranging in values between -32,768 and 32,767. Uses 2 bytes of memory.
Long	Used to represent integers ranging in values between -2,147,483,648 to 2,147,483,647. Uses 4 bytes of memory.
2. Real numbers for high precision operation	
Currency	<p>The two scaled integer data types, Currency and Decimal, provide a high level of accuracy. These are also referred to as fixed-point data types. They are not as precise as the floating-point data types — that is, they cannot represent numbers as large or as small. However, if you cannot afford rounding errors, and you do not require as many decimal places as the floating-point data types provide, you can use the scaled integer data types. Internally, the scaled integer types represent decimal values as integers by multiplying them by a factor of 10.</p> <p>The Currency data type uses 8 bytes of memory and can represent numbers with fifteen digits to the left of the decimal point and four to the right, in the range of -922.337.203.685.477,5808 to 922.337.203.685.477,5807.</p> <p>The Decimal data type uses 17 bytes of memory and can have between 0 and 28 decimal places.</p>
Decimal	
3. Real numbers	
Single	Used to represent real numbers ranging in negative values between -3.402823×10^{38} and $-1.401298 \times 10^{-45}$ and in positive values between 1.401298×10^{-45} and 3.402823×10^{38} . Uses 4 bytes of memory.
Double	Used to represent real numbers ranging in negative values between $-1.79769313486232 \times 10^{308}$ and $-4.94065645841247 \times 10^{-324}$ and in positive values between $4.94065645841247 \times 10^{-324}$ and $1.79769313486232 \times 10^{308}$. Uses 8 bytes of memory.
DATE AND TIME	
DateTime	Self explanatory. Uses 8 bytes of memory.
TEXT	
Text	String containing up to 255 characters. Uses 2 bytes of memory per character.
Memo	String containing up to 65,536 characters. Uses 2 bytes of memory per character.

Querying tables with SQL

The simple SELECT statement

The most basic and most often used SQL statement is the SELECT statement. SELECT statements are the workhorses of all SQL statements, and they are commonly referred to as *select queries*. You use the SELECT statement to retrieve data from the database tables, and the results are usually returned in a set of records (or rows) made up of any number of fields. You must designate which table or tables to select from with the FROM clause. The basic structure of a SELECT statement is:

SELECT *field list*
FROM *table list*

To select all the fields from a table, you can use an asterisk (*). For example, the following statement selects all the fields and all the records from the customers table:

```
SELECT *
FROM customers
```

ID	company name	contact name	contact title	address
1	Alfreds Futterkiste	Maria Anders	Sales Representative	Obere Str. 57
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Owner	Avda. de la Constitución 2
3	Antonio Moreno Taquería	Antonio Moreno	Owner	Mataderos 2312
4	Around the Horn	Thomas Hardy	Sales Representative	120 Hanover Sq.

To limit the fields retrieved by the query, simply use the field names instead. For example:

```
SELECT [company name], phone, fax
FROM customers
```

company name	phone	fax
Alfreds Futterkiste	030-0074321	030-0076545
Ana Trujillo Emparedados y helados	(5) 555-4729	(5) 555-3745
Antonio Moreno Taquería	(5) 555-3932	(null)
Around the Horn	(171) 555-7788	(171) 555-6750

To designate a different name to a field in the result set, use the AS keyword to establish an alias for that field. The AS keyword is very handy when a field is the result of a function:

```
SELECT ID AS [Customer id]
FROM tblCustomers
```

Customer ID
1
2
3
4

Note that aliases are very handy when some fields are the result of a function: without aliases, JET 4.0 names these fields Expr1000, Expr1001, Expr1002, ...

Keywords for SELECT

The ALL keyword

The ALL keyword is the default keyword that is used when no predicate is declared in an SQL statement. It simply means that all records will be retrieved that match the qualifying criteria of the SQL statement. Returning to our invoices database example, let's select all records from the customers table:

```
SELECT *
FROM customers
```

ID	company name	contact name	contact title	address
1	Alfreds Futterkiste	Maria Anders	Sales Representative	Obere Str. 57
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Owner	Avda. de la Constitución 2
3	Antonio Moreno Taquería	Antonio Moreno	Owner	Mataderos 2312
4	Around the Horn	Thomas Hardy	Sales Representative	120 Hanover Sq.

Notice that although the ALL keyword was not declared, it is the default predicate. We could have written the statement like this:

```
SELECT ALL *
FROM customers
```

ID	company name	contact name	contact title	address
1	Alfreds Futterkiste	Maria Anders	Sales Representative	Obere Str. 57
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Owner	Avda. de la Constitución 2
3	Antonio Moreno Taquería	Antonio Moreno	Owner	Mataderos 2312
4	Around the Horn	Thomas Hardy	Sales Representative	120 Hanover Sq.

The DISTINCT keyword

The DISTINCT keyword is used to control how duplicate values in a result set are handled. Based on the field(s) specified in the field list, the rows that have duplicate values in the specified fields are filtered out. If more than one field is specified, it is the combination of all of the fields that is used as the filter. For example, if you query the Customers table for distinct last names, only the unique names will be returned; any duplicate names will result in only one instance of that name in the result set.

```
SELECT DISTINCT Department
FROM jobs_ident
```

Department
cooling
electricity
heating
painting

It is important to note that the result set returned by a query that uses the DISTINCT keyword cannot be updated; it is read-only.

The DISTINCTROW keyword

The DISTINCTROW keyword is similar to the DISTINCT keyword except that it is based on entire rows, not just individual fields. It is useful only when based on multiple tables, and only when you select fields from some, but not all, of the tables. If you base your query on one table, or select fields from every table, the DISTINCTROW keyword essentially acts as an ALL keyword.

For example, in our SQL Pocket Guide database, every customer can have no invoices, or one or more invoices. Let's suppose that we want to find out how many customers have one or more invoices. We will use the DISTINCTROW keyword to determine our list of customers.

```
SELECT DISTINCTROW [company name]
FROM customers INNER JOIN jobs
ON customers. ID = jobs.Customer
```

company name
Alfreds Futterkiste
Ana Trujillo Emparedados y helados
Antonio Moreno Taquería
Around the Horn

If we had left off the DISTINCTROW keyword, we would have gotten a row returned for every invoice each customer has. (The INNER JOIN statement will be covered in a later section.)

The TOP keyword

The TOP keyword is used to return a certain number of rows that fall at the top or bottom of a range that is specified by an ORDER BY clause. The ORDER BY clause is used to sort the rows in either ascending or descending order. If there are equal values present, the TOP keyword will return all rows that have the equal value. Let's say that we want to determine the highest three invoice amounts in our invoices database. We'd write a statement like this:

```
SELECT TOP 10 invoice_date, amount
FROM jobs
ORDER BY amount DESC
```

invoice_date	amount
02/25/2004	1499.35
10/12/2004	1497.31
10/31/2005	1497.26
08/24/2004	1496.65

We can also use the optional PERCENT keyword with the TOP keyword to return a percentage of rows that fall at the top or bottom of a range, top or bottom (DESC or ASC) is specified by the ORDER BY clause. The code looks like this:

```
SELECT TOP 25 PERCENT invoice_date, amount
FROM jobs
WHERE amount IS NOT NULL
ORDER BY amount ASC
```

invoice_date	amount
05/16/2005	50.34
11/06/2004	51.97
03/14/2006	55.84
02/28/2005	56.27

Note that if you do not specify an ORDER BY clause, the TOP keyword will not be helpful: it will return a random sampling of rows.

