Gestione immagini con PHP: Filesystem o Database?

a cura di Ludovico Caldara
ludovico@ludovicocaldara.net

Indice

1.Introduzione

Ecco un argomento che è stato spesso oggetto di dibattito su Usenet:
Meglio inserire le immagini direttamente nel database oppure inserire solo il path e lasciare il file su disco?
Non c'è una risposta definitiva a questa domanda, e probabilmente non ci sarà mai; mi piacerebbe però fornire un elenco di punti da tenere in considerazione quando ci si trova di fronte alla fatidica scelta. Entrambe le soluzioni forniscono vantaggi indubbi ma anche problemi di gestione. Per venire incontro alle mie capacità mentali, come linguaggio di riferimento userò PHP anche se il discorso si può estendere ad altri linguaggi, mentre cercherò di tenere slegata (o variegata) dai prodotti la parte relativa ai database, fatta eccezione per i test di velocità per cui ho preso in considerazione mysql e postgresql.

1.1. Le immagini

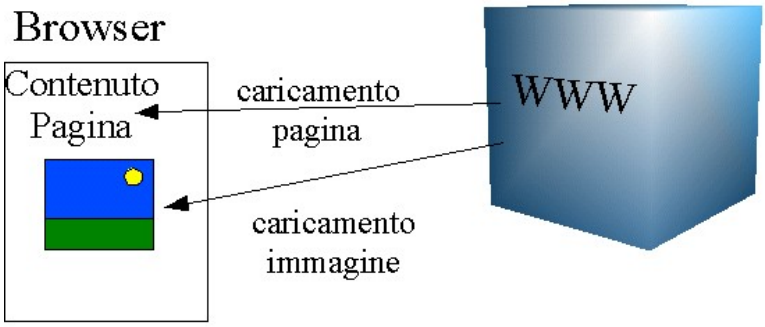
Due note veloci che serviranno (forse) per capire meglio il problema: le immagini su web si dividono in due grandi categorie: le immagini che fanno parte della **pagina** (bottoni, bordi, freccie, loghi ecc.) e quelle che fanno parte del **contenuto** (soprattutto foto, thumbnails e grafici). Quelle che rientrano nel nostro problema sono le seconde. Il problema di decidere dove inserirle si pone:

- quando la pagina è dinamica (stiamo parlando anche di PHP, no?)
- quando si tratta di elaborare un *archivio* di foto
- quando le foto in questione sono ragionevolmente “abbastanza” da dover pensare ad una gestione su database dei dati inerenti (descrizione, data, dimensione) con tutte le agevolazioni del caso: ricerca, inserimento, aggiornamento ecc.

Supponiamo che ogni chiamata via web ad un'immagine sia fatta col tag <img>:

```
<!-- html stuff-->
<img src=”/path_to_image/image”>
<!-- other html stuff-->
```

Salta subito all'occhio un particolare: la chiamata alla pagina è distinta dalla chiamata all'immagine:
Quindi una pagina html che richiama una foto col seguente codice:




```
<html><body> <img src='foto.jpg'>  <p>Beccati 'sta foto!</p>
</body></html>
```

generarà due chiamate HTTP come evidenzia il log del webserver:

```
[...] "GET /pagina.html HTTP/1.1" 200 92
[...] "GET /foto.jpg HTTP/1.1" 200 259558
```

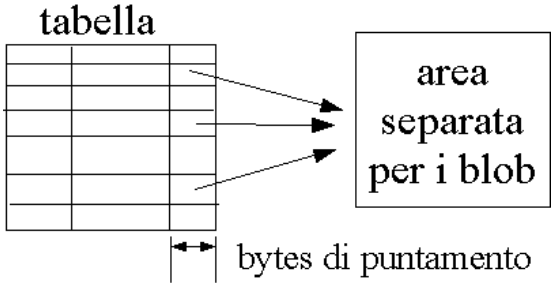
2.Immagini nel Database: Vantaggi

In questo capitolo vediamo quali sono i vantaggi che ci fanno propendere per una soluzione su DB. Lascieremo quindi spazio agli svantaggi nel capitolo 3 e poi ai test.

