# Una classe Php per l'accesso a Firebird

## Parte 2: Implementazione metodi specifici per la gestione dei campi Blob

Con una serie di 3 articoli vediamo come sviluppare una soluzione RDBMS accedendo al database Firebird con PHP

**di Raoul Scarazzini**

Nel primo articolo (pubblicato nel mese di ottobre, ndr) abbiamo iniziato a costruire una classe PHP per l'accesso ad un database Firebird. Sulla base di proprietà e metodi specifici abbiamo inziato ad effettuare query ed a navigare nei resultset di queste.

Quello che ci apprestiamo a fare ora è entrare nello specifico, costruendo metodi che ci permettano di gestire un tipo di campo particolare denominato BLOB.

Quanto segue si basa totalmente sulla classe presentata e costruita nello scorso articolo, senza quella base i nuovi metodi implementati non hanno senso.

### Panoramica sui campi BLOB

In genere quando si definisce un campo in una tabella bisogna sapere a priori quale dovrà essere la sua dimensione massima. Molte volte però non è possibile stabilire a priori quanti dati dovrà contenere un campo, basti pensare alle note di un'ordine: come si può stabilire quanto un cliente scriverà all'interno di queste? La risposta si trova appunto nei campi BLOB.

Un campo BLOB a differenza di tutte le altre tipologie definibili in un RDBMS ha la caratteristica di essere dimensionato dinamicamente, non viene definita cioè la grandezza del campo in fase di creazione.

In genere campi di questo tipo vengono utilizzati per gestire grandi quantità di dati di grandezza varibile: immagini bitmap, file audio, file video, capitoli di libri e tantissime altre tipologie.

Proprio a causa di questa particolare attitudine, i campi BLOB non si possono gestire in maniera "comune" tramite operazioni di SELECT o INSERT, ma necessitano di misure particolari sia per la scrittura che per la lettura. Questo perchè non viene registrato direttamente il contenuto del campo all'interno della tabella, ma solo il suo identificativo, il BLOB_ID. I dati del BLOB vengono registrati in un'altra area del database, suddivisi in segmenti. Il BLOB_ID rappresenta l'indirizzo di partenza di questi segmenti.

Questo concetto è spiegato egregiamente in Figura 1.

### Definire un campo BLOB

In Firebird la definizione di un campo BLOB all'interno di una tabella avviene in questo modo:

```
NOMECAMPO BLOB SUB_TYPE <tipo subtype> SEGMENT SIZE
<grandezza>
```

Questo tipo di sintassi ci permette di dichiarare il nome del campo, il fatto che sia di tipo BLOB, il sottotipo (SUB_TYPE) e la grandezza dei sagmenti ad esso associati (SEGMENT SIZE).

SUB_TYPE rappresenta il sotto-tipo del campo, un valore che descrive la natura del dato in esso contenuto. In firebird ne esistono 9 tipi (da 0 a 8):

- 0
- 1 o TEXT
- 2 o BLR
- 3 o ACL
- 4 o RANGES
- 5 o SUMMARY
- 6 o FORMAT
- 7 o TRANSACTION_DESCRIPTION
- 8 o EXTERNAL_FILE_DESCRIPTOR:

Una spiegazione completa di ciascuno di questi particolari sottotipi la si può trovare a questo URL:

```
http://www.ibphoenix.com/a502.htm
```

per i nostri esempi noi adotteremo il subtipo TEXT, usato appunto per registrare e manipolare testo. Se non si è certi di quale tipo scegliere, si può non indicare nulla in quanto di default verrà considerato il sottotipo 0 che generalmente viene usato per dati binari o indefiniti.

SEGMENT SIZE rappresenta la grandezza massima che i segmenti del campo BLOB possono assumere. Il valore indicato può arrivare fino a 32767 bytes. Se non si indica il valore di grandezza del segmento, Firebird considera default

SEGMENT SIZE 80.