Functions and calculated columns

Aside of retrieving data from the database tables, the SELECT statement may also be used to retrieve system data or to retrieve data on which some calculations or functions have been applied.

The table below list a series of functions available through Jet 4.0 SQL.

Table 3: Functions available for Jet 4.0 SQL

Function	Description	Syntax
Text & Strings Functions		
ASC	Returns the character code corresponding to the first letter in a string.	ASC(character) ASC(TextField) In the TextField case returns an Integer representing the character code corresponding to the first letter.
CHR	Returns the character associated with the specified character code.	CHR(charcode)
&	Used to force string concatenation	<i>TextField1 & TextField2 [...& TextFieldN]</i>
LCASE	Returns a String that has been converted to lowercase	LCASE(TextField)
LEFT	Returns String containing a specified number of	LEFT(TextField, length)

	characters from the left side of a string	
LEN	Returns the number of characters in a string	LEN (string)
LTRIM, RTRIM, TRIM	Returns a String containing a copy of a specified string without leading spaces (LTrim), trailing spaces (RTrim), or both leading and trailing spaces (Trim).	LTRIM (TextField) RTRIM (TextField) TRIM (TextField)
MID	Returns a String containing a specified number of characters from a string.	MID (TextField, start[, length]) start : Character position in TextField at which the part to be taken begins. If start is greater than the number of characters in TextField , MID returns a zero-length string (""). length : Number of characters to return. If omitted or if there are fewer than length characters in the text (including the character at start), all characters from the start position to the end of the string are returned.
RIGHT	Returns a String containing a specified number of characters from the right side of a string	RIGHT (TextField, length)
UCASE	Returns a String that has been converted to uppercase	UCASE (TextField)
Numeric Functions		
+	Used to sum two numbers	NumField1 + NumField2
-	Used to find the difference between two numbers or to indicate the negative value of a numeric expression	Syntax 1 NumField1 - NumField2 Syntax 2 - NumField
*	Used to multiply two numbers	NumField1 * NumField2 NumField * Number
/	Used to divide two numbers and return a floating-point result	NumField1 / NumField2 NumField / Number
\	Used to divide two numbers and return an integer result (See also Mod)	NumField1 \ NumField2 NumField \ Number
^	Used to raise a number to the power of an exponent	NumField ^ Number
ABS	Returns the absolute value of a number	ABS (NumField)
EXP	Returns e (the base of natural logarithms) raised to a power	EXP (NumField)
LOG	Returns the natural logarithm of a number	LOG (NumField)
MOD	Used to divide two numbers and return only the remainder	NumField1 MOD NumField2 NumField MOD Number
ROUND	Returns a number rounded to a specified number of decimal places	ROUND (NumField,[NumDecimalPlaces]) NumDecimalPlaces : Optional. A number indicating how many places to the right of the decimal are included in the rounding. If omitted, integers are returned by the Round function
SGN	Returns an Integer indicating the sign of a number	SGN (NumField) -1 = negative 0 = zero

		1 = positive
SQR	Returns the square root of a number	SQR (NumField)
Date & Time Functions		
DATE	Returns the current system date	DATE ()
DATEADD	Returns a date to which a specified time interval has been added	DATEADD (interval ⁽¹⁾ , DateField, date)
DATEDIFF	Returns the number of time intervals between two specified dates	DATEDIFF (interval ⁽¹⁾ , DateField1, DateField2 [, FirstDayOfWeek ⁽²⁾ [, FirstWeekOfYear ⁽³⁾]])
DATEPART	Returns the specified part of a given date	DATEPART (interval ⁽¹⁾ , DateField [, FirstDayOfWeek ⁽²⁾ [, FirstWeekOfYear ⁽³⁾]])
<p>(1) The <i>interval</i> argument may have one of following values:</p> <p>yyyy = year q = quarter m = month y = day of year d = day h = hours in a hh:mm:ss time format n = minutes in a hh:mm:ss time format s = seconds in a hh:mm:ss time format ww = week</p> <p>(2) The <i>FirstDayOfWeek</i> argument may have one of following values:</p> <p>1 = Sunday 2 = Monday 3 = Tuesday 4 = Wednesday 5 = Thursday 6 = Friday 7 = Saturday</p> <p>(3) The <i>FirstWeekOfYear</i> argument may have one of following values:</p> <p>1 = Start with week in which January 1 occurs (default) 2 = Start with the first week that has at least four days in the new year 3 = Start with first full week of the year</p>		
DAY	Returns a whole number between 1 and 31, inclusive, representing the day of the month	DAY (DateField)
MONTH	Returns a whole number between 1 and 12, inclusive, representing the month of the year	MONTH (DateField)
NOW	Returns the current system date	NOW
YEAR	Returns a whole number representing the year	YEAR (DateField)
Data type conversion functions		
CCUR	Coerce data to the Currency data type	CCUR (NumField)
CDATE	Coerce data to the DateTime data type	CDATE (TextField) <i>TextField</i> must represent a date according to the local settings of the system. The correct order of day, month, and year may not be determined if it is provided in a format other than one of the recognized date settings. In addition, a long date format is not recognized if it also contains the day-of-the-week string.
CDBL	Coerce data to the Double data type	CDBL (NumField)
CDEC	Coerce data to the Decimal data type	CDEC (NumField)

CINT	Coerce data to the Short data type	CINT (NumField)
CLNG	Coerce data to the Long data type	CLNG (NumField)
CSNG	Coerce data to the Single data type	CSNG (NumField)
CSTR	Coerce data to the Text data type	CSTR (NumField) CSTR (DateField)
!!! Important: Before you try to convert a field to a specific data type, be sure this field has no records with a null value !!!		
Conditional function		
IIF	Returns one of two parts, depending on the evaluation of an expression	IIF (<i>expr</i> , <i>truepart</i> , <i>falsepart</i>) <i>expr</i> : Expression you want to evaluate. <i>truepart</i> : Value or expression returned if <i>expr</i> is True. <i>falsepart</i> : Value or expression returned if <i>expr</i> is False.

Manipulating numeric fields

We can use **SELECT** to add or multiply fields together, or to add a number to or multiply a field by a number, or subtract or divide one field by an other, ...

```
SELECT
  amount * 1.21 AS [tot amount]
FROM jobs
```

tot amount
259.3272
1775.1668
93.4725
963.3899

```
SELECT
  ROUND(amount * 1.21, 2) AS [tot amount]
FROM jobs
```

tot amount
259.33
1775.17
93.47
963.39

We can use **SELECT** to coerce the data type of one field to a specified data type. First thing to do is to exclude all null values contained in the field.

```
SELECT
  [12345]*1.21 AS [double], CCUR([12345]*1.21) AS [currency]
FROM
  (SELECT [12345]
   FROM ['numbered colhead$']
   WHERE [12345] IS NOT NULL)
```

double	currency
57.94027283	57.9403
34.09595717	34.096
72.28512533	72.2851
31.24989802	31.2499

Manipulating DateTime fields

One of the biggest drawbacks of Jet 4.0 databases, and it's not much better with SQL server databases, is it's very limited ability to work on DateTime data type. In fact both database engines recognize only the most basic DateTime data type, being any combination of the trio Day, Month and Year.