2.1. Sfatiamo i luoghi comuni

2.1.1. “Le immagini appesantiscono il database”

E' abbastanza diffuso il pensiero che aumentando la quantità di bytes del database salvandoci dentro le immagini questo diventi più lento nelle ricerche e scansioni in genere. Non è vero. In tutti i database che supportano campi blob, l'informazione non è **mai** immagazzinata all'interno della struttura della tabella, ma sempre al di fuori: i campi a lunghezza limitata vengono allocati nello stesso spazio di allocazione dell record, mentre al posto del campo blob viene inserito un puntatore ad una zona di allocazione separata.



A seconda del DBMS la lunghezza di questo puntatore è variabile; ad esempio, per i puntatori di mysql vengono allocati da 5 a 9 bytes aggiuntivi (4 bytes di puntatore più i bytes necessari per salvarne la lunghezza), per postgresql ci sono 4 bytes aggiuntivi che contengono l'oid interno di postgres che rappresenta l'oggetto salvato), per interbase/firebird ci sono 8 bytes aggiuntivi. Ognuno ovviamente ha meccanismi di allocazione e lettura del dato diversi a seconda dell'architettura del DBMS.
I bytes aggiuntivi per il puntamento sono da considerarsi pochi se consideriamo che sono a lunghezza fissa (quindi non inducono alla frammentazione dei blocchi di salvataggio dei dati come avviene per i campi a lunghezza variabile) e sono inferiori rispetto a quanto richiederebbe il salvataggio del path se le immagini fossero salvate su filesystem. Pertanto, per quanto riguarda la **scansione** della tabella, inserire un campo blob piuttosto che un campo varchar(40) (ad esempio) è più performante.

2.1.2. “Se inserisci le immagini nel database non puoi sfruttare la cache del browser”

Non è vero. A prima impressione può sembrare, visto che le chiamate ad uno script PHP per estrapolare un'immagine in genere sono nella forma:
`http://sito/script.php?id=12345`
questo viene interpretato dal browser come segno di dinamismo e quindi non cacha l'immagine. E' però possibile richiamare lo script anche così:
`http://sito/script.php/12345.jpg`
in questo modo si può sfruttare la variabile predefinita 'PATH_INFO' presente nel superglobal array $_SERVER[].
L'esempio seguente illustra come sia possibile cachare un'immagine caricata dal database

fino a mezzanotte:

```
<?php

//generazione data per scadenza

$domani = mktime (0,0,0,date("m")  ,date("d")+1,date("Y"));
$domani=date('D, d M Y H:i:s',$tomorrow)

header("Expires: $domani");
header("Cache-Control: max-age=86400, min-fresh=86400, max-
stale=0");

list($id,$foo)=split("\.",substr($_SERVER['PATH_INFO'],1));

header ("Content-type: image/jpeg");

$sql="select foto from foto where id = ".$id;
//... estrazione blob dal database
?>
```

Giocando con la funzione header() si possono ottenere tutti i controlli voluti sul caching dell'immagine.

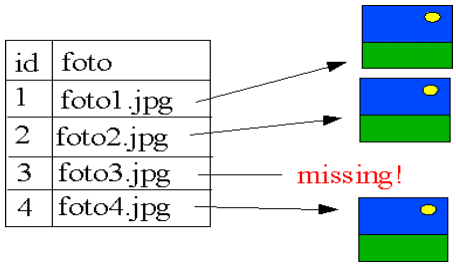
2.2. Dati e immagini insieme: omogeneità e robustezza

2.2.1. L'importanza della consistenza: le transazioni

Uno dei target principali nell'implementazione di una base di dati è riuscire a mantenere la *consistenza* dei dati a fronte di qualsiasi problema: che sia il crash di sistema o del database, o il crash del webserver o una “richiesta di interruzione” da parte dell'utente, i dati devono essere sempre *coerenti* con la logica dell'applicazione. Cosa vuol dire? Rifacciamoci al nostro caso; supponiamo di inserire le immagini nel filesystem e di tenere sul database solamente il nome (per comodità niente path...)

```
<?php
header("Content-type: image/jpeg");
mysql_connect("localhost","","");
mysql_select_db("test");

//come template di creazione estraggo i dati da un file fisso e li
//riverso dentro un file con nome generato a caso.
$origfile="./foto.jpg";
$fd = fopen ($origfile, "rb");
$contents = fread ($fd, filesize ($origfile));
fclose ($fd);

$filename=substr(md5(uniqid("")),16).".jpg";

$sql="insert into fotofs (foto) values ('".$filename."')";
mysql_query($sql) or die(mysql_error());

$fd=fopen("./".$filename,"wb");
fwrite($fd,$contents);
fclose($fd);
print $contents;
?>
```

Ok, ho inserito il nome nel db e ho creato la foto su filesystem. Cosa succede però se per qualsiasi motivo (e ci sono tanti *point of failure*) l'immagine non può essere creata sul filesystem? Ho una situazione sporca: un record su database che punta ad un file che non esiste.