Anche in questo caso sarebbe bene approfondire il concetto di SEGMENT SIZE sulla documentazione ufficiale di Interbase 6.0 nel pdf denominato DataDef.pdf, pagine relative ai campi BLOB (Le pagine dalla 76 in poi). Questa documentazione pur riferendosi ad una versione di Interbase passata (la prima ed unica OpenSource, 6.0) si adatta per i concetti esposti anche a Firebird. La sipuò trovare sul sito ufficiale di Firebird a questo indirizzo:

```
http://firebird.sourceforge.net/index.php?op=doc&id=userdoc
```

scaricare e consultare questa documentazione non è affatto un lavoro inutile, in particolare, relativamente ai campi BLOB, il capitolo 8 del libro "EMBEDDED SQL GUIDE".

Seppur datata risulta in diverse occasioni assai preziosa.

### Prepararsi al lavoro

Nel precedente articolo avevamo già definito nel database di test una tabella con un campo BLOB:

```
CREATE TABLE PEOPLE (
    pcode    INTEGER NOT NULL,
    psurname VARCHAR(30) NOT NULL,
    pname    VARCHAR(30) NOT NULL,
    pquote   BLOB SUB_TYPE TEXT SEGMENT SIZE 240,
    PRIMARY KEY (pcode)
);
```

La riga che a noi interessa è chiaramente quella contenente la dichiarazione del campo BLOB:

```
    pquote   BLOB SUB_TYPE TEXT SEGMENT SIZE 240,
```

Questa segnala a Firebird di creare un campo BLOB di sottotipo TEXT (quello che ci interessa memorizzare sono frasi "memorabili", quindi essenzialmente testo) con segmenti di 240 bytes.

Questo campo ci permetterà di sperimentare le funzioni PHP per la gestione dei BLOB in Firebird.

### Utilizzare i campi BLOB in PHP

Ciò che ci interessa implementare nella nostra classe PHP relativamente ai campi BLOB, sono le due operazioni fondamentali: lettura e scrittura.

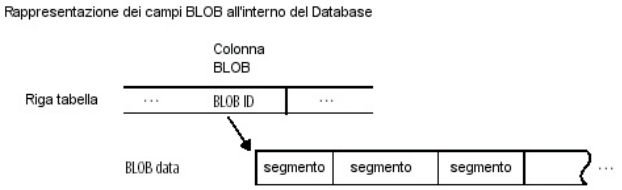


Figura 1

Come già accennato precedentemente, per reperire un campo BLOB da una tabella non è sufficiente effettuare una SELECT su questo e scriverne il contenuto in output semplicemente con una echo, come se si trattasse di un campo qualsiasi (ad esempio VARCHAR).

Se si dovesse procedere in questa maniera, non si riceverebbe alcun errore dal compilatore PHP, ma a video non apparirebbe il testo contenuto nel BLOB, bensì il suo ID, il quale apparendo in forma binaria risulterebbe del tutto incomprensibile.

Quindi procedendo correttamente i passi da svolgere per reperire il contenuto di un campo BLOB diventano i seguenti:

- Ricavare tramite SELECT il BLOB_ID del campo interessato;
- Aprire l'area di memoria del database identificata dal BLOB_ID;
- Reperire il contenuto del BLOB da quest'area per elaborarlo secondo le nostre esigenze;
- Chiudere l'area di memoria in questione.

Lo stesso concetto va applicato (anche se in maniera opposta) all'inserimento, i cui passi possono essere rappresentati come segue:

- Creare un'area di memoria nel database;
- Inserire in questa i valori relativi al contenuto (Testo o dati binari relativi ad immagini, video etc.);
- Registrare l'ID di quest'area di memoria nel campo BLOB della tabella;
- Chiudere l'area di memoria.

Per effettuare queste due serie di operazioni creeremo due

*Senza* ***blob_field_out***

| Nome | Frase memorabile |
|---|---|
| Vito | x] 00 19 00 08 ^ 00 19 00 08 € |
| Michael | x] 00 19 00 08 ^ 00 19 00 08 € 00 02 |
| Sonny | x] 00 19 00 08 ^ 00 19 00 08 € 00 04 |
| Fredo | x] 00 19 00 08 ^ 00 19 00 08 € 00 06 |
| Tom | x] 00 19 00 08 ^ 00 19 00 08 € 00 08 |
| Peter | x] 00 19 00 08 ^ 00 19 00 08 € |
| Kay | x] 00 19 00 08 ^ 00 19 00 08 € 00 0F |

Figura 2

metodi dal nome autoesplicativo: blob_field_in e blob_field_out.