Although not really useful for pbconnect users, it's nice to know you can use **SELECT** not only to retrieve data from a table, but also from the operating system itself as demonstrated by the **DATE()** and **NOW** functions.

SELECT DATE()

Expr1000
08/03/2006

SELECT NOW AS [today_date]

today_date
08/03/2006

Luckily enough, we can use the **DATEPART** and **&** (concatenate) functions in a **SELECT** statement to extract specific intervals out of a date, and concatenate those intervals together to create a string representing other type of DateTime format, like DayOfYear/Year or Quarter/Year. Although the functions' table also mentions Week, this interval is of no use because of the unpredictable results it generates for the first week of the year (See App C, ISO Week Date).

SELECT

```
req_date,
DatePart('y', req_date) & '/' & Datepart('yyyy', req_date) AS [dayofyear/year]
FROM jobs
ORDER BY amount DESC
```

req_date	dayofyear_year
01/08/2004	8/2004
07/24/2004	206/2004
07/20/2005	201/2005
05/07/2004	128/2004

SELECT

```
invoice_date,
DatePart('q', invoice_date) & '/' & Datepart('yyyy', invoice_date) AS quarter_year
FROM jobs
WHERE invoice_date IS NOT NULL
ORDER BY amount DESC
```

invoice_date	quarter_year
02/25/2004	1/2004
10/12/2004	4/2004
10/31/2005	4/2005
08/24/2004	3/2004

Narrowing the scope of a SELECT statement:

the WHERE clause

More often than not, you will not want to retrieve all records from a table. You will want only a subset of those records based on some qualifying criteria. To qualify a SELECT statement, you use a WHERE clause, which will allow you to specify exactly which records you want to retrieve.

```
SELECT ID, amount, invoice_date
FROM jobs
WHERE ID = 156
```

ID	amount	invoice_date
156	953.75	08/09/2004

Note the `job_code = 156` portion of the WHERE clause. A WHERE clause can contain up to 40 such expressions, and they can be joined with the And or Or logical operators. Using more than one expression allows you to further filter out records in the result set.

```
SELECT job_code, amount, invoice_date
FROM jobs
WHERE job_code = 1 AND invoice_date >= #01/01/2005#
ORDER BY customer DESC
```

job_code	amount	invoice_date
1	840.59	02/22/2006
1	648.89	06/26/2006
1	644.33	10/05/2005
1	546.94	10/21/2005



Note that the date string is enclosed in number signs (#). If you are using a regular string in an expression, you must enclose the string in single quotation marks ('). For example:

```
SELECT *
FROM [address list]
WHERE [first name] = 'Etienne'
```

Note that when working in pbconnect, we will most of the time use a variable to filter on a string or a number (being [Comp], [TransComp], [Loc], [TransLoc], [Measure] or [TransMeasure]) and use functions (MONTH(date) = [Month] AND YEAR(date) = [Year]) to filter. It is important to know the data type of the field we are filtering on:

When the data type is Text, the variable is further enclosed in single quotation marks ('). For example:

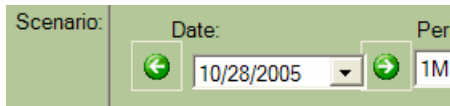
```
SELECT ID, [company name], country
FROM customers#csv
WHERE country = '[TransLoc]'
```

Scenario:	Date:	Period:	Comparative:	Location:	Measure:	Variables:
	 10/28/2005 	1M	Actual	United S		

ID	company name	country
32	Great Lakes Food Market	USA
36	Hungry Coyote Import Store	USA
43	Lazy K Kountry Store	USA
45	Let's Stop N Shop	USA

When we are filtering with the functions DAY(*date*), MONTH(*date*) or YEAR(*date*), the result of the function is of the type Short for DAY and MONTH or Long for YEAR, so the variables [Day], [Lday], [Month], [LMonth], [ShortYear] or [Year] are never enclosed in single quotation marks (').

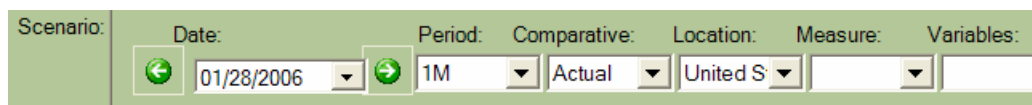
```
SELECT *
FROM jobs#csv
WHERE MONTH(invoice_date) = [Month] AND YEAR(invoice_date) = [Year]
```



job_code	customer	req_date	finish_date	amount	invoice_date
4	27	06/25/2005	08/09/2005	648.59	09/01/2005
2	13	06/18/2005	08/09/2005	455.69	09/07/2005
12	25	06/12/2005	08/19/2005	242.09	09/01/2005
12	29	06/08/2005	08/20/2005	1356.58	09/06/2005

When we filter on fields created with a SELECT statement and the function DATEPART, like for example a field containing a date formatted as "Week/Year", "DayOfYear/Year" or "Quarter/Year", these fields DO NOT have the DateTime data type but are evaluated as Text (Jet 4.0 OLEDB Provider only recognize the combinations day-month-year as DateTime)

```
SELECT *
FROM (
    SELECT amount, invoice_date, DATEPART('q', invoice_date) & '/' &
        DATEPART('yyyy', invoice_date) AS [invdate_byquarter]
    FROM jobs)
WHERE [inv_date by quarter] = '[Quarter]/[Year]'
```



amount	invoice_date	invdate_byquarter
577.39	10/01/2005	4/2005
468.04	10/05/2005	4/2005
788.87	10/06/2005	4/2005
516.05	10/06/2005	4/2005

Note the nested SELECT statement required in this type of query: the second level SELECT retrieves the fields **amount** and **invoice_date** and creates a field **invdate_byquarter** by use of the functions DATEPART and **& (concatenate)**, the first level SELECT retrieves all the fields specified in the second level SELECT either by name or by alias but restricts the result of the query to the combination ' [Quarter]/[Year]' in the field **invdate_byquarter**.

The IN operator

The IN operator is used to determine if the value of an expression is equal to any of several values in a specified list. If the expression matches a value in the list, the IN operator returns **True**. If it is not found, the IN operator returns **False**. Let's suppose that we want to find all jobs with code 16, 17 and 18. Although we could write an SQL statement with a long WHERE clause that uses the OR logical operator, using the IN operator will shorten our statement.

```
SELECT *
FROM jobs
```

WHERE job_code IN (16,17,18)
ORDER BY amount DESC

ID	job_code	customer	req_date	finish_date	amount	invoice_dat
256	18	44	08/17/2004	10/09/2004	1483.32	11/03/2004
470	17	70	03/25/2005	04/24/2005	1483.25	05/24/2005
836	18	45	02/22/2006	03/25/2006	1481.14	04/24/2006
663	17	59	07/29/2005	10/15/2005	1475.04	11/13/2005

SELECT *
FROM jobs
WHERE job_code = 16 OR job_code = 17 OR job_code = 18
ORDER BY amount DESC

ID	job_code	customer	req_date	finish_date	amount	invoice_dat
256	18	44	08/17/2004	10/09/2004	1483.32	11/03/2004
470	17	70	03/25/2005	04/24/2005	1483.25	05/24/2005
836	18	45	02/22/2006	03/25/2006	1481.14	04/24/2006
663	17	59	07/29/2005	10/15/2005	1475.04	11/13/2005

By using the NOT logical operator, we can check the opposite of the IN operator. This statement will return all jobs where the code is *not* 16, 17 or 18:

SELECT *
FROM jobs
WHERE job_code NOT IN (16,17,18)
ORDER BY customer ASC

ID	job_code	customer	req_date	finish_date	amount	invoice_date
358	1	1	12/01/2004	12/31/2004	815.6	01/23/2005
927	12	1	06/17/2006	(null)	(null)	(null)
369	2	1	12/10/2004	01/15/2005	292.29	02/01/2005
896	11	1	03/21/2006	06/01/2006	887.65	06/17/2006

Note that if the job_code field was of data type Text instead of Short, as is the case in the table jobs, you should have used IN ('16','17','18')! So, if you get an error with one of the statements above, try the same statement with the values enclosed in single quote character (remember Excel and its "best guess")!!!