Il codice di inserimento, steso con poca attenzione volutamente, può essere irrobustito parecchio:

```
<?php
mysql_connect("localhost","","");
mysql_select_db("test");

$origfile="./foto.jpg";
$fd = fopen ($origfile, "rb");
$contents = fread ($fd, filesize ($origfile));
fclose ($fd);

//per evitare due files uguali, tribuliamo un po':
$i=10;
while($esiste=file_exists($filename=substr(md5(uniqid("")),16).".jpg"
)&&$i--);
if($esiste) die("Failure: unable to decide filename");


//proviamo a inserire il file
$fd=fopen("./".$filename,"wb") or die("Failure: writing ".$filename);

fwrite($fd,$contents) or die("Failure: writing ".$filename);
fclose($fd); //questo fallirà mica?

//nessun errore in scrittura ma l'ha inserito e lo vedo?
if(!file_exists("./".$filename)) {
    die("Failure: file doesn't exists");
}

//file inserito. inseriamo il record!
$sql="insert into fotofs (foto) values ('".$filename."')";
if(!mysql_query($sql)) {
    //non inserisce. ricancelliamo il file!
    unlink("./".$filename) or die("Failure: there is the file but not
the record, say admin to check!");;
    die("Failure: ".mysql_error());
} else {
    header("Content-type: image/jpeg");
    print $contents;
}
?>
```

Come si può osservare, il codice viene complicato notevolmente per avere **qualche**

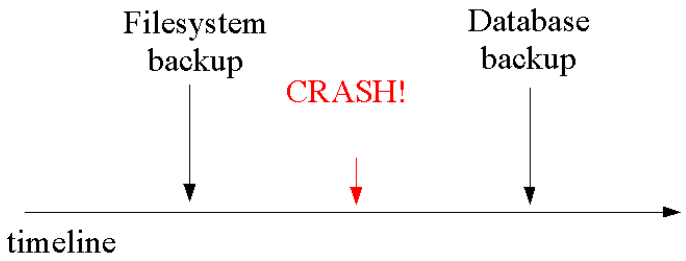
sicurezza in più in fase di inserimento. Gli stessi accorgimenti andrebbero utilizzati in tutte le operazioni di questo genere.
Un punto di failure è però in agguato: cosa succede se il sistema crasha (“panico del nocciolo”, come dice la traduzione di google!) mentre è in corso l'operazione di inserimento nel database? Come possiamo avere la certezza che il record sia stato inserito senza creare incongruenze? Ricorrere alla funzione **register_shutdown_function** potrebbe non bastare. Il sistema a *transazioni* dei database ci viene incontro, dove supportato, garantendoci sempre e comunque un esito della transazione come compiuto (record + immagine) o incompiuto (né record né immagine). Il recente supporto alle transazioni di mysql ha bussato alle porte di tutti i suoi utilizzatori, andandosi a mettere in pari su questo discorso con databases come Oracle, PostgreSQL, Interbase/Firebird ecc. anche se in particolare per questo esempio di mysql non ce n'è bisogno:

```
<?php
mysql_connect("localhost","","");
mysql_select_db("test");
$filename="./foto.jpg";
$fd = fopen ($filename, "rb");
$contents = fread ($fd, filesize ($filename));
fclose ($fd);
$sql="insert into fotodb (foto) values
('".addslashes($contents)."')";
mysql_query($sql) or die("Failure: ".mysql_error());
header("Content-type: image/jpeg");
print $contents;
?>
```

L'operazione di inserimento su mysql viene garantita come atomica, e visto che i blob in mysql possono essere inseriti con una semplice insert, non c'è bisogno di ricorrere alle transazioni.
Con altri database (oltre al diverso trattamento dei blob che vedremo), sarebbero stati necessari gli steps aggiuntivi di inizio e fine transazione: *begin* e *commit*.