PHP offre le funzioni di gestione dei campi BLOB in Interbase/Firebird, ma purtroppo sul sito ufficiale queste non sono documentate, poco male se si pensa che fino a poco tempo fa non erano nemmeno menzionate!

Un elenco completo delle funzioni disponibili lo si trova qui:

http://it2.php.net/manual/it/ref.ibase.php

### Reperire valori da un campo BLOB

Dettati i presupposti possiamo inziare a creare un metodo che visualizzi il contenuto di un campo BLOB ricavato dal resultset di una query.

I parametri di cui il nostro metodo avrà bisogno saranno sicuramente il nome del campo interessato ed il formato di uscita dei dati, in modo da poter effettuare una pre-elaborazione dell'output e velocizzare i tempi di lavorazione.

Partendo dal presupposto che tramite il metodo exec_query (illustrato nel precedente articolo) abbiamo nell'array Record della nostra classe una riga del resultset della query effettuata, possiamo iniziare a concentrarci sui metodi PHP:

Per prima cosa è necessario reperire il BLOB_ID del nostro campo, la prima funzione che viene in nostro aiuto è ibase_blob_open: a questa va passato semplicemente il valore binario ricavato dalla query per farsi restituire il BLOB_ID relativo. Ora che conosciamo qual è l'indirizzo di memoria dei dati che ci interessano, possiamo inziare a reperirli tramite la funzione ibase_blob_get. A questa è necessario passare chiaramente il BLOB_ID ed in più la lunghezza in byte di quanti dati si vogliono reperire. La funzione restituirà tanti byte quanti ne sono stati richiesti fino a che non arriverà alla fine del campo, momento in cui il valore restituito diventerà FALSE.

Essendo il valore BLOB diviso in segmenti e non sapendo a priori la grandezza di questi, per effettuare un reperimento preciso useremo un comodo espediente: generiamo un numero casuale tramite la funzione rand e utilizziamo come lunghezza di byte il resto della divisione tra questo numero e 1024. Questo numero non potrà mai superare 1024, di conseguenza ci consentirà di reperire aree dati minori di 1024 bytes (1 Kb).

Con *blob_field_out*

| Nome | Frase memorabile |
|---|---|
| Vito | I'll make him an offer he can't refuse. |
| Michael | Don't ask me about my business, Kay. |
| Sonny | Goddamn FBI don't respect nothin'. |
| Fredo | I'm smart and I want respect ! |
| Tom | Mr. Corleone never asks a second favor once he's refused the first, understood ? |
| Peter | Mikey, why don't you tell that nice girl you love her ? |
| Kay | It made me think of what you once told me:<br>"In five years the Corleone family will be completely legitimate."<br>That was seven years ago. |

Figura 3

Seppur questo step possa sembrare superfluo (si potrebbe passare un valore fisso inferiore a 1024 sempre e comunque) è necessario e fondamentale per non sovraccaricare la memoria durante le operazioni di reperimento di grosse moli di dati da campi BLOB...provare per credere!

Accodando i vari "pezzi" del nostro BLOB in una variabile potremmo finalmente avere in locale i dati che ci interessano ed operare su di essi per emetterli a seconda del formato d'uscita richiesto.

Una volta che quanto ci interessa si trova in una variabile locale, possiamo chiudere il BLOB attraverso la funzione ibase_blob_close alla quale va passato solo il BLOB_ID interessato.

I formati d'uscita potrebbero essere 3: restituire il dato così come lo si riceve, restituire il dato con i caratteri speciali html convertiti in "escape sequences" per prevenire l'esecuzione di codice html o anche javascript maligno ed infine restituire il dato formattato per l'html (quindi con gli "a capo" trasformati in tag <br>), ma sempre con il discorso sicurezza applicato ai caratteri speciali html.