The BETWEEN operator

The BETWEEN operator is used to determine if the value of an expression falls within a specified range of values. If the expression's value falls within the specified range, including both the beginning and ending range values, the BETWEEN operator returns **True**. If the expression's value does not fall within the range, the BETWEEN operator returns **False**. Let's suppose that we want to find all invoices that have an amount between €50 and €100. We'd use the BETWEEN operator in the WHERE clause with the AND keyword that specifies the range.

SELECT *
FROM [jobs\$]
WHERE amount BETWEEN 50 AND 500

job_code	customer	req_date	finish_date	amount	invoice_date
20	25	01/12/2004	01/28/2004	214.32	02/25/2004
1	72	01/18/2004	02/02/2004	77.25	02/22/2004
16	42	01/02/2004	02/05/2004	191.1	02/29/2004
18	62	01/16/2004	02/18/2004	324.22	03/09/2004

You could generate the same result by using the older statement making use only of comparison and logical operators. Although this method returns exactly the same amount of records, it is not recommended, since it executes slower than the BETWEEN operator.

```
SELECT *
FROM [jobs$]
WHERE amount >= 50 AND price <= 500
```

job_code	customer	req_date	finish_date	amount	invoice_date
20	25	01/12/2004	01/28/2004	214.32	02/25/2004
1	72	01/18/2004	02/02/2004	77.25	02/22/2004
16	42	01/02/2004	02/05/2004	191.1	02/29/2004
18	62	01/16/2004	02/18/2004	324.22	03/09/2004

By using the NOT logical operator, we can check the opposite of the BETWEEN operator to find invoice amounts that fall *outside* that range.

```
SELECT *
FROM [jobs$]
WHERE amount NOT BETWEEN 50 AND 500
```

job_code	customer	req_date	finish_date	amount	invoice_date
9	41	01/02/2004	01/31/2004	1467.08	02/17/2004
20	44	01/08/2004	02/03/2004	796.19	02/18/2004
20	81	01/08/2004	02/08/2004	1499.35	02/25/2004
2	24	01/04/2004	02/09/2004	685.36	03/05/2004

Note that the range can be in reverse order and still achieve the same results (BETWEEN 100 AND 50), but many ODBC-compliant databases require that the range follows the begin-value-to-end-value method. If you plan for your application to be scaled or upsized to an ODBC-compliant database, you should always use the begin-value-to-end-value method.

The LIKE operator

The LIKE operator is used to determine if the value of an expression compares to that of a pattern. A *pattern* is either a full string value, or a partial string value that also contains one or more wildcard characters. By using the LIKE operator, you can search a field within a result set and find all of the values that match the specified pattern.

Note that SQL for Jet 4.0 OLEDB Provider is NOT case sensitive, so filtering on 'usa', 'USA' or 'Usa' returns the same result.

```
SELECT id, [company name], country
FROM customers
WHERE country LIKE 'USA'
```

ID	company name	country
32	Great Lakes Food Market	USA
36	Hungry Coyote Import Store	USA
43	Lazy K Kountry Store	USA
45	Let's Stop N Shop	USA

To return all customers who live in a country that starts with the letter F, use the percent wildcard character.

```
SELECT id, [company name], country
FROM customers
WHERE country LIKE 'F%'
ORDER BY id DESC
```

ID	company name	country
90	Wilman Kala	Finland
87	Wartian Herkku	Finland
85	Vins et alcools Chevalier	France
84	Victuailles en stock	France

To return all customers who live in a country that starts with the letter F and counts exactly 6 characters, use 5 underscore wildcard characters after the letter F.

```
SELECT id, [company name], country
FROM customers
WHERE country LIKE 'F_____'
ORDER BY id DESC
```

ID	company name	country
85	Vins et alcools Chevalier	France
84	Victuailles en stock	France
74	Spécialités du monde	France
57	Paris spécialités	France

By using the NOT logical operator, we can check the opposite of the LIKE operator and filter out the USA from the list.

```
SELECT id, [company name], country
FROM customers
WHERE country NOT LIKE 'USA'
```

id	company name	country
1	Alfreds Futterkiste	Germany
2	Ana Trujillo Emparedados y helados	Mexico
3	Antonio Moreno Taquería	Mexico
4	Around the Horn	UK

To return all characters equal to A, B, C or D, use the character list wildcard.

```
SELECT *
FROM characters
WHERE Chr LIKE '[ABCD]'
```

Or the character range wildcard.

```
SELECT *
FROM characters
WHERE Chr LIKE '[A-D]'
```

chr	ascii
A	65
B	66
C	67
D	68
a	97
b	98
c	99
d	100

Note: You might think that using the BETWEEN operator would return the same. This is true ONLY if non-accented characters are used in the table! If using accented characters (lots of southern or central European countries...) then the query return all rows where [Last Name] starts with A, B, C, D, À, Á, Â, Ã, Ä, Å, Æ or Ç or the lowercase of those characters.

```
SELECT *
FROM characters
WHERE Chr BETWEEN 'A' AND 'D'
```

chr	ascii	chr	ascii
A	65	À	196
B	66	Á	197
C	67	Æ	198
D	68	Ç	199
a	97	à	224
b	98	á	225
c	99	â	226
d	100	ã	227
À	192	ä	228
Á	193	å	229
Ä	194	æ	230
Å	195	ç	231

To return all customers who live in a country that starts with the letter A to M, use the character list wildcard followed by the percent wildcard character.

```
SELECT id, [company name], country
FROM customers
WHERE country LIKE '[A-M]%'
```

id	company name	country
1	Alfreds Futterkiste	Germany
2	Ana Trujillo Emparedados y helados	Mexico
3	Antonio Moreno Taquería	Mexico
6	Blauer See Delikatessen	Germany

Would you like to return all customers who live in a country that starts with any letter but A to M, use the character list wildcard character.

```
SELECT id, [company name], country
FROM customers
WHERE country LIKE '[!A-M]%'
```

Or

```
SELECT id, [company name], country
FROM customers
WHERE country NOT LIKE '[A-M]%'
```

id	company name	country
4	Around the Horn	UK
5	Berglunds snabbköp	Sweden
8	Bólido Comidas preparadas	Spain
11	B's Beverages	UK

There are a number of wildcard characters to choose from, and the following table details what they are and what they can be used for.