2.2.2. Backup, restore, recover *“point in time”*

Come diceva una *MCT* che ho conosciuto, i dati sono la cosa più importante. Troppo spesso capita di commettere errori (diciamocelo: più spesso dei failures di sistema) per cui i dati vengono compromessi o cancellati; la pianificazione di un buon **piano di backup** deve sempre essere al centro dell'interesse quando si progetta un nuovo ambiente. C'è chi considera la parte del backup un punto a sfavore per la gestione delle immagini su db, secondo me si tratta invece di avere la possibilità di scegliere sempre come e quando backuppare i propri dati. Soprattutto, se il database interessato prevede operazioni di recover *“point in time”*, avere tutti i dati centralizzati in un punto facilita notevolmente le cose.
Infatti, backuppando separatamente i dati su nastro e quelli su filesystem, in caso di crash o perdita di dati dovremmo recoverare i dati forzatamente all'ora dell'ultimo backup su filesystem se vogliamo avere la consistenza dei dati, mentre tenendo tutto su database, anche scegliendo tra diverse strategie (backup full ecc.) il meccanismo di log dei database permette in moltissimi casi di recuperare lo stato del database a un attimo prima della perdita dei dati. Inoltre, il backup si può centralizzare in un punto solo, diminuendo per



certi versi i punti di failure.

2.3. Quante immagini dobbiamo gestire?

Di questo bisogna certamente farsene un'idea. Il sito conterrà le foto dell'ultimo Interrail, l'album personale o le fototessere dei dipendenti di una grossa azienda? O non sarà mica uno share via web di appassionati di fotografia? A seconda dei casi, abbiamo diversi ordini di grandezza: poche decine, poche centinaia, poche migliaia, **milioni** di record. Se i dati sono tanti, database possono contenere con certezza svariati milioni di record (ad eccezione di qualche database meno robusto, non si fanno nomi) e gestiscono l'allocazione interna in modo indipendente dall'effettiva allocazione su disco. Non si può fare lo stesso discorso per i files.
L'allocazione dei files (su UNIX) avviene tramite la scrittura di un "*inode*" contenente i dati di sistema del file (owner, gruppo, permessi ecc.) e la sua collocazione. A seconda del filesystem la struttura degli inode cambia, insieme al numero di step necessari per risalire ai dati del file e anche la dimensione dell'inode può variare (a volte può essere decisa al momento della creazione del filesystem, ma in genere è > 100bytes). In sostanza, un filesystem è migliore di un altro per contenere tanti files:
ad esempio i filesystems *VFS*, *EXT2* e *EXT3* applicano una struttura lineare degli inodes, quindi l'accesso ad un file richiede la scansione sequenziale di tutta la struttura, mentre filesystems come *ReiserFS*, *XFS* e (dalle ultime versioni) *FFS*, applicano una struttura ad albero degli inodes, riducendo notevolmente i tempi di accesso.
Secondo "voci" sentite sulla rete, in accesso si nota un punto di collasso delle prestazioni intorno ai 20.000 files per i filesystems a struttura lineare, mentre si va oltre il mezzo milione di files per i filesystems a struttura ad albero; in ogni caso, oltre una certa soglia (ben più bassa) si hanno difficolta nella gestione dei files da linea di comando. Questi dati sono chiaramente indicativi e slegati dalla reale architettura hardware a disposizione. C'è un modo per ovviare questo problema, anche se può essere un po' macchinoso: si può implementare un meccanismo che bilancia la creazione dei files in una struttura a sottodirectories; per esempio, si possono creare 36 directories (una per ogni lettera dell'alfabeto e per ogni cifra) e inserire in una o nell'altra il file a seconda del primo carattere che compone il nome del file (analogamente, se i files sono tanti, si può procedere allo stesso modo per il secondo, il terzo e così via). Il numero di modi per bilanciare il numero dei files in varie sottodirectories dipende solo dalla fantasia del programmatore.
Essendo gli inodes maggiori come dimensione rispetto ai puntatori allocati all'interno del database, viene da se pensare che la ricerca su database sia comunque più performante sull'elevata quantità di dati. Se poi lo script php anziché stampare direttamente il nome del file dentro il tag image, richiama un altro script per la gestione del file (come spesso è meglio fare), si effettuerebbe comunque una ricerca sul database e una aggiuntiva sul filesystem, aumentando i tempi di ricerca.

