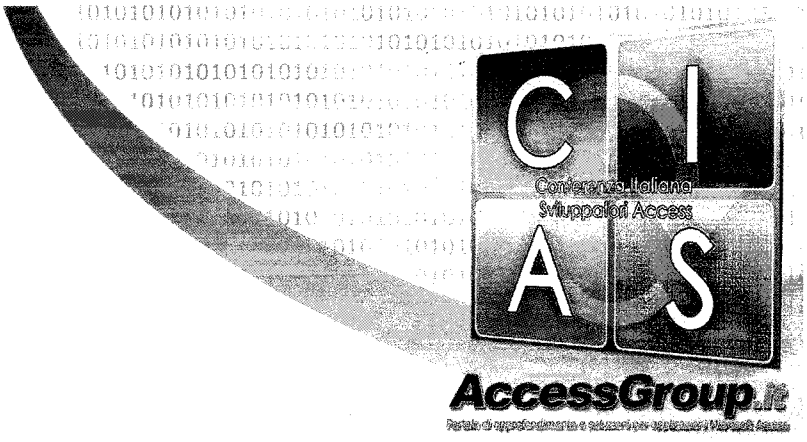

Giorgio Rancati

Mdb/Accdb e Microsoft SQL Server via Odbc

Giorgio Rancati
giorgio.rancati@accessgroup.it

Introduzione

In questa sessione analizzeremo come Access gestisce le tabelle allegate a Microsoft Sql Server via ODBC, l'analisi dei comandi inviati da Access a Sql Server, ci permetterà di capire cosa succede quando interroghiamo una tabella o quando inseriamo/variamo/cancelliamo un record.

La maggior causa del degrado di prestazioni si verifica quandono si eseguono query su tabelle di grosse dimensioni e quando il database engine di Access decide di non inviare l'intera frase Sql al server spezzandola in due o più select per poi eseguire le join in locale, per questo e altri motivi la conoscenza delle interazioni tra Access e Sql server ci aiuterà ad ottimizzare la nostra applicazione spostando la maggior parte delle elaborazioni lato server ottenendo così un minor traffico di rete e di elaborazioni locali.
Ci sono 2 strumenti che ci permettono di vedere i comandi inviati da Access al server.
Il primo, Sql Profier, è un tool incluso nella versione Developer e nelle versioni a pagamento di Sql Server 2005 dalla standard in su. Il secondo strumento è attivabile modificando l'opzione TraceSQLMode nel registro di configurazione di windows nella chiave HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Jet\4.0\Engines\ODBC per Jet4 e nella chiave HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Office\12.0\Access Connectivity Engine\Engines\ODBC per il nuovo Access2007.
Questo secondo sistema non è molto comodo perché le interrogazioni che access invia a Sql Server vengono scritte nel file sqlout.txt che viene creato nel percorso restituito dalla funzione VBA CurDir(), per leggerlo bisogna aprirlo con un editor di testo, ad esempio Notepad che ovviamente ci mostra una situazione statica con i dati del momento, inoltre nel file sqlout.txt, vengono registi i comandi nel momento del loro invio quindi non possiamo sapere quando l'eleborazione viene terminata da Sql Server.

Configurazione del driver ODBC

Quando apriamo una tabella, per impostazione predefinita vengono lette 10 righe ogni 10 secondi fino al raggiungimento di 100 righe, questo comportamento è modificabile creando la tabella **MsysConf** nel database posto sul server con la seguente struttura.

Column Name	Datatype	Description
Config	SMALLINT	The number of the configuration option
chValue	VARCHAR(255)	The text value of the configuration option
nValue	INTEGER	The integer value of the configuration option
Comment	VARCHAR(255)	A description of the configuration option

Config	nValue	Meaning
101	0	Don't allow storing user & password in linked tables
101	1	Allow storing user & password in linked tables (the default)
102	D	Access delays D seconds between each background chunk fetch (default=10)
103	N	Access fetches N rows on each background chunk fetch (default=100)

Oltre alla modifica del numero di records e del tempo che intercorre tra una lettura e l'altra, è possibile inserire un vincolo per impedire la memorizzazione dello user e della password nella proprietà connection delle linked table. Maggiori informazioni sono reperibili scaricando il documento "Jet Database Engine ODBC Connectivity" a questo link http://support.microsoft.com/kb/128385/EN-US/

Odbc DataType

La tabella sottostante mostra i tipi di dato standard Odbc e la corrispondenza per Jet 3.5 e Jet 4

ODBC to Jet Data Type Mappings

ODBC SQL Type	Precision	Scale	Jet 3.5 Type	Jet 40 Type
SQL_BIT	N/A	N/A	Boolean	Boolean
SQL_TINYINT	N/A	N/A	Byte*	Byte*
SQL_TINYINT	N/A	N/A	Integer*	Integer*
SQL_SMALLINT	N/A	N/A	Integer	Integer
SQL_INTEGER	N/A	N/A	Long	Long
SQL_REAL	N/A	N/A	Single	Single
SQL_FLOAT	N/A	N/A	Double	Double
SQL_DOUBLE	N/A	N/A	Double	Double
SQL_DECIMAL	0 To 4	0	Integer	Decimal
SQL_DECIMAL	5 To 9	0	Long	Decimal
SQL_DECIMAL	10 to 15	0	Double	Decimal
SQL_DECIMAL	<=15	>0	Double	Decimal
SQL_DECIMAL	16 To 28	N/A	Text	Decimal
SQL_DECIMAL	> 28	N/A	Text	Text
SQL_NUMERIC	0 To 4	0	Integer	Decimal
SQL_NUMERIC	5 To 9	0	Long	Decimal
SQL_NUMERIC	10 to 15	0	Double	Decimal
SQL_NUMERIC	<=15	>0	Double	Decimal
SQL_NUMERIC	16 To 28	N/A	Text	Decimal
SQL_NUMERIC	> 28	N/A	Text	Text
SQL_CHAR	<= 255	N/A	Text	Text
SQL_CHAR	> 255	N/A	Memo	Memo
SQL_VARCHAR	<= 255	N/A	Text	Text
SQL_VARCHAR	> 255	N/A	Memo	Memo
SQL_LONGVARCHAR	N/A	N/A	Memo	Memo
SQL_WCHAR	<= 255	N/A	Unsupported	Text
SQL_WCHAR	> 255	N/A	Unsupported	Memo
SQL_WVARCHAR	<= 255	N/A	Unsupported	Text
SQL_WVARCHAR	> 255	N/A	Unsupported	Memo
SQL_WLONGVARCHAR	N/A	N/A	Unsupported	Memo
SQL_DATE	N/A	N/A	DateTime	DateTime
SQL_TIME	N/A	N/A	DateTime	DateTime
SQL_TIMESTAMP	N/A	N/A	DateTime	DateTime
SQL_BINARY	<=255	N/A	Binary	Binary
SQL_BINARY	256 To 510	N/A	LongBinary	Binary
SQL_BINARY	> 510	N/A	LongBinary	LongBinary
SQL_VARBINARY	<=255	N/A	Binary	Binary
SQL_VARBINARY	256 To 510	N/A	LongBinary	Binary
SQL_VARBINARY	> 510	N/A	LongBinary	LongBinary
SQL_LONGVARBINARY	N/A	N/A	LongBinary	LongBinary
SQL_GUID	N/A	N/A	Text	Guid

```
* An unsigned SQL_TINYINT maps to a Jet Byte,
a signed SQL_TINYINT maps to an Jet Integer.


Special ODBC-to-Jet Data Type Mappings For SQL Server
ODBC SQL Type     Precision  Scale Jet 3.5 Type Jet 40 Type
-----------------------------------------------------------
SQL_DECIMAL       10         4     Currency     Currency

SQL_DECIMAL       19         4     Currency     Currency
SQL_NUMERIC       10         4     Currency     Currency
SQL_NUMERIC       19         4     Currency     Currency
```