Table 4: Wildcard characters available for Jet 4.0 SQL

Wildcard character	Description
%	Matches any number of characters and can be used anywhere in the pattern string.
_ (underscore)	Matches any single character and can be used anywhere in the pattern string.
#	Matches any single digit and can be used anywhere in the pattern string.
[char1-char2]	Matches any single character within the list that is enclosed within brackets, and can be used anywhere in the pattern string.
[!char1-char2]	Matches any single character not in the list that is enclosed within the square brackets.

The IS NULL operator

A null value is one that indicates missing or unknown data. Null is neither Text, Number or DateTime data type but is an Object. This is the reason why the IS NULL operator is used to determine if the value of an expression is equal to the null value, and not the numerical comparison operator '=' or the string comparison operator 'LIKE'. The use of '=' or 'LIKE' to find a null value returns an error.

```
SELECT COUNT(werkopdracht)
FROM [dsv zeist#csv]
WHERE [uiterste uitvoerdatum] IS NULL OR [technische gereed] IS NULL OR
[financieel gereed] IS NULL
```

Expr1000
2625

By adding the NOT logical operator, we can check the opposite of the IS NULL operator. In this case, the statement will weed out invoices with missing or unknown amounts.

```
SELECT COUNT(werkopdracht)
FROM [dsv zeist#csv]
WHERE boekdatum IS NOT NULL
```

Expr1000
9533

Sorting the Result Set

Although not particularly important when working with pbconnect, you might wish for any reason to specify a particular sort order on one or more fields in the result set: use the optional ORDER BY clause. Records can be sorted in either ascending (ASC) or descending (DESC) order; ascending is the default.

Fields referenced in the ORDER BY clause do not have to be part of the SELECT statement's field list, and sorting can be applied to string, numeric, and date/time values. Always place the ORDER BY clause at the end of the SELECT statement.

```
SELECT *
FROM [address list#csv]
ORDER BY [last name], [first name] DESC
```

ID	first name	last name	address	postal code	city	phone	mobile	last earning
6	Maurice	Berden	Beversestr	5431SH	Cuijk	31 (0)485 3	(null)	115.28
3	Walter	Haazen	Kretenburg	2640	Mortsel	32 (0)3 454	32 (0)495 5	137.48
4	Bea	Haazen	Volharding	2650	Edegem	(null)	32 (0)497 4	52.87
5	Etienne	Magnée	rue Beau S	4540	Amay	32 (085) 31	32 (0)496 2	259.96

You can also use the field numbers (or positions) instead of field names in the ORDER BY clause.

```
SELECT *
FROM ['address list$']
ORDER BY 2, 3 DESC
```

ID	first name	last name	address	postal code	city	phone	mobile	last earning
1	Ala	Owaidah	Rawdah St	21422	Jeddah	966 (0)2 66	966 (0)50 4	125.39
4	Bea	Haazen	Volharding	2650	Edegem	(null)	32 (0)497 4	52.87
5	Etienne	Magnée	rue Beau S	4540	Amay	32 (085) 31	32 (0)496 2	259.96
6	Maurice	Berden	Beversestr	5431SH	Cuijk	31 (0)485 3	(null)	115.28

Using Aggregate Functions to Work with Values

Aggregate functions are used to calculate statistical and summary information from data in tables. These functions are used in SELECT statements, and all of them take fields or expressions as arguments.

To count the number of records in a result set, use the **Count** function. Using an asterisk with the **Count** function causes **Null** values to be counted as well.

```
SELECT COUNT(*) AS [Number of Invoices]
FROM jobs
```

Number of Invoices
1000

To count only non-Null values, use the **Count** function with a field name:

```
SELECT COUNT(amount) AS [Number of Valid Invoice Amounts]
FROM jobs
```

Number of Valid Invoice Amounts
902

To find the average value for a field or expression of numeric data, use the **AVG** function:

```
SELECT AVG(amount) AS [Average Invoice Amount]
FROM jobs
```

Average Invoice Amount
779.2446

To find the total of the values in a field or expression of numeric data, use the **Sum** function:

```
SELECT SUM(amount) AS [Total Invoice Amount]
FROM Jobs
```

Total Invoice Amount
702878.64

To find the minimum value for a field or expression, use the **Min** function:

```
SELECT MIN(amount) AS [Minimum Invoice Amount]
FROM jobs
```

Minimum Invoice Amount
50.34

To find the maximum value for a field or expression, use the **Max** function:

```
SELECT MAX(amount) AS [Maximum Invoice Amount]
FROM jobs
```

Maximum Invoice Amount
1499.35

Table 5: Aggregate functions available for Jet 4.0 SQL

Function	Description	Syntax
AVG	Calculates the arithmetic mean of a set of values contained in a specified field on a query	AVG(NumField)
COUNT	Calculates the number of records returned by a query	COUNT(Field) <i>Field may be of any datatype including Text or DateTime.</i>
FIRST - LAST	Return a field value from the first or last record in the result set returned by a query	FIRST(NumField) LAST(NumField)
MIN - MAX	Return the minimum or maximum of a set of values contained in a specified field on a query	MIN(NumField) MAX(NumField)
STDEV - STDEVP	Return estimates of the standard deviation for a population or a population sample represented as a set of values contained in a specified field on a query Use STDEV for a population sample and STDEVP for a population	STDEV(NumField) STDEVP(NumField)
SUM	Returns the sum of a set of values contained in a specified field of fields on a query	SUM(NumField) SUM(NumField+NumField+NumField)
VAR - VARP	Return estimates of the variance for a population or a population sample represented as a set of values contained in a specified field on a query Use VAR for a population sample and VARP for a population	VAR(NumField) VARP(NumField)

Grouping Records in a Result Set

Sometimes there are records in a table that are logically related, as in the case of the jobs table. Since one type of job can generate many invoices, it could be useful to treat all the invoices for one type of job as a group, in order to find statistical and summary information about the group.

The key to grouping records is that one or more fields in each record must contain the same value for every record in the group. In the case of the jobs table, the job_code field value is the same for every invoice a particular job_code generates.

To create a group of records, use the GROUP BY clause with the name of the field or fields you want to group with.

```
SELECT job_code, COUNT(*) AS [Number of Invoices],
      AVG(amount) AS [Average Invoice Amount]
FROM jobs
GROUP BY job_code
```

job_code	Number of Invoices	Average Invoice Amount
1	56	800.390535714286
2	62	773.169677419355
3	52	749.957115384615
4	50	722.0414
5	43	812.02488372093
6	50	870.7052

Note that the statement will return one record that shows the job_code, the number of invoices per job_code, and the average invoice amount per job_code that has generated an invoice record in the jobs table. Because each job's invoices are treated as a group, we are able to count the number of invoices, and then determine the average invoice amount.

You can specify a condition at the group level by using the HAVING clause, which is similar to the WHERE clause. For example, the following query returns only those records for each job_code whose average invoice amount is higher than 800, this time rounding up the amount to 2 digits right of the decimal point:

```
SELECT job_code, COUNT(*) AS [Number of Invoices],
      ROUND(AVG(amount),2) AS [Average Invoice Amount]
FROM jobs
GROUP BY job_code
HAVING Avg(amount) > 800
```

job_code	Number of Invoices	Average Invoice Amount
1	56	800.39
5	43	812.02
6	50	870.71
9	49	847.62
13	43	845.96
14	57	817.53

Joins

In a relational database system like Access, you will often need to extract information from more than one table at a time. This can be accomplished by using an SQL *JOIN statement*. A JOIN statement enables you to retrieve records from tables that have defined relationships, whether they are one-to-one, one-to-many, or many-to-many.