C'è anche un altro aspetto legato agli inodes da considerare: *il numero di inodes è limitato* in fase di creazione del filesystem. Il numero disponibile è comunque monitorabile tramite il comando unix *df -i*, ma bisogna tenere conto del numero in fase iniziale e calcolarne la dimensione per avere prestazioni ottimali, cosa che non è necessaria per la gestione su database.

2.4. Altri vantaggi

- Se i files vengono richiamati direttamente nel tag <img>, bisogna inserire i files nella document_root: la sicurezza va gestita con più attenzione.
- Se l'architettura è distribuita (webserver, dbserver e fileserver separati), può essere comodo avere un punto unico di acquisizione (a scapito però delle performances).
- I files potrebbero dare problemi con il nome: ad esempio sotto unix i nomi files sono case-sensitive mentre non lo sono sotto windows. Anche un'estensione .jpg piuttosto che .JPG può generare problemi. Inoltre i path cambiano (/ sotto unix, \ sotto windows ecc.) quindi bisogna pensare alla conversione dei path memorizzati nel db.

3.Immagini nel database: Svantaggi

Passiamo agli svantaggi che ha la gestione delle immagini nel db che per certi versi sono più scoraggianti...

3.1. BLOB... Cos'è, si mangia???

3.1.1. I campi blob non sono previsti su tutti i databases...

I campi blob sono quasi universalmente supportati: il loro utilizzo è previsto in databases molto diffusi come *oracle*, *mysql*, *postgresql*, *interbase*, *dbase*, *paradox*, *picoSQL* ecc.. Non c'è però uno standard de facto sul trattamento dei dati binari. Non solo, questo tipo di dato non è previsto in tutti i DBMS. Per esempio, *msql* o *berkeley db* non supportano ancora i blob: se si sceglie di inserire le immagini sul database, bisogna anche tenere conto di quale database scegliere.

3.1.2. Blob sì, ma a modo mio!

Sebbene i DBMS che gestiscono i campi blob siano molto numerosi, non esiste un modello di gestione omogeneo, ad esempio:

- mySQL supporta il tipo di campo blob alla pari degli altri a patto che venga fatto un “escape” in C-style sullo stream in ingresso
- Interbase supporta i dati blob ma necessita di operazioni particolari (allocazione del blob, retrieve dell'oid, inserimento dell'oid)
- postgreSQL supporta i tipi *oid* (che funzionano grossomodo come i blob di interbase) e *bytea* (simile al blob di mysql ma con differenze nella modalità di escape)
- Oracle supporta i blobs in modo molto più esteso, si tratta comunque generalmente di precreazione del blob e successivo popolamento
- MSSQL supporta il campo *image* che deve essere popolato tramite stream in base64 (quindi un notevole spreco di banda).

Per ognuno di questi databases, la gestione dei blob va studiata, capita e implementata in maniera differente. Questo complica notevolmente le cose, e se si vogliono scrivere funzioni o classi di astrazione per accedere a db differenti dall'applicazione c'è un bel po' di lavoro da fare (vedere l'appendice per gli esempi), inoltre, se si utilizzano classi di astrazione già fatte, si avranno mediamente due tipi di classi: quelle che implementano la gestione dei blob ma pochi database e quelli che possono essere utilizzati su molti database ma senza la gestione dei blobs (ad esempio basta vedere le classi PEAR::DB e PEAR::MDB).

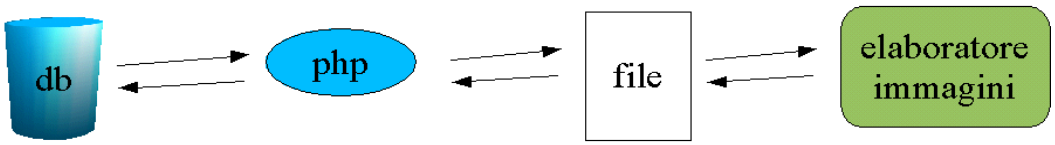
3.1.3. Architetture eterogenee

Se il progetto che abbiamo messo in piedi, prevede nostro malgrado lo scambio di dati tra architetture eterogenee (si prenda per esempio, lo scambio di documenti tra due aziende che usano una mysql e l'altra oracle), non è possibile effettuare lo scambio in modo indolore utilizzando la sintassi SQL standard (per via delle differenti implementazioni dei blob) ma è necessario scaricare i blob su filesystem per ricaricarli dall'altra parte: il “file grezzo” infatti è l'unico “protocollo” che ci permette con scioltezza di migrare dei dati binari da un sistema ad un altro differente. Implementare su filesystem la gestione delle immagini ci toglie quindi diversi grattacapi!