INFO: Improved ODBC DataType Mappings with Jet 4.0
http://support.microsoft.com/kb/214854/en-us

Creiamo una nuova tabella, inseriamo i vari data type di Sql Server 2005, colleghiamola al nostro db tramite il driver Sql Server e vediamo le differenze di interpretazione tra Jet 4 e il nuovo motore di Access2007

Nome Campo	Tipo di dato Sql Server 2005	Tipo di dato Jet4	Dimensione	Tipo di dato Acc2007	Dimensione
C_Bit	Bit	Sì/No		Sì/No	
C_Tinyint	Tinyint	Numerico	Byte	Numerico	Byte
C_Smallint	Smallint	Numerico	Intero	Numerico	Intero
C_Int	Int	Numerico	Intero lungo	Numerico	Intero lungo
C_BigInt	Bigint	Testo	255	Testo	255
C_Real	Real	Numerico	Precisione singola	Numerico	Precisione singola
C_Float	Float	Numerico	Precisione doppia	Numerico	Precisione doppia
C_Decimal_10_04	Decimal(10, 4)	Valuta		Valuta	
C_Decimal_18_04	Decimal(18, 4)	Numerico	Decimale	Numerico	Decimale
C_Decimal_19_04	Decimal(19, 4)	Valuta		Valuta	
C_Decimal_29_02	Decimal(29, 2)	Testo	29	Testo	29
C_Char_255	char(255)	Testo	255	Testo	255
C_Char_256	char(256)	Memo		Memo	
C_NChar_255	nchar(255)	Testo	255	Testo	255
C_NChar_256	nchar(256)	Memo		Memo	
C_Varchar_255	Varchar(255)	Testo	255	Testo	255
C_Varchar_256	Varchar(256)	Memo		Memo	
C_Varchar_MAX	Varchar(MAX)	Memo		Memo	
C_text	Text	Memo		Memo	
C_NVarchar_255	nvarchar(255)	Testo	255	Testo	255
C_NVarchar_256	nvarchar(256)	Memo		Memo	
C_NVarchar_MAX	nvarchar(MAX)	Memo		Memo	
C_Ntext	Ntext	Memo		Memo	
C_Datetime	datetime	Data/ora		Data/ora	
C_Smalldatetime	smalldatetime	Data/ora		Data/ora	
C_Timestamp	***timestamp***	**Binario**	**8**	**Binario**	**8**
C_Binary_255	binary(255)	Binario	255	Binario	255
C_Bynary_256	binary(256)	Binario	256	**Oggetto OLE**	
C_Binary_510	binary(510)	Binario	510	**Oggetto OLE**	
C_Binary_511	binary(511)	Oggetto OLE		Oggetto OLE	
C_VarBinary_255	varbinary(255)	Binario	255	Binario	255

C_VarBynary_256	varbinary(256)	Binario	256	**Oggetto OLE**	
C_VarBinary_510	varbinary(510)	Binario	510	**Oggetto OLE**	
C_VarBinary_511	varbinary(511)	Oggetto OLE		Oggetto OLE	
C_VarBinary_MAX	varbinary(MAX)	Oggetto OLE		Oggetto OLE	
C_Image	image	Oggetto OLE		Oggetto OLE	
C_Uniqueidentifier	uniqueidentifier	Numerico	ID replica	Numerico	ID replica

La differenza principale tra Odbc standard e Sql Server è timestamp.
In Sql server il tipo di dato timestamp non è un datetime ma un varbinary di 8 caratteri che si incrementa automaticamente quando si inserisce o si aggiorna una riga che include una colonna di tipo timestamp, questo tipo di campo viene utilizzato per memorizzare la versione della riga. Vedremo più avanti come timestamp può essere utile per velocizzare le operazioni di update e per risolvere alcuni problemi.
Le differenze tra Jet4 e il nuovo motore di Access2007 sono segnate in rosso.
Ho usato il driver Sql Server perché Jet4 non interpreta correttamente i nuovi tipi Nvarchar(Max), Varchar(Max) e VarBinary (Max), infatti li vede come testo di 255 caratteri. Access 2007 li interpreta correttamente con il tipo di dato memo, ma curiosamente in questa versione non memorizza più di 8000 byte nel tipo Varchar(Max) e non più di 4000 nei tipi Nvarchar(Max) e Varbinary(Max).

Indici e primary key

Una tabella collegata via Odbc può essere aggiornabile solo se possiede una chiave primaria o un indice univoco, questa è una necessità di Access+Odbc, infatti Sql server non ha nessun problema a scrivere righe su tabelle senza indici. Se ad esempio vogliamo rendere scrivibile una tabella che ha un campo di tipo autonumber non indicizzato, possiamo creare la definizione di una chiave primaria nella tabella collegata, e perché no, anche in una vista.
Es: **Currentdb.Execute "ALTER TABLE NomeTabella ADD CONSTRAINT PK Primary Key (Campo1)"**
Ovviamente non è una vera chiave primaria, è solo una definizione memorizzata nello schema del nostro file Mdb o Accdb.