INNER JOINS

The INNER JOIN, also known as an *equi-join*, is the most commonly used type of join. This join is used to retrieve rows from two or more tables by matching a field value that is common between the tables. The fields you join on must have similar data types, and you cannot join on MEMO or OLEOBJECT data types. To build an INNER JOIN statement, use the INNER JOIN keywords in the FROM clause of a SELECT statement. Let's use the INNER JOIN to build a result set of all customers who have invoices, plus the dates and amounts of those invoices.

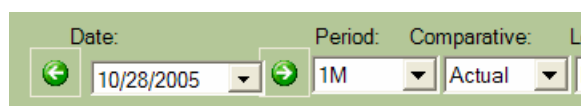
```
SELECT department, invoice_date, price  
FROM jobs_ident INNER JOIN jobs  
ON jobs_ident.job_code=jobs.job_code  
ORDER BY invoice_date
```

department	invoice_date	price
ventilation	02/17/2004	1467.08
telecommunication	02/18/2004	796.19
painting	02/22/2004	77.25
telecommunication	02/25/2004	214.32
telecommunication	02/25/2004	1499.35
sanitary	02/29/2004	191.1
painting	03/02/2004	774.98
cooling	03/04/2004	1114.15

Notice that the table names are divided by the INNER JOIN keywords and that the relational comparison is after the ON keyword. For the relational comparisons, you can also use the <, >, <=, >=, or <> operators, and you can also use the BETWEEN keyword. Also note that the job_code fields from both tables are used only in the relational comparison, they are not part of the final result set.

To further qualify the SELECT statement, we can use a WHERE clause after the join comparison in the ON clause. In the following example, we have narrowed the result set to include only invoices for a defined Month/Year combination.

```
SELECT department, invoice_date, price  
FROM jobs_ident INNER JOIN jobs  
ON jobs_ident.job_code=jobs.job_code  
WHERE MONTH(invoice_date)=[LMonth] AND YEAR(invoice_date)=[Year]  
ORDER BY invoice_date
```



department	invoice_date	price
electricity	09/01/2005	242.09
heating	09/01/2005	648.59
electricity	09/06/2005	1356.58
telecommunication	09/07/2005	367.25
painting	09/07/2005	455.69
telecommunication	09/08/2005	630.61
cooling	09/10/2005	843.02
heating	09/10/2005	637.54

In cases where you need to join more than one table, you can nest the INNER JOIN clauses. In this example, we will build on a previous SELECT statement to create our result set, but we will also include the city and state of each customer by adding the INNER JOIN for the tblCustomers table.

```
SELECT job_name, [company name], amount, invoice_date
FROM (tblCustomers INNER JOIN jobs
ON tblCustomers.ID=jobs.customer)
INNER JOIN jobs_ident
ON jobs_ident.job_code=jobs.job_code
ORDER BY invoice_date DESC
```

job_name	company name	amount	invoice_date
ventilation repair	Die Wandernde Kuh	536.4	06/30/2006
painting exterior	The Big Cheese	397.11	06/29/2006
telecommunication installation	GROSELLA-Restaurante	1349.55	06/28/2006
electricity installation	Alfreds Futterkiste	598.84	06/27/2006

Note that the first JOIN clause is enclosed in parentheses to keep it logically separated from the second JOIN clause. It is also possible to join a table to itself by using an alias for the second table name in the FROM clause. Let's suppose that we want to find all customer records that have duplicate last names. We do this by creating the alias "A" for the second table and checking for first names that are different.

```
SELECT tblCustomers.[Last Name],
tblCustomers.[First Name]
FROM tblCustomers INNER JOIN tblCustomers AS A
ON tblCustomers.[Last Name]=A.[Last Name]
WHERE tblCustomers.[First Name]<>A.[First Name]
ORDER BY tblCustomers.[Last Name]
```

OUTER JOINS

The OUTER JOIN is used to retrieve records from multiple tables while preserving records from one of the tables, even if there is no matching record in the other table. There are two types of OUTER JOINS that the Jet database engine supports: LEFT OUTER JOINS and RIGHT OUTER JOINS. Think of two tables that are beside each other, a table on the left and a table on the right. The LEFT OUTER JOIN will select all rows in the right table that match the relational comparison criteria, and it will also select all rows from the left table, *even if no match exists in the right table*. The RIGHT OUTER JOIN is simply the reverse of the LEFT OUTER JOIN; all rows in the right table are preserved instead.

As an example, let's suppose that we want to determine the total amount invoiced to each customer, but if a customer has no invoices, we want to show it by displaying the word "NONE."

```
SELECT [company name],
IIF(Sum(Amount) IS NULL,'NONE',Sum(Amount)) AS Total
FROM customers LEFT OUTER JOIN jobs
ON customers.ID=jobs.customer
GROUP BY [company name]
```

There are a few things going on in the previous SQL statement. The first is the use of the string concatenation operator "&". This operator allows you to join two or more fields together as one string. The second is the *immediate if* (IIF) statement, which checks to see if the total is null. If it is, the statement returns the word "NONE." If the total is not null, the value is returned. The final thing is the OUTER JOIN clause. Using the LEFT OUTER JOIN preserves the rows in the left table so that we see all customers, even those who do not have invoices.

OUTER JOINS can be nested inside INNER JOINS in a multi-table join, but INNER JOINS cannot be nested inside OUTER JOINS.

The Cartesian product

A term that often comes up when discussing joins is the *Cartesian product*. A Cartesian product is defined as "all possible combinations of all rows in all tables." For example, if you were to join two tables without any kind of qualification or join type, you would get a Cartesian product.

```
SELECT *  
FROM customers, jobs
```

This is not a good thing, especially with tables that contain hundreds or thousands of rows. You should avoid creating Cartesian products by always qualifying your joins.

The UNION operator

Although the UNION operator, also known as a *union query*, is not technically a join, it is included here because it does involve combining data from multiple sources of data into one result set, which is similar to some types of joins. The UNION operator is used to splice together data from tables, SELECT statements, or queries, while leaving out any duplicate rows. Both data sources must have the same number of fields, but the fields do not have to be the same data type. Let's suppose that we have an Employees table that has the same structure as the Customers table, and we want to build a list of names and e-mail address by combining both tables.

```
SELECT [Last Name], [First Name], Email  
FROM customers  
UNION  
SELECT [Last Name], [First Name], Email  
FROM tblEmployees
```

If we wanted to retrieve all fields from both tables, we could use the TABLE keyword, like this:

```
TABLE tblCustomers  
UNION  
TABLE tblEmployees
```

The UNION operator will not display any records that are exact duplicates in both tables, but this can be overridden by using the ALL predicate after the UNION keyword, like this:

```
SELECT [Last Name], [First Name], Email  
FROM tblCustomers  
UNION ALL  
SELECT [Last Name], [First Name], Email  
FROM tblEmployees
```

The TRANSFORM statement

Although the TRANSFORM statement, also known as a *crossstab query*, is also not technically considered to be a join, it is included here because it does involve combining data from multiple sources of data into one result set, which is similar to some types of joins.