3.2. Filesystem mon amour!

3.2.1. Quando manipolare le immagini mi costa caro...

Inserire le immagini sul database rende l'accesso alle immagini più difficile: un pregio se si tratta di sicurezza, un grave difetto quando si ha necessità di manipoalarle. Supponiamo ad esempio di dover elaborare con ImageMagick tutte le foto del nostro archivio in modo da migliorare il contrasto:
convert image.jpg -contrast image.jpg
questa operazione è facilmente eseguibile in modalità batch se le immagini si trovano già su filesystem, mentre se vogliamo eseguire le stesse operazioni sulle immagini nel database dobbiamo ricorrere forzatamente a questo flusso di operazioni:



quindi si ha un lavoro notevole per estrarre una ad una tutte le immagini per poi salvarle su file ed infine elaborarle, ovviamente una sola per volta e facendo attenzione ai problemi di memoria (ovviamente non possiamo caricare tutte le immagini in php se sono troppe!).

3.2.2. Il nome del file è già un'informazione

Rifacendoci all'introduzione, il tag immagine genera una chiamata HTTP separata. Se ci va bene che l'utente richieda direttamente l'immagine, senza effettuare altri controlli, possiamo immettere nel tag <img> direttamente il nome del file estratto dal database:

```
<?php
mysql_connect("localhost","","");
mysql_select_db("test");
$sql="select foto from fotofs where id=".$_GET['id'];
$result=mysql_query($sql);
$riga=mysql_fetch_row($result);
$filename=basename($riga[0]);
print "<img src='images/".$filename."'><br>".$filename;
?>
```

che in certi casi ci fa risparmiare un accesso al database. Non in tutti però: visto che per la stampa del tag è possibile immettere uno script PHP con l'id del record senza accedere al database, a volte la connessione in fase di stampa del tag è evitabile. Quando però è possibile stampare direttamente il tag, l'accesso al file è notoriamente più veloce passando per il webserver (tenendo conto delle limitazioni discusse nel paragrafo 2.3).

4. Qualche test di velocità

Attenzione: i test effettuati sono molto ridotti come numero, come quantità di dati e come casistiche. In particolare vorrei sottolineare che il filesystem considerato è quasi sempre di tipo ext3, non possiede quindi una gestione ad albero degli inodes, i databases su cui ho fatto prove sono esclusivamente mysql e postgresql, quindi i risultati sono da ritenersi puramente indicativi e legati all'architettura presente sulla macchina di prova (che è molto ridotta nelle prestazioni).

4.1. Test 1

Creazione di 20000 immagini png molto piccole (~160 bytes): il nome è casuale e lungo 16 caratteri + ".png" + nome della directory.
Le immagini vengono caricate in campi blob per la prova su db oppure il nome viene caricato in campi varchar per la prova su fs. Il test prevede l'estrazione di **1000** immagini (su un parco di 20000) con id generati casualmente: le immagini vengono caricate da database oppure vengono caricate da fs dopo aver fetchato il nome dal database; non vengono stampate a video ma semplicemente caricate in una variabile PHP e viene calcolato il tempo.

Benchmark 1:
- DB mysql 3.23.47
- FS fat32
- Windows2000
- PIII 500 – 192Mb

risultati:
BLOB: total: 11.846894264221 average: 0.011846894264221
FS: total: 66.91615152359 average: 0.06691615152359

Benchmark 2:
- DB mysql 4.0.11a-gamma
- FS ext3
- Linux 2.4.21-0.13mdk
- P200 – 92Mb

risultati:
BLOB: total: 8.73918318748 average: 0.00873918318748
FS: total: 23.9139792919 average: 0.0239139792919

Benchmark 3:
- DB PostgreSQL 7.3.2
- FS ext3
- Linux 2.4.21-0.13mdk
- P200 - 92Mb

risultati:
BLOB (bytea): total: 6.38801133633 average: 0.00638801133633
BLOB (oid): total: 21.825137496 average: 0.021825137496
FS: total: 23.0698097944 average: 0.0230698097944