Quando si esegue il collegamento di una tabella, Access memorizza nel database locale lo schema dei campi e la definizione degli indici, questa struttura è statica, quindi eventuali modifiche apportate alla tabella posta nel database server, oltre a non essere viste, possono generare problemi. Per questo motivo bisogna riallegare o eseguire il RefreshLink delle tabelle quando necessario.

L'apertura di una tabella comporta nel tempo, la lettura dei valori memorizzati in tutte le righe del campo appartenente alla chiave primaria, in mancanza di una chiave primaria, Access sceglie il primo indice univoco in ordine alfabetico, per questo motivo una chiave primaria o un indice univoco composto da un campo di tipo int (4 Byte) genera un minor flusso di dati dal server al client.

Una differenza sostanziale tra un indice univoco di Sql Server e un indice univoco di Jet, è che l'indice di Sql Server è univoco in ogni caso, anche in presenza del valore Null, al contrario Jet permette di inserire più righe nella tabella con il campo indicizzato valorizzato a Null.

Ci sono diversi sistemi per simulare il comportamento dell'indice univoco di Jet, ad esempio si può creare un indice non univoco e controllare la presenza di duplicati per mezzo di un vincolo check che interroga una funzione, oppure per mezzo di un trigger o di una vista indicizzata.

Gli esempi che seguono sono basati sulla tabella Contact che ho preso dal database AdventureWorks, lo scopo è quello di rendere univoco il campo EmailAddress se valorizzato. La tabella in questione ha già un indice non univoco che si riferisce al campo EmailAddress.

Esempio 1 con vincolo che richiama una funzione.

```
-- Creo la funzione
/*****************************************
 Rende 1 se l'indirizzo mail è unico
 0 se duplicato
 *****************************************/
Create Function dbo.ufn_IsEmailAddressUnica
(@EmailAddress nvarchar(50),@ContactId int)
Returns bit
AS BEGIN
IF @EmailAddress IS NULL
   Return 1

IF EXISTS (Select *
           From dbo.Contact
           Where EmailAddress=@EmailAddress
             AND ContactId<>@ContactId
           )
   Return 0

Return 1
END
GO
--Aggiungo il vincolo alla tabella
ALTER TABLE dbo.Contact ADD CONSTRAINT
CK_EmailAddressUnica CHECK(dbo.ufn_IsEmailAddressUnica(EmailAddress,ContactId)=1)
GO
```

Esempio 2 con Trigger

```
--Creo il trigger
CREATE TRIGGER TR_ContactCkeckDuplicati ON dbo.Contact
AFTER INSERT,UPDATE
AS BEGIN
IF EXISTS(Select 1
          From   dbo.Contact AS C
             Inner Join
                INSERTED I ON C.EmailAddress=I.EmailAddress
          Group By C.EmailAddress
          Having Count(*)>1)
   BEGIN
      Raiserror ('Impossibile inserire un valore duplicato nel campo EmailAddress',16,1)
      Rollback Transaction
   END

END
```

Esempio 3 con vista indicizzata

```
-- Metodo 3 con vista indicizzata
CREATE VIEW dbo.uvs_ContactCkDuplicatiMail
WITH SCHEMABINDING
AS
SELECT ContactId,EmailAddress
FROM   dbo.Contact
WHERE  EmailAddress IS NOT NULL
GO
-- Creo l'indice
CREATE UNIQUE CLUSTERED INDEX PK_uvs_ContactCkDuplicatiMail
ON uvs_ContactCkDuplicatiMail (EmailAddress)
GO
```

Quest' ultimo metodo è da valutare con cura perché le operazioni di manutenzione che Sql Server esegue su una vista indicizzata possono essere maggiori di quelle richieste da un indice di tabella.

Analisi dei comandi inviati da Access a Sql server

Apertura di una tabella

Attiviamo l'opzione TraceSQLMode nel registro di configurazione, apriamo la tabella di esempio Contact e vediamo quali comandi vengono memorizzati nel file SqlOut.Txt.

1) SQLExecDirect: SELECT Config, nValue FROM MSysConf

2) SQLExecDirect: SELECT "dbo"."Contact"."ContactID" FROM "dbo"."Contact"

3) SQLPrepare: SELECT "ContactID","NameStyle","Title","FirstName","LastName","EmailAddress","Phone","rowguid","ModifiedDate" FROM "dbo"."Contact" WHERE "ContactID" = ?

4) SQLExecute: (GOTO BOOKMARK)

5) SQLPrepare: SELECT "ContactID","NameStyle","Title","FirstName","LastName","EmailAddress","Phone","rowguid","ModifiedDate" FROM "dbo"."Contact" WHERE "ContactID" = ? OR "ContactID" = ? OR "ContactID" = ? OR "ContactID" = ? OR "ContactID" = ? OR "ContactID" = ? OR "ContactID" = ? OR "ContactID" = ? OR "ContactID" = ? OR "ContactID" = ?

6) SQLExecute: (MULTI-ROW FETCH)

La prima riga legge le informazioni dalla tabella MSysConf vista in precedenza, il tipo di istruzione è denominato **SQLExecDirect** e come si può intuire indica che la frase Sql viene eseguita direttamente.

La seconda riga, sempre di tipo **SQLExecDirect**, legge la chiave primaria della nostra tabella, i valori letti costituiranno il cursore lato client. Se le righe della tabella non sono molte, (circa 5 - 6 mila per il tipo int) tutti i ContactID vengono trasferiti al client, in caso contrario viene letta solo una parte lasciando in sospeso l'istruzione che verrà ripresa quando ci spostiamo in un record con Id non ancora memorizzato nel client. L'istruzione termina quando ci spostiamo in prossimità delle ultime righe. Come detto in precedenza la lettura del file SqlOut.Txt non permette di vedere quando la lettura degli Id termina, per vederlo bisogna analizzare i comandi con Sql profile.

La terza riga, di tipo **SQLPrepare** prepara una stored procedure che andrà a leggere un'intera riga, la clausola where filtrerà un solo record il cui id corrisponderà al primo letto dalla riga 2

La quarta riga, di tipo **SQLExecute** esegue la stored procedure preparata dalla terza riga leggendo il record. (GOTO BOOKMARK) indica che questo è il segnalibro da cui partiranno le sequenze di lettura in blochi da 10 dei prossimi records.