In conclusione il nostro metodo potrebbe presentarsi così:

```
function blob_field_out($field_name, $html)
{
 $blob_text = "";
 if ($this->Record[$field_name])
 {
  $blob_id = ibase_blob_open($this-
>Record[$field_name]);
  while ($piece = ibase_blob_get ($blob_id, rand()
➥ % 1024))
   $blob_text .= $piece;
  ibase_blob_close($blob_id);
  switch ($html)
  {
   case "s": break;
   case 0 || NULL || FALSE: $blob_text =
➥ htmlspecialchars($blob_text); break;
   default: $blob_text =
➥ nl2br(htmlspecialchars($blob_text)); break;
  }
 }
 return $blob_text;
}
```

Tutta la funzione dipende comunque dal fatto che $this ->Record[$field_name] sia valorizzato, in caso contrario il valore restituito sarà una stringa vuota.

Il ciclo while serve per accodare i valori reperiti dal campo BLOB all'interno della variabile $blob_text e fino a che verrà assegnato un valore a $piece, $blob_text continuerà ad incrementarsi.

Il costrutto switch ci consente invece di selezionare il

tipo di formato output: passando quindi “s” come secondo parametro al nostro metodo, considereremo il contenuto del BLOB affidabile e così come ci viene restituito, se invece questo non sarà valorizzato (0, NULL o FALSE) restituiremo il tutto “corretto” da eventuali entità html (la funzione htmlspecialchars non fa altro che questo), mentre se il parametro $html un valore qualsiasi diverso da questi oltre ad effettuare un controllo sulle entità html convertiremo anche i caratteri “\n” o newline in accapo html ossia “<br>”.

### Inserire valori in un campo BLOB

Inserire dei valori BLOB all’interno di una tabella significa creare un’area di memoria all’interno del database, scrivere in questa i dati che ci interessano, registrare il BLOB_ID associato nella tabella interessata e chiudere l’area di memoria aperta.

Il nostro metodo dovrà richiedere in input due parametri: il primo è il database handle, che rappresenterà l’indirizzo relativo alla connessione sulla quale stiamo lavorando, il secondo conterrà i dati da inserire nel campo BLOB.

Tramite il metodo PHP ibase_blob_create al quale va passato l’handle del nostro database reperiremo il BLOB_ID relativo ad una nuova area di memoria creata. Questo prezioso valore ci consentirà di inserire dati all’interno di quest’area tramite la funzione ibase_blob_add alla quale passeremo il BLOB_ID e la variabile contenente i dati da registrare. Infine faremo restituire al nostro metodo quanto riceveremo dalla funzione ibase_blob_close, ossia il BLOB_ID da registrare all’interno del campo della tabella.

```
function blob_field_in($dbh == NULL, $blob_text)
{
 if (!is_null($dbh) && is_resource($dbh))
  $this->Link_ID = $dbh;
 else
  $this->connect();

 $blob_id = ibase_blob_create($this->Link_ID);
 ibase_blob_add($blob_id, $blob_text);
 return ibase_blob_close($blob_id);
}
```

Se $dbh è valorizzato (non deve essere nullo e deve essere una risorsa), allora verrà usato questo handle per svolgere le operazioni, in caso contrario verrà richiamata la funzione connect.

### Dalla teoria alla pratica

Una volta aggiunti questi due nuovi metodi alla nostra classe, possiamo procedere con la creazione di un esempio effettivo di inserimento di un record contenente un BLOB e la visualizzazione di questo.

Il codice contenuto nel Riquadro 1 riassume tutti i concetti esposti sinora.

Si parte con l’inclusione del file classe, la creazione di questa mediante il suo costruttore, e si inserisce tramite i parametri il nuovo record all’interno della tabella.

Particolare attenzione va posta alla riga

```
$db->add_param($db->blob_field_in(0, $blob));
```

che aggiunge alla lista dei parametri anche il BLOB_ID ricavato dalla funzione blob_field_in che ci siamo costruiti, a questa vengono passati due parametri: 0 che rappresenta una connessione vuota ed obbliga quindi la funzione a stabilirne una nuova e $blob ossia la variabile che contiene il testo (con accapo) da inserire nel nostro BLOB.