4.2. Test 2

Il test 2 è del tutto simile al test 1 (stesse immagini, stesso numero) ma cambia leggermente il modello di estrazione. Per l'estrazione delle immagini dal database vengono selezionati e fetchati **tutti e 20000 gli id** e disposti in un array. Poi vengono estratti a caso **1000** numeri all'interno di un ciclo e viene ripetuto il collegamento al database per l'estrazione dell'immagine in base all'id contenuto nell'array. Per l'estrazione da filesystem invece vengono selezionati e fetchati **tutti e 20000 i nomi dei files** e nel ciclo di generazione casuale il nome del file viene estratto direttamente dall'array e viene aperto direttamente il file senza più passare 1000 volte per il database (da notare che a ogni ciclo, per simulare il comportamento reale, la connessione veniva aperta e chiusa).

Benchmark 1:
- DB mysql 4.0.11a-gamma
- FS ext3
- Linux 2.4.21-0.13mdk
- P200 – 92Mb

risultati:
BLOB: total: 10.1007918119 average: 0.0101007918119
FS: total: 22.7593261003 average: 0.0227593261003

Benchmark 2:
- DB PostgreSQL 7.3.2
- FS ext3
- Linux 2.4.21-0.13mdk
- P200 – 92Mb

risultati:
BLOB (bytea): total: 7.28376805782 average: 0.00728376805782
FS: total: 22.6445019245 average: 0.0226445019245
BLOB (oid): total: 25.841062665 average: 0.025841062665

4.3. Test 3

Questo test è uguale al test 1 ma cambiano i numeri:
Anziché 20000 immagini piccole, proviamo a operare su **400 immagini più grandi**: il nome del file varia da 9 a 20 caratteri, sono immagini .jpg che occupano da 40k a 500k (**280k** di media), così questa volta la dimensione delle immagini si avvicina di più ad un album fotografico di medie dimensioni. L'estrazione è avvenuta per **100** immagini casuali anziché 1000.

Benchmark 1:
- DB mysql 4.0.11a-gamma
- FS ext3
- Linux 2.4.21-0.13mdk
- P200 – 92Mb

risultati:
BLOB: total: 3.95623004436 average: 0.0395623004436
FS: total: 7.60730409622 average: 0.0760730409622

Benchmark 2:
- DB PostgreSQL 7.3.2
- FS ext3
- Linux 2.4.21-0.13mdk
- P200 – 92Mb

risultati:
BLOB (oid): total: 3.90436601639 average: 0.0390436601639
FS: total: 6.99397492409 average: 0.0699397492409
BLOB (bytea)*: total: 79.4350948334 average: 0.794350948334 decoding: 34.9940547943

4.4. Test 4

Questo test è uguale al test 2 per modalità di accesso alle immagini ma le dimensioni ed il numero sono uguali al test 3.

Benchmark 1:
- DB mysql 4.0.11a-gamma
- FS ext3
- Linux 2.4.21-0.13mdk
- P200 – 92Mb

risultati:
BLOB: total: 4.8049749136 average: 0.048049749136
FS: total: 6.97686755657 average: 0.0697686755657

Benchmark 2:
- DB PostgreSQL 7.3.2
- FS ext3
- Linux 2.4.21-0.13mdk
- P200 – 92Mb

risultati:
BLOB (oid): total:4.31843268871 average: 0.0431843268871
FS: total: 11.6424782276 average: 0.116424782276
BLOB (bytea)*: total: 86.9752392769 average: 0.869752392769 decoding: 38.6086765528

*Il tempo “decoding” è il tempo necessario per effettuare la conversione da stream in character set locale a stream binario. Per chiarimenti sull'elevato tempo di elaborazione di bytea su file di dimensioni considerevoli, leggere il thread “**BLOB performance test FYI** “ che parte dal post <**HEECIHEEJDBMCCGMGIOBCEBOCHAA.jshevland@j-elite.com**> presente in usenet.