La quinta riga, di tipo **SQLPrepare** prepara la stored procedure che andrà a leggere 10 records alla volta corrispondenti ai 10 id che verranno passati al filtro nella clausola Where.

La sesta riga che ho riportato una sola volta, di tipo **SQLExecute** esegue la stored procedure preparata dalla quinta riga. (MULTI-ROW FETCH) indica appunto la lettura dei blocchi da 10 records. Nel file SqlOut.Txt troverete diverse righe come questa e servono a leggere i blocchi di 10 records necessari a riempire la tabella aperta in visualizzazione da Acces.

Vediamo come si presentata la tabella appena aperta,

Contact

ContactID	NameStyle	Title	FirstName	LastName	EmailAddre	Phone	rowguid
15851	0		Gabriel	Mitchell	gabriel42@adv	1 (11) 500 555-(	)004C710BD47}
19851	0		Alejandro	Liang	alejandro20@a	1 (11) 500 555-(	)00CE977B267}
1500	0		Christy	Tang	christy23@adv	1 (11) 500 555-(	00D06DD049C}
1960	0		Katherine	Smith	katherine71@a	855-555-0169	)0110419421D}
11665	0	Mr.	Scott	Gode	scott4@adven	164-555-0145	)01388BBC5B2}
133	0		Colleen	Ma	colleen16@ad	1 (11) 500 555-(	)017BF82A615}
9218	0		Karla	Kumar	karla8@adven	1 (11) 500 555-(	)01859BF51B6}
7867	0		Devin	Clark	devin18@adve	1 (11) 500 555-(	)01901B18EB6}
1077	0		Ross	Subram	ross13@adven	1 (11) 500 555-(	)01913355A8D}
15348	0		Ricardo	Kumar	ricardo7@adve	1 (11) 500 555-(	)02476CA9E8A}
8464	0		Melvin	She	melvin0@adve	1 (11) 500 555-(	0252DA5171A}
6576	0		Austin	Butler	austin10@adv	1 (11) 500 555-(	)026E5D12B2B}
1259	0		Barbara	Wang	barbara11@ad	1 (11) 500 555-(	)029805FC83A}
16097	0		Jordan	Nelson	jordan63@adv	614-555-0137	)02A16EE8B7C}

Come si può notare le righe non sono ordinate in base al valore della chiave primaria e sembra non esserci un ordine preciso, in realtà i records sono ordinati in base al valore del campo rowguid. Nella nostra tabella questo campo è associato a una chiave univoca e Sql Server ha scelto la strada più veloce per recuperare i valori di del campo ContactId, ovvero, quella di scorrere le pagine dell'indice associato al campo Rowguid le quali contengono anche i valori della chiave primaria.

Inserimento record

Proviamo ora ad inserire un nuovo record popolando solo i campi FirstName e LastName
Nel file Sqlout.txt vengono aggiunte queste righe

1) SQLExecDirect: INSERT INTO "dbo"."Contact" ("FirstName","LastName") VALUES (?,?)
2) SQLExecDirect: SELECT @@IDENTITY
3) SQLExecute: (GOTO BOOKMARK)
4) SQLExecute: (MULTI-ROW FETCH)
5) SQLExecute: (MULTI-ROW FETCH)

La prima riga inserisce il record popolando solo i campi modificati, la seconda legge il valore del campo ContactId appena inserito, la terza riga legge il record appena inserito, la quarta e la quinta riga leggono gli step di 10 righe partendo dal ContactId letto dalla terza riga che è il nuovo segnalibro da cui partono le successive letture.

Variazione record

Modifichiamo il campo Title del record appena inserito

1) SQLExecute: (GOTO BOOKMARK)
2) SQLExecDirect: UPDATE "dbo"."Contact" SET "Title"=? WHERE "ContactID" = ? AND "NameStyle" = ? AND "Title" = ? AND "FirstName" = ? AND "LastName" = ? AND "EmailAddress" IS NULL AND "Phone" IS NULL AND "rowguid" = ? AND "ModifiedDate" = ?

La prima riga posiziona il segnalibro, nel caso che un altro utente abbia modificato il record prima di questa rilettura, Access ci mostra l'avviso "I dati sono stati modificati … modificare di nuovo il record". La seconda riga esegue l'update pssando nella clausola where il valore del campo ContactId e il valore di tutti i campi prima della modifica. Ma non basta passare l'ID del record da modificare? La risposta è no, perché è necessario controllare che altri utenti non abbiano modificato il record nel tempo tra la rilettura del segnalibro e prima della nostra scrittura, in caso contrario l'istruzione update non potrà trovare il record e Access mostrerà l'avviso di modifica contemporanea di record.
Dopo l'update della riga il record non viene riletto, dando per scontato che i dati appena inseriti nel server siano uguali a quelli memorizzati nel client, questo comporta problemi se la tabella in questione ha un trigger che modifica un campo, ad esempio, un campo contenente la data dell'ultima modifica del record.

Cancellazione Record

Ora cancelliamo il record appena modificato.

1) SQLExecute: (GOTO BOOKMARK)
2) SQLPrepare: DELETE FROM "dbo"."Contact" WHERE "ContactID" = ? AND "NameStyle" = ? AND "Title" = ? AND "FirstName" = ? AND "LastName" = ? AND "EmailAddress" = ? AND "Phone" = ? AND "rowguid" = ? AND "ModifiedDate" = ?
3) SQLExecDirect: DELETE FROM "dbo"."Contact" WHERE "ContactID" = ? AND "NameStyle" = ? AND "Title" = ? AND "FirstName" = ? AND "LastName" = ? AND "EmailAddress" IS NULL AND "Phone" IS NULL AND "rowguid" = ? AND "ModifiedDate" = ?

La prima riga esegue una rilettura del record per controllare che il non sia stato modificato da un altro utente, la seconda riga prepara la stored per la cancellazione, la tesza riga la esegue.

Come per la modifica di un record, anche la cancellazione si assicura di eseguire l'operazione solo se i dati non sono stati modificati da altri operatori, in caso contrario ci verrà segnalato un messaggio di errore.