Una volta eseguito con successo l’inserimento vengono proposti i due tipi di visualizzazione del risultato: nel primo il campo viene presentato così come è estratto dalla tabella, e come si può vedere dalla Figura 2, ciò che appare è assolutamente incomprensibile, nel secondo grazie alla riga

```
    echo “<td>” . $db->blob_field_out(“PQUOTE”, 1)
. “</td>”;
```

Viene visualizzato il risultato corretto, ossia quanto restituito dalla funzione blob_field_out, alla quale abbiamo passato il nome del campo interessato “PQUOTE” e 1 per indicare di convertire gli “a capo” in “<br>”.

### Conclusioni

Ancora una volta ci tengo ad indicare come l’implementazione di tutti i metodi presentati è puramente indicativa. Si può fare tutto e più di tutto con questi strumenti che l’OpenSource ci mette a disposizione, questa è solo una delle tante vie.

In questo secondo articolo abbiamo cercato di capire il concetto di BLOB e di implementare metodi che ne consentano l’utilizzo, nel prossimo parleremo di transazioni, cercando anche in questo caso di creare metodi specifici che ci consentano di lavorare al meglio spremendo Firebird fino all’ultimo bit.

**Raoul Scarazzini** - http://web.tiscali.it/rascasoft
<rascasoft@tiscali.it>

## Approfondimento

Tutti i sorgenti citati nel testo sono riportati nella versione elettronica ottenibile all’indirizzo **www.dossier.duke.it** indicando il Codice Documento **L0411RS2**

## Riquadro 1: Inserimento di un campo BLOB, visualizzazione scorretta e corretta

```
<html>
<body>
<?php
 // Inclusione file ib_class
 include("ib_class.inc");

 // Creazione classe db
 $db = new ib_class("localhost", "/home/janet/dbtest.gdb", "",
0, 3, "SYSDBA", "masterkey");

 // Inserimento BLOB

 // Stringa sql
 $ssql = "INSERT INTO PEOPLE (PCODE, PNAME, PSURNAME, PQUOTE)
VALUES (?, ?, ?, ?)";

 // Definizione variabile BLOB (con caratteri accapo)
 $blob = "It made me think of what you once told me:\n\"In
five years the Corleone family will be completely legitima-
te.\"\n
That was seven years ago.";

 // Passaggio parametri
 $db->add_param(6);
 $db->add_param("Kay");
 $db->add_param("Adams Corleone");
 $db->add_param($db->blob_field_in(0, $blob));

 // Esecuzione query
 if (!$db->exec_query(0, $ssql))
  // C'è un errore, lo visualizzo
  echo "Si è verificato un errore: " . $db->Error;
 else
 {
  echo "Inserimento blob completato con successo !<p>";

  // Stringa sql
  $ssql = "SELECT a.PNAME, a.PQUOTE FROM PEOPLE a";

  // Visualizzazione senza utilizzo di blob_field_out
  echo "<p><i>Senza <b>blob_field_out</b></i><br>";

  // Esecuzione query
  if (!$db->exec_query(0, $ssql))
   // C'è un errore, lo visualizzo
   echo "Si è verificato un errore: " . $db->Error;
  else
  {
   echo "<table border=\"1\">\n";
   echo "<th>Nome</th><th>Frase memorabile</th>\n";

   // Creo le righe della tabella
   while ($db->next_record())
   {
    echo "<tr>";
    echo "<td>" . $db->Record["PNAME"] . "</td>";
    echo "<td>" . $db->Record["PQUOTE"] . "</td>";
    echo "</tr>\n";
   }
   echo "</table>";
  }

  // Visualizzazione con blob_field_out
  echo "<p>Con <i><b>blob_field_out</b></i><br>";

  // Esecuzione query
  if (!$db->exec_query(0, $ssql))
   // C'è un errore, lo visualizzo
   echo "Si è verificato un errore: " . $db->Error;
  else
  {
   echo "<table border=\"1\">\n";
   echo "<th>Nome</th><th>Frase memorabile</th>\n";

   // Creo le righe della tabella
   while ($db->next_record())
   {
    echo "<tr>";
    echo "<td>" . $db->Record["PNAME"] . "</td>";
    echo "<td>" . $db->blob_field_out("PQUOTE", 1) . "</td>";
    echo "</tr>\n";
   }
   echo "</table>";
  }
 }
?>
</body>
</html>
```