A TRANSFORM statement is used to calculate a sum, average, count, or other type of aggregate total on records. It then displays the information in a grid or spreadsheet format with data grouped both vertically (rows) and horizontally (fields). The general form for a TRANSFORM statement is this:

```
TRANSFORM aggregating function  
SELECT statement  
PIVOT field heading field
```

Let's suppose that we want to build a datasheet that displays the invoice totals for each customer on a year-by-year basis. The vertical headings will be the customer names, and the horizontal headings will be the years. Let's modify a previous SQL statement to fit the transform statement.

TRANSFORM

IIF(Sum([Amount]) IS NULL,'NONE',Sum([Amount]))

AS Total

SELECT [company name]

FROM customers LEFT JOIN jobs

ON customers.ID=jobs.customer

GROUP BY [company name]

PIVOT Format(Invoice_date, 'yyyy')

IN ('2004','2005','2006')

Note that the aggregating function is the SUM function, the vertical headings are in the GROUP BY clause of the SELECT statement, and the horizontal headings are determined by the field listed after the PIVOT keyword.

Appendix A - Reserved words

A	DAY	LOGICAL, LOGICAL1	SIZE
ABSOLUTE	DEC, DECIMAL	LONG	SMALLDATETIME
ADD	DECLARE	LONGBINARY	SMALLINT
ADMINDB	DELETE	LONGCHAR	SMALLMONEY
ALL	DESC	LONGTEXT	SOME
ALPHANUMERIC	DISALLOW	LOWER	SPACE
ALTER	DISCONNECT	MATCH	SQL, SQLCODE
ALTER TABLE	DISTINCT	MAX	SQLERROR, SQLSTATE
AND	DISTINCTROW	MEMO	STDEV
ANY	DOMAIN	MIN	STDEVP
ARE	DOUBLE	MINUTE	STRING
AS	DROP	MOD	SUBSTRING
ASC		MONEY	SUM
ASSERTION	E	MONTH	SYSNAME
AUTHORIZATION	EQV		SYSTEM_USER
AUTOINCREMENT	EXCLUSIVECONNECT	N-P	
AVG	EXEC, EXECUTE	NATIONAL	T-Z
	EXISTS	NCHAR	TABLE
B	EXTRACT	NONCLUSTERED	TABLEID
BEGIN		NOT	TEMPORARY
BETWEEN	F-H	NTEXT	TEXT
BINARY	FALSE	NULL	TIME
BIT	FETCH	NUMBER	TIMESTAMP
BIT_LENGTH	FIRST	NUMERIC	TIMEZONE_HOUR
BOOLEAN	FLOAT, FLOAT8	NVARCHAR	TIMEZONE_MINUTE
BOTH	FLOAT4	OCTET_LENGTH	TINYINT
BY	FOREIGN	OLEOBJECT	TO
BYTE	FROM	ON	TOP
	GENERAL	OPEN	TRAILING
C	GRANT	OPTION	TRANSACTION
CASCADE	GROUP	OR	TRANSFORM
CATALOG	GUID	ORDER	TRANSLATE
CHAR, CHARACTER	HAVING	OUTER	TRANSLATION
CHAR_LENGTH	HOOR	OUTPUT	TRIM
CHARACTER_LENGTH		OWNERACCESS	TRUE
CHECK	I	PAD	UNION
CLOSE	IDENTITY	PARAMETERS	UNIQUE
CLUSTERED	IEEEDOUBLE	PARTIAL	UNIQUEIDENTIFIER
COALESCE	IEEESINGLE	PASSWORD	UNKNOWN
COLLATE	IGNORE	PERCENT	UPDATE
COLLATION	IMAGE	PIVOT	UPDATEIDENTITY
COLUMN	IMP	POSITION	UPDATEOWNER
COMMIT	IN	PRECISION	UPDATESECURITY
COMP, COMPRESSION	INDEX	PREPARE	UPPER
CONNECT	INDEXCREATEDB	PRIMARY	USAGE
CONNECTION	INNER	PRIVILEGES	USER
CONSTRAINT	INPUT	PROC, PROCEDURE	USING
CONSTRAINTS	INSENSITIVE	PUBLIC	VALUE
CONTAINER	INSERT	Q-S	VALUES
CONTAINS	INT, INTEGER, INTEGER4	REAL	VAR
CONVERT	INTEGER1, INTEGER2	REFERENCES	VARBINARY
COUNT	INTERVAL	RESTRICT	VARCHAR
COUNTER	INTO	REVOKE	VARP
CREATE	IS	RIGHT	VARYING
CURRENCY	ISOLATION	ROLLBACK	VIEW
CURRENT_DATE		SCHEMA	WHEN
CURRENT_TIME	J-M	SECOND	WHENEVER
CURRENT_TIMESTAMP	JOIN	SELECT	WHERE
CURRENT_USER	KEY	SELECTSCHEMA	WITH
CURSOR	LANGUAGE	SELECTSECURITY	WORK
D	LAST	SET	XOR
DATABASE	LEFT	SHORT	YEAR
DATE	LEVEL	SINGLE	YESNO
DATETIME	LIKE		ZONE

Appendix B - Schema.ini File (Text File Driver)

When the Text driver is used, the format of the text file is determined by using a schema information file. The schema information file, which is always named Schema.ini and always kept in the same directory as the text data source, provides the IISAM with information about the general format of the file, the column name and data type information, and a number of other data characteristics. A Schema.ini file is always required for accessing fixed-length data; you should use a Schema.ini file when your text table contains DateTime, Currency, or Decimal data or any time you want more control over the handling of the data in the table.

Note The Text ISAM will obtain initial values from the registry, not from Schema.ini. The same default file format applies to all new text data tables. All files created by the CREATE TABLE statement inherit those same default format values, which are set by selecting file format values in the **Define Text Format** dialog box with <default> chosen in the **Tables** list. If the values in the registry are different from the values in Schema.ini, the values in the registry will be overwritten by the values from Schema.ini.

Understanding Schema.ini Files

Schema.ini files provide schema information about the records in a text file. Each Schema.ini entry specifies one of five characteristics of the table:

1. The text file name
2. The file format
3. The field names, widths, and types
4. The character set
5. Special data type conversions

The following sections discuss these characteristics.

1. Specifying the File Name

The first entry in Schema.ini is always the name of the text source file enclosed in square brackets. The following example illustrates the entry for the file Sample.txt:

[Sample.txt]

2. Specifying the File Format

The **Format** option in Schema.ini specifies the format of the text file. The Text IISAM can read the format automatically from most character-delimited files. You can use any single character as a delimiter in the file except the double quotation mark ("). The **Format** setting in Schema.ini overrides the setting in the Windows Registry on a file-by-file basis. The following table lists the valid values for the **Format** option.

Format specifier	Table format	Schema.ini Format statement
Tab Delimited	Fields in the file are delimited by tabs.	Format=TabDelimited
CSV Delimited	Fields in the file are delimited by commas (comma-separated values).	Format=CSVDelimited
Custom Delimited	Fields in the file are delimited by any character you choose to input into the dialog box. All except the double quote (") are allowed, including blank.	Format=Delimited(<i>custom character</i>) -or- With no delimiter specified: Format=Delimited()
Fixed Length	Fields in the file are of a fixed length.	Format=FixedLength

3. Specifying the Fields

You can specify field names in a character-delimited text file in two ways:

Include the field names in the first row of the table and set **ColNameHeader** to **True**.