Il tipo di dato timestamp

Abbiamo visto che per ogni variazione o cancellazione record viene controllato che la vecchia versione dei dati sia ancora presente nel database. In presenza di molti campi questo comporta una clausola Where complicata. Per semplificare il comando SQL di variazione/cancellazione, basta inserire un campo di tipo **timestamp** nella tabella.
In Sql server il tipo di dato **timestamp** è un binario di 8 byte, il suo valore è univoco nell'ambito del database e viene incrementato automaticamente dal server ogni volta che si modifica/inserisce un record in una tabella che possiede tale campo.
Aggiungiamo il campo timestamp alla tabella Contact, rialleghiamola al nostro db e proviamo ad eseguire la modifica di un record.

```
1) SQLExecute: (GOTO BOOKMARK)
2) SQLExecDirect: UPDATE "dbo"."Contact" SET "Title"=?  WHERE "ContactID" = ? AND "timestamp" = ?
3) SQLExecute: (GOTO BOOKMARK)
```

Come nella precedente update la prima riga riposiziona il segnalibro, e la seconda esegue l'update.
Si può notare che la clausola Where è più semplice della precedente perché vengono presi in considerazione solo i campi ContactID e timestamp, che viene usato per conoscere la versione del record. Oltre a questi benefici il campo timestamp evita i problemi derivanti dai campi di tipo float o real che come sappiamo sono approssimati, quindi non è detto che i valori passati dal client nel filtro corrispondano esattamente a quelli valutati dalla Where.
La terza riga che nella precedente update non esisteva, riposiziona il bookmark rileggendo il record, non potrebbe fare altrimenti perché è necessario leggere il nuovo valore del campo timestamp. Grazie a questo meccanismo, un eventuale trigger che modifica uno o più campi della tabella non genera nessun problema.

Il tipo di dato Bit

Il tipo di dato bit in Sql Server può memorizzare i valori 0 , 1 e null, Access lo interpreta con il tipo booleano Sì/No i cui valori possibili sono 0 , -1 e null. Questa differenza può generare problemi nel caso si usi in una query un filtro su tale campo espresso con il valore numerico -1 al posto di true.
Ad esempio, il filtro della seguente query:

```
----
SELECT Contact.EmailAddress, Contact.NameStyle
FROM Contact
WHERE (((Contact.NameStyle)=-1));
----
```

viene passato a Sql Server così come è

```
----
SQLExecDirect: SELECT "dbo"."Contact"."ContactID" FROM "dbo"."Contact" WHERE ("NameStyle" = -1 )
----
```

Con il risultato che nessun record viene restituito al client.

Se al posto di -1 usiamo true

```
----
SELECT Contact.EmailAddress, Contact.NameStyle
FROM Contact
WHERE (((Contact.NameStyle)=True));
----
```

Il filtro viene tradotto correttamente.

```
----
SQLExecDirect: SELECT "dbo"."Contact"."ContactID" FROM "dbo"."Contact" WHERE ("NameStyle" = 1)
----
```

Il problema si pone quando usiamo un filtro derivante da una checkbox posta in una maschera

```
----
SELECT Contact.EmailAddress, Contact.NameStyle
FROM Contact
WHERE (((Contact.NameStyle)=Forms!Maschera1!CasellaControllo0));
----
```

In questo caso viene passato il valore numerico -1 con il risultato che nessun record viene trovato.

Una soluzione è quella di passare il valore assoluto:

```
----
SELECT Contact.EmailAddress, Contact.NameStyle
FROM Contact
WHERE (((Contact.NameStyle)=Abs([Forms]![Maschera1]![CasellaControllo0])));
----
```

Se nel nostro progetto ci sono molte maschere e report con questo tipo di filtro allora si può pensare di sostituire nella tabella il tipo di dato Bit con il tipo Smallint avendo cura di sostituire i valori 1 in -1 e successivamente inserire un vincolo che accetti solo i valori 0 e -1.

Il tipo di dato DateTime e SmallDateTime

Il tipo di dato datetime in Sql server può memorizzare valori che vanno dal 01/01/1753 al 31/12/9999, la parte ora comprende anche i millisecondi con la precisione di 3,33 millisecondi.
Il tipo SmallDatetime può memorizzare valori che vanno dal 01/01/1900 al 06/06/2070, la sua parte ora non consente di memorizzare i secondi.
In una tabella di Access il limite inferiore per le date è il 01/01/0100 quindi l'upsize di una tabella con date inferiori al 01/01/1753 per datetime e inferiore a 01/01/1900 per smalldatetime non potrà essere eseguito.
In Access la data 0 corrisponde al 30/12/1899, in Sql server la data 0 corrisponde al 01/01/1900, quindi è impossibile memorizzare la sola parte ora in un campo smalldatetime.
Se la nostra applicazione interroga dati gestiti da altre applicazioni, ad esempio un progetto ADP di Access, e se si desidera impostare un filtro su un campo datetime in cui è memorizzata solo l'ora, bisogna anteporre la data 01/01/1900 all'ora da filtrare.
Un aspetto curioso derivato dalla differenza tra la data 0 di Sql server e la data 0 di Access possiamo vederlo eseguando questa query.

```
----
SELECT DataOra.Id, DataOra.Data, DataOra.OraIngresso, DataOra.OraUscita
FROM DataOra
WHERE (((DataOra.Data)=#6/7/2007#) AND ((DataOra.OraIngresso)=#12/30/1899 15:30:0#));
----
```

L'istruzione inviata a Sql server è questa

```
----
SQLExecDirect: SELECT "dbo"."DataOra"."Id" FROM "dbo"."DataOra"
WHERE (("Data" = {d '2007-06-07'} ) AND ("OraIngresso" = {t '15:30:00'} ) )
----
```

La parte data del filtro posto sul campo OraIngresso non c'è quindi Sql server cercherà il valore 01/01/1900 15:30:00 e non troverà il record corrispondente.

Passiamo la parte ora tramite un parametro o una funzione vba

```
----
SELECT DataOra.Id, DataOra.Data, DataOra.OraIngresso, DataOra.OraUscita
FROM DataOra
WHERE (((DataOra.Data)=#6/7/2007#) AND ((DataOra.OraIngresso)=CDate('15.30.00')));
----
```

L'istruzione inviata a Sql server è

```
----
SQLExecDirect: SELECT "dbo"."DataOra"."Id" FROM "dbo"."DataOra"
WHERE (("Data" = {d '2007-06-07'} ) AND ("OraIngresso" = ? ) )
----
```

Dal file sqlout.txt non si vede il parametro passato ma possiamo vederlo da profiler

```
----
exec sp_executesql N'SELECT "dbo"."DataOra"."Id"
FROM "dbo"."DataOra"
 WHERE (("Data" = {d "2007-06-07"} ) AND ("OraIngresso" = @P1 ) ) ',N'@P1 datetime','dic 30 1899  3:30:00:000PM'
----
```

Ora il filtro posto sul campo OraIngresso è completo e il record viene trovato.

Questo non è un grosso problema perché in genere filtriamo i records tramite un'apposita maschera e i riferimenti ai controlli di una maschera vengono passati come il parametro del secondo esempio.

Blocco delle pagine dati e deadlock

La connessione a Sql Server viene eseguita impostando il livello di isolamento Read Committed che in breve impedisce ad altre transazioni di modificare le righe di una tabella mentre la nostra transazione sta eseguendo la lettura.
Come detto in precedenza, quando si apre una linked table o una maschera la cui origine record attinge i dati da una tabella collegata, tutti i valori della chiave primaria, o in mancanza di essa, del primo indice univoco scelto in ordine alfabetico, vengono trasferiti al client. Se i records sono molti viene trasferita solo una parte di essi senza terminare la lettura, generando così errori di timeout o innescando una situazione di stallo che costringe sql server a interrompere una delle due transazioni.

Vediamo un esempio, Apriamo la maschera Contact del nostro file accdb, senza toccare nulla, apriamo un'altra sessione di Access, e anche da questa apriamo il nostro accdb e quindi la maschera

Contact, posizioniamoci nel record ***6.553***, modifichiamo un campo e dopo la conferma vedremo che access si blocca. Dopo 60 secondi ci verrà notificato il messaggio di errore
"ODBC: Operazione di update non riuscita su tabella collegata 'Contact' TimeOut Scaduto".

La tabella Contact non è molto grande, occupa 5.456 Kb ed ha 19.972 righe, quindi non è raro che questo tipo di blocco possa presentarsi.

Per evitare il blocco delle pagine dati basta inserire dei filtri appropriati nelle nostre maschere in modo da limitare il numero di bookmark richiesti al server, oppure se la nostra esigenza è quella di poter scorrere molti records, possiamo modificare il livello di isolamento Read Committed impostando la nuova opzione di Sql Server 2005 che utilizzare il controllo delle versioni delle righe.

```
---
USE MASTER
GO
ALTER DATABASE Cisa3
    SET READ_COMMITTED_SNAPSHOT ON;
---
```

Per maggiori informazioni su questa opzione consultare il manual di Sql Server 2005
url:ms-help://MS.SQLCC.v9/MS.SQLSVR.v9.it/udb9/html/753c016d-cfcc-4266-8a60-5b271070794c.htm

6.553
Non è detto che la riga causante il blocco sia sempre la 6.553, il numero dipende dalla pagina dati bloccata che lo contiene.
Per determinare esattamente il numero della riga, si può aprire una traccia con Profiler ed eseguire una query di aggiornamento dalla seconda sessione di Access.

```
---
UPDATE Contact SET NameStyle=0;
---
```

Quando Access si blocca leggere il numero del ContactId dall'ultima riga di Profiler.

I Recordset

Non c'è molta differenza tra l'uso dei dao.recordset su tabelle di Access e su tabelle allegate a Sql Server via Odbc, basta ricordare che una tabella allegata via Odbc può essere aperta solo con un recordset di tipo Dynaset o Snapshot, quindi non è possibile usare l'oggetto Index per cercare un record, inoltre se la nostra tabella ha un campo con la proprietà Identity, (l'autonuber di Access), bisogna inserire nel parametro [Options] la costante dbSeeChanges.

```
----
Set Rs = CurrentDb.OpenRecordset("NomeTabella", dbOpenDynaset, dbSeeChanges)
----
```

L'impossibilità di usare l'oggetto Index non è un problema, al suo posto possiamo usare il metodo FindFirst e se la ricerca viene eseguita su un campo indicizzato, Access ottimizza le istruzioni da inviare a Sql Server e il nostro record viene restituito in modo veloce.
Vediamo un esempio, la tabella Contact ha un indice non univoco sul campo EmailAddress, proviamo a cercare un record passando un valore in tale campo.

```
----
Sub FindFirstConIndice()

'Esegue FindFirst su campo indicizzato

Dim Rs As DAO.Recordset
Set Rs = CurrentDb.OpenRecordset("Select * From Contact Order By EmailAddress", _
        dbOpenDynaset, dbSeeChanges)

Rs.FindFirst "EmailAddress='harold14@adventure-works.com'"

Debug.Print "EmailAddress= " & Rs!EmailAddress & vbCrLf & _
            "ContactId=" & Rs!ContactID

Rs.Close
Set Rs = Nothing

End Sub
----
```

Il record ci viene restituito immediatamente, apriamo il file SqlOut.txt e vediamo le righe create.

```
1) SQLExecDirect: SELECT "dbo"."Contact"."ContactID" FROM "dbo"."Contact" ORDER BY
   "dbo"."Contact"."EmailAddress"

2) SQLPrepare: SELECT
   ContactID","NameStyle","Title","FirstName","LastName","EmailAddress","Phone","rowguid
   ","ModifiedDate" FROM "dbo"."Contact" WHERE "ContactID" = ?

3) SQLExecute: (GOTO BOOKMARK)

4) SQLExecDirect: SELECT "dbo"."Contact"."ContactID" FROM "dbo"."Contact" WHERE
   "EmailAddress" = 'harold14@adventure-works.com'

5) SQLExecute: (GOTO BOOKMARK)
```

La prima riga carica i valori della primary key nel client, la seconda riga prepara la procedura per leggere le righe della tabella, la terza riga legge il record corrispondente al primo ContactId restituito dalla riga 1, la quarta riga viene generata dal metodo FindFirst e legge il ContactId corrispondente all'indirizzo mail cercato, la quinta riga legge il record corrispondente al ContactId rilevato dalla quarta riga.
Possiamo applicare questo metodo anche al recordsetClone di una maschera.