Specify each column by number and designate the column name and data type.

You must specify each column by number and designate the column name, data type, and width for fixed-length files.

Note The **ColNameHeader** setting in Schema.ini overrides the **FirstRowHasNames** setting in the Windows Registry on a file-by-file basis.

The data types of the fields can also be determined. Use the **MaxScanRows** option to indicate how many rows should be scanned when determining the column types. If you set **MaxScanRows** to **0**, the entire file is scanned. The **MaxScanRows** setting in Schema.ini overrides the setting in the Windows Registry on a file-by-file basis.

The following entry indicates that Microsoft Jet should use the data in the first row of the table to determine field names and should examine the entire file to determine the data types used:

ColNameHeader=True
MaxScanRows=0

Do NOT use **MaxScanRows** if you intend to explicitly define your fields (columns) !

The next entry designates fields in a table **without column header** by using the column number (**Coln**) option, which is optional for character-delimited files and required for fixed-length files. The example shows the Schema.ini entries for two fields, a 10-character *CustomerNumber* text field and a 30-character *CustomerName* text field:

Col1=CustomerNumber Text Width 10
Col2=CustomerName Text Width 30

The syntax of **Coln** is:

Coln=ColumnName type [Width #]

The following table describes each part of the **Coln** entry.

Parameter	Description
<i>ColumnName</i>	The text name of the column. If the column name contains embedded spaces, you must enclose it in double quotation marks.
<i>type</i>	Data types are: Microsoft Jet data types Bit Byte (<i>positive integer</i> with value ranging from 0 to 255) Short (<i>integer</i> with value ranging from -32768 through 32767) Long (<i>integer</i> with value ranging from -9223372036854775808 through 9223372036854775807) Currency : never to be used in pbconnect !!! Single (single-precision <i>floating-point numbers</i> ranging in value from -3.4028235E+38 through -1.401298E-45 for negative values and from 1.401298E-45 through 3.4028235E+38 for positive values. Single-precision numbers store an approximation of a real number.) Double (double-precision <i>floating-point numbers</i> ranging in value from -1,79769313486231570E+308 through -4,94065645841246544E-324 for negative values and from 4,94065645841246544E-324 through

	<p>1,79769313486231570E+308 for positive values. Double-precision numbers store an approximation of a real number.) - Always use this type for currency values (more precision than single).</p> <p>DateTime</p> <p>Text</p> <p>Memo : to be used in pbconnect ONLY if the field contains more than 255 characters (commentary table...)</p> <p>ODBC data types</p> <p>Char (same as Text)</p> <p>Float (same as Double)</p> <p>Integer (same as Short)</p> <p>LongChar (same as Memo)</p> <p>Date <i>date format</i></p>
--	--

4. Selecting a Character Set

You can select from two character sets: ANSI and OEM. The **CharacterSet** setting in Schema.ini overrides the setting in the Windows Registry on a file-by-file basis. The following example shows the Schema.ini entry that sets the character set to ANSI:

CharacterSet=ANSI

5. Specifying Data Type Formats and Conversions

The Schema.ini file contains a number of options that you can use to specify how data is converted or displayed. The following table lists each of these options.

Option	Description
DateTimeFormat	Can be set to a format string indicating dates and times. You should specify this entry if all date/time fields in the import/export are handled with the same format. All Microsoft Jet formats except A.M. and P.M. are supported. In the absence of a format string, the Windows Control Panel short date picture and time options are used.
DecimalSymbol	Can be set to any single character that is used to separate the integer from the fractional part of a number.
NumberDigits	Indicates the number of decimal digits in the fractional portion of a number.
NumberLeadingZeros	Specifies whether a decimal value less than 1 and greater than -1 should contain leading zeros; this value can either be False (no leading zeros) or True.

Note1 If you omit an entry, the default value in the Windows Control Panel is used.

Note2 All currency formats have been deleted as this data type is NOT accepted by pbconnect.

DateTimeFormat='d-m-yyyy'
decimalSymbol=','
NumberDigits=2

Appendix C - ISO Week Date

From Wikipedia, the free encyclopedia

The **ISO week date** system is a [leap week calendar](#) system that is part of the [ISO 8601](#) date and time standard. The system is used (mainly) in [government](#) and [business](#) for [fiscal years](#), as well as in timekeeping.

The system uses the same cycle of 7 weekdays as the [Gregorian calendar](#). Weeks start with Monday. ISO years have a [year numbering](#) which is approximately the same as the Gregorian years, but not exactly (see below). An **ISO year** has 52 or 53 full weeks (364 or 371 days). The extra week is called a leap week, a year with such a week a [leap year](#).

A date is specified by the ISO year in the format YYYY, a **week number** in the format ww prefixed by the letter W, and the **weekday number**, a digit d from 1 through 7, beginning with Monday and ending with Sunday. For example, 2006-W52-7 (or in its most compact form 06W527) is the Sunday of the 52nd week of 2006. In the Gregorian system this day is called 31 December 2006.

The system has a 400-year cycle of 146097 days (20871 weeks), with an average year length of exactly 365,2425 days, just like the Gregorian calendar. Since non-leap years have 52 weeks, in every 400 years there are 71 leap years.

Relation with the Gregorian calendar

The ISO year number deviates from the number of the Gregorian year on, if applicable, a Friday, Saturday, and Sunday, or a Saturday and Sunday, or just a Sunday, at the start of the Gregorian year (which are at the end of the previous ISO year) and a Monday, Tuesday and Wednesday, or a Monday and Tuesday, or just a Monday, at the end of the Gregorian year (which are in week 01 of the next ISO year). In the period 4 January–28 December and on all Thursdays the ISO year number is always equal to the Gregorian year number.

Other week numbering systems

For an overview of week numbering systems see [week number](#). The US system has weeks from Sunday through Saturday, and partial weeks at the beginning and the end of the year. An advantage is that no separate year numbering like the ISO year is needed, while correspondence of lexicographical order and chronological order is preserved.

See [Weeknumber grouped by definition](#)

ISO leap years between 1900 en 2299 (one 400 years cycle)

1903	1908	1914	1920	1925	1931	1936	1942	1948	1953
1959	1964	1970	1976	1981	1987	1992	1998	2004	2009
2015	2020	2026	2032	2037	2043	2048	2054	2060	2065
2071	2076	2082	2088	2093	2099	2105	2111	2116	2122
2128	2133	2139	2144	2150	2156	2161	2167	2172	2178
2184	2189	2195	2201	2207	2212	2218	2224	2229	2235
2240	2246	2252	2257	2263	2268	2274	2280	2285	2291
2296	2303	2308	2314	2325	2329	2335	2340	2346	2352

Converting a date to a week date (ISO & US)

To convert any date from 2003 to 2023 to a ISO or US week date, join the table you are using to the date_to_weekdate table.

```
SELECT (SUM(amount)
FROM (jobs INNER AS J JOIN date_to_weekdate AS D
ON J.invoice_date=D.DATE_DMY)
WHERE ISO_WEEKDATE LIKE '[Year]-[LWeek]-_')
```