Lo scenario cambia notevolmente se eseguiamo la stessa ricerca su un campo non indicizzato, proviamo a cercare un record utilizzando la ricerca sul campo FirstName

```
---
Sub FindFirstSenzaIndice_Lenta()

'Esegue FindFirst su campo non indicizzato
'Metodo lento che richiede la lettura di tutti i record sino
'a trovare quello interessato

Dim Rs As DAO.Recordset
Set Rs = CurrentDb.OpenRecordset("Select * From Contact " _
                                 & "Order By FirstName,ContactId", _
         dbOpenDynaset, dbSeeChanges)

Rs.FindFirst "FirstName='Harold'"

Debug.Print "FirstName= " & Rs!FirstName & vbCrLf & _
            "ContactId=" & Rs!ContactID

Rs.Close
Set Rs = Nothing

End Sub
---
```

Il record ci viene restituito dopo qualche secondo, se apriamo il file sqlOut.txt vedremo le prime tre righe simili all'esempio precedente, segue poi un enorme numero di SQLExecute: (GOTO BOOKMARK) , circa 8000. In pratica è stata eseguita la lettura di tutte le righe a partire dal primo bookmark, fino a raggiungere quella che soddisfa la ricerca effettuata.

Il metodo appena visto è estremamente inefficiente, gli effetti sono gli stessi di quando premiamo il pulsante Trova da Access, però conoscendo i meccanismi che recuperano le righe, possiamo modificare il nostro codice VBA per spostare la ricerca della riga interessata lato Server.
Con un piccolo trucco possiamo leggere il ContactId corrispondente al FistrName da cercare tramite la funzione Dmin() e poi lo passiamo al metodo FindFirst.

```
---
Sub FindFirstSenzaIndice_Veloce()

'Esegue FindFirst su campo non indicizzato

Dim Rs As DAO.Recordset
Set Rs = CurrentDb.OpenRecordset("Select * From Contact Order By FirstName,ContactId", _
         dbOpenDynaset, dbSeeChanges)

'Cerco l'Id corrispondente al FirstName da cercare
Dim ContactID As Variant
ContactID = DMin("ContactID", "Contact", "FirstName='Harold'")

'Mi posiziono sull'Id trovato
If IsNull(ContactID) = False Then
   Rs.FindFirst "ContactId=" + Str(ContactID)
   Debug.Print "FirstName= " & Rs!FirstName & vbCrLf & _
               "ContactId=" & Rs!ContactID
End If

Rs.Close
Set Rs = Nothing

End Sub
---
```

Le istruzioni inviate a Sql Server sono

```
----
1) SQLExecDirect: SELECT "dbo"."Contact"."ContactID"
FROM "dbo"."Contact" ORDER BY "dbo"."Contact"."FirstName" ,
"dbo"."Contact"."ContactID"

2) SQLPrepare: SELECT
   "ContactID","NameStyle","Title","FirstName","LastName","EmailAddress","Phone","rowgui
   d","ModifiedDate"  FROM "dbo"."Contact"  WHERE "ContactID" = ?

3) SQLExecute: (GOTO BOOKMARK)

4) SQLExecDirect: SELECT MIN("ContactID" )  FROM "dbo"."Contact" WHERE ("FirstName" =
   'Harold' )

5) SQLExecDirect: SELECT "dbo"."Contact"."ContactID" FROM "dbo"."Contact" WHERE
   "ContactID" = 8646

6) SQLExecute: (GOTO BOOKMARK)
----
```

Le prime tre righe sono uguali ai precedenti esempi, poi segue l'istruzione generata dalla funzione Dmin() alla riga 4, quindi la riga 5 che è generata dal metodo FindFirst e infine la lettura del record. Ovviamente anche questo metodo non è molto efficiente perché eseguiamo una ricerca su un campo non indicizzato, Sql Server eseguirà una scansione nelle pagine dell'indice cluster o in mancanza di esso una scansione della tabella, però abbiamo il vantaggio di spostare l'elaborazione lato server.

Come abbiamo detto precedentemente, possiamo aprire recordset di tipo Dynaset o SnapShot.
Se non abbiamo bisogno di un recordset aggiornabile e se le righe da recuperare non sono molte, possiamo utilizzare un recordset di tipo SnapShot, quest'ultimo non utilizza il cursore lato client perché memorizza tutte le righe richieste in locale, può essere utile nelle maschere o sottomaschere che visualizzano dati non aggiornabili

Le Query

Query DML

Le query che eseguono la modifica dei dati su un set di record, (Update, Delete, Insert) sono poco performanti perché Access invia a Sql server tante istruzioni quanti sono i records da aggiornare/cancellare o inserire. Ad esempio l'istruzione **Update Contact Set NameStyle=0** viene eseguita per ogni singolo record della tabella, quindi se la tabella ha 19.972 righe, l'istruzione viene inviata a Sql Server 19.972 volte utilizzando il solito medodo del cursore lato client. Lo stesso effetto si riscontra utilizzando il metodo RunQuery o il metoto Currentdb.Execute.
Per eseguire in modo efficiente questo tipo di query possiamo usare una query pass-through come nell'esempio seguente.

```
---
Function EseguiQueryPt(ByVal strSql As String)

    Dim db As DAO.Database
    Dim Qdf As DAO.QueryDef

    On Local Error GoTo EseguiQuery_Err

    Set db = CurrentDb

    'Creo l'oggetto queryDef
    Set Qdf = db.CreateQueryDef("", strSql)
    'Assegno la stringa di connessione
    Qdf.Connect = db.TableDefs("Contact").Connect
    'Assegno la proprietà ReturnsRecords a False
    Qdf.ReturnsRecords = False
    'Assegno la stringa Sql
    Qdf.SQL = strSql
    'eseguo la query
    Qdf.Execute

EseguiQuery_Fine:
    Set Qdf = Nothing
    Set db = Nothing
    Exit Function

EseguiQuery_Err:
    MsgBox Err.Description, vbCritical
    Resume EseguiQuery_Fine

End Function
---
```

Eseguendo la funzione, **EseguiQueryPt "Update Contact Set NameStyle=0"** possiamo inviare direttamente a Sql Server l'update da eseguire. Bisogna considerare che la sintassi Sql che costituisce la nostra istruzione deve essere scritta nel linguaggio nativo del database Server, nel nostro caso T-SQL.

Un altro metodo per inviare direttamente l'istruzione al server, ed è quello di aprire un database remoto ODBCDirect. Non prenderemo in considerazione questo sistema perché nella nuova versione di Access non è più supportato.

Join esterni.

In genere tutte le query vengono eseguite lato server a patto che non vi siano funzioni specifiche di Access, come le funzioni finanziarie, le funzioni VBA che richiedono in ingresso il valore di un campo della tabella, le funzioni di aggregazione come Dlookup e le subquery nella clausola select.

Una nota dolente sono i Join esterni, se la nostra query ha più di un join esterno Access spezza l'istruzione inviando il primo join esterno e crea una procedura per recuperare le righe del secondo join. Vediamo un esempio:

```
---
SELECT Prodotti.Codice, Prodotti.Descrizione, Prodotti.Um
FROM (Prodotti LEFT JOIN Distinta ON Prodotti.Codice = Distinta.Padre)
     LEFT JOIN Distinta AS Distinta_1 ON Prodotti.Codice = Distinta_1.Figlio
WHERE (((Distinta.Id) Is Null) AND ((Distinta_1.Id) Is Null));
---
```

Questa query ricerca tutte e le righe della tabella prodotti dove il codice prodotto non ha distinta base e che non fa parte di nessuna distinta base. In pratica è la query di dati non corrispondenti che siamo abituati ad usare con i database di Access.
Proviamo ad eseguirla e vediamo cosa viene inviato a Sql Server.

```
1) SQLExecDirect: SELECT "dbo"."Prodotti"."Codice" ,
"dbo"."Prodotti"."Descrizione" ,
"dbo"."Prodotti"."Um" ,"dbo"."Distinta"."Id"  FROM {oj "dbo"."Prodotti" LEFT OUTER JOIN
"dbo"."Distinta" ON ("dbo"."Prodotti"."Codice" = "dbo"."Distinta"."Padre" ) }

2) SQLPrepare: SELECT "Id" ,"Figlio"  FROM "dbo"."Distinta" "Distinta_1"  WHERE
   ("Figlio" = ?)

3) SQLExecute: (PARAMETERIZED QUERY)
4) SQLExecute: (PARAMETERIZED QUERY)
x) SQLExecute: (PARAMETERIZED QUERY)
Ecc. Ecc.
```

La prima riga esegue una sola Left Join, la seconda riga prepara una procedura per recuperare le righe corrispondenti alla seconda Left join, dalla terza riga in poi vengono letti tutti i records passando come filtro il codice prodotto recuperato dalla riga 1, tutto questo genera un grosso volume di istruzioni da Access al Server.
Per questo caso si può trovare una query corrispondente che venga eseguita interamente lato server usando al posto delle due Join esterne l'operatore Not IN nella clausola Where

```
----
SELECT Prodotti.Codice, Prodotti.Descrizione, Prodotti.Um
FROM Prodotti
WHERE (((Prodotti.Codice) Not In (Select padre From Distinta)
  And (Prodotti.Codice) Not In (Select Figlio From Distinta)));
----
```

Sql Server riceve questa unica istruzione

```
SQLExecDirect: SELECT "Codice" ,"Descrizione" ,"Um"  FROM "dbo"."Prodotti"
WHERE ((("dbo"."Prodotti"."Codice" )<> ALL
      (SELECT "MS2"."Padre"  FROM "dbo"."Distinta" "MS2" ))
  AND (("dbo"."Prodotti"."Codice" )<> ALL
      (SELECT "MS1"."Figlio"  FROM "dbo"."Distinta" "MS1" )))
```

l'operatore Not In viene trasformato in <> All , il risultato ovviamente non cambia e la nostra query viene eseguita in un'unica soluzione.
Con i database di Access l'uso dell'operatore Not In nella clausola Where non è performante, al contrario Sql Server ottimizza il piano di esecuzione della query trasformando l'operatore Not in con una Left Anti SemiJoin rendendo la query performante.

Le cose si complicano se abbiamo bisogno di leggere i valori dei campi dalle tabelle esterne.

```
----
SELECT Prodotti.Codice, Distinta.Qt AS QtPadre, Distinta_1.Qt AS QtFiglio
FROM (Prodotti LEFT JOIN Distinta ON Prodotti.Codice = Distinta.Padre)
     LEFT JOIN Distinta AS Distinta_1 ON Prodotti.Codice = Distinta_1.Figlio;
----
```

Per questo caso si può creare una query pass-through di comodo e con una sub tipo questa

```
---
Sub GeneraQryPassTroughGenerica(ByVal StrSql As String)

    Dim db As Database
    Dim Qdf As QueryDef

    On Local Error GoTo GeneraQryPassTroughGenerica_Err

    Set db = CurrentDb
    Set Qdf = db.QueryDefs("qry_PassThroughGenerica")
    Qdf.SQL = StrSql
    Qdf.Connect = db.TableDefs("Contact").Connect
    db.QueryDefs.Refresh

GeneraQryPassTroughGenerica_Fine:
    Set Qdf = Nothing
    Set db = Nothing
    Exit Sub

GeneraQryPassTroughGenerica_Err:
    MsgBox Err.Description, vbCritical
    Resume GeneraQryPassTroughGenerica_Fine

End Sub
---
```

che sostituisce la proprietà Sql, infine inseriamo il nome della query nell'origine record della maschea o del report interessato.

La Clausola TOP

La clausola Top di Access è diversa da quella di Sql Server, Access include gli eventuali duplicati in presenza di un ordinamento, Sql server no.
Una query con la clausola Top richiede tutte le righe al server per poi eseguire la Top in locale.

Suggerimenti generali

Cercare di ridurre la quantità di dati richiesta al server inserendo i filtri nelle maschere o nei report, i filtri sono uno strumento molto potente perché inviano al Server la clausola Where corrispondente.

Il pulsante Filtro della barra strumenti è gestito bene, rigenera il cursore locale inviando al server la clausola Where corrispondente.

Il pulsante Ordinamento della barra strumenti è gestito bene ed invia al server la Clausola Order By corrispondente.

Utilizzare il comando trova solo su recordset di piccole dimensioni, come abbiamo visto invia un grosso volume di istruzioni al server.

Richiedere nelle query solo i campi necessari, la Select * è comoda ma se non c'è bisogno di tutti i campi è anche meno performante.

Evitare caselle combinate o caselle di riepilogo con molti records.

Controllare con Sql Profiler o attivando TraceSQLMode che le nostre query vengano eseguite lato server.