A. Appendice

A.1. Un po' di codice

Generazione di 20000 immagini di piccola dimensione (le immagini grandi sono prese dal mio album di foto digitale quindi niente script!)

```
<?php
$max=20000;

$h=100; $w=100;
$r=0; $g=0; $b=0;
for($i=0 ; $i<$max ; $i++){
    if($r!=255) {
        $r+=5;
    } elseif ($g!=255) {
        $r=0;
        $g+=5;
    } elseif ($b!=255) {
        $r=0;
        $g=0;
        $b+=5;
    } else {
        $r=0;
        $g=0;
        $b=0;
    }
    $im = @imagecreate ($w, $h)
            or die ("Cannot Initialize new GD image stream");
    $background = imagecolorallocate ($im, $r, $g, $b);
    imagefill ($im, $w/2, $h/2, $background);
    $text_color = imagecolorallocate ($im, 255, 255, 255);
    imagestring ($im, 1, 5, 5,  $r.".".$g.".".$b, $text_color);
    $f=10; //10 tentativi per nomefile random
    while($esiste=file_exists($filename="images/".substr(md5(uniqid("
")),16).".png")&&$f--);
    if($esiste)die("Failure: unable to decide filename");
    print "File ".$filename.": ".$r.".".$g.".".$b."<br>\n";
    imagepng ($im,$filename);
}
?>
```

Inserimento nel db mysql dei nomi dei files associati ad un id autoincrementante:

```
<?php
$conn = mysql_connect ("localhost","","");
mysql_select_db ("test");
function tree($dir_corrente, $dir_iniziale=FALSE) {
        chdir($dir_corrente);  // Entro nella directory
        if (!$dir_iniziale) {
                $dir_iniziale = $dir_corrente;
        }
   $d = opendir (".");  // Apro directory iniziale
   while ($file = readdir($d)) {
      if ($file != ".." && $file != ".") {
         if (is_dir($file)) {
             $tree[] = "DIR :: $file";
             $tree[] = tree(getcwd().'/'.$file, getcwd());
         } else {
            $tree[] = "FILE :: $dir_corrente/$file";
                $file=realpath("$dir_corrente/$file");
```

```
                if (eregi("png", $file)) {
                        print $file."\n";
                       inserisci($file);
                }
            }
          }
    }
    closedir($d);
    chdir($dir_iniziale);
     return $tree;
}
$list = tree("/system_path_to/images/");
function inserisci($filename) {
    $rs = mysql_query("insert into fotofs (foto)
values('".addslashes($filename)."')", $GLOBALS['conn']);
}
?>
```

Inserimento nel db mysql delle immagini nei campi blob (sempre con id autoincrementante):

```
<?php
$conn = mysql_connect ("localhost","","");
mysql_select_db ("test");

function tree($dir_corrente, $dir_iniziale=FALSE) {
        chdir($dir_corrente);  // Entro nella directory
        if (!$dir_iniziale) {
                $dir_iniziale = $dir_corrente;
        }
    $d = opendir (".");  // Apro directory iniziale

    while ($file = readdir($d)) {
       if ($file != ".." && $file != ".") {
          if (is_dir($file)) {
              $tree[] = "DIR :: $file";
              $tree[] = tree(getcwd().'/'.$file, getcwd());
          } else {
             $tree[] = "FILE :: $dir_corrente/$file";
                $file=realpath("$dir_corrente/$file");
                if (eregi("png", $file)) {
                        print $file."\n";
                   inserisci($file);
                }
              }
          }
     }
    closedir($d);
    chdir($dir_iniziale);
     return $tree;
}
$list = tree("/system_path_to/images/");
function inserisci($filename) {
    $fd = fopen ($filename, "rb");
    $contents = fread ($fd, filesize ($filename));
    fclose ($fd);
    $rs = mysql_query("insert into fotodb (foto)
values('".addslashes($contents)."')", $GLOBALS['conn']);
}
?>
```

Script generico di inserimento immagine in MySQL:

```
<?php
mysql_connect("localhost","","");
mysql_select_db("test");
$filename="./foto.jpg";
$fd = fopen ($filename, "rb");
$contents = fread ($fd, filesize ($filename));
fclose ($fd);
$sql="insert into fotodb (foto) values
('".addslashes($contents)."')";
mysql_query($sql) or die("Failure: ".mysql_error());
?>
```

Script generico di estrazione e visualizzazione immagine da mysql:

```
<?php
header ("Content-type: image/jpeg");
mysql_connect("localhost","","");
mysql_select_db("test");
$sql="select foto from fotodb where id=".$_GET['id'];
$result=mysql_query($sql);
$riga=mysql_fetch_row($result);
print $riga[0];
?>
```

Script generico di inserimento di un Large Object (oid) in postgreSQL:

```
<?php
//lettura file
$filename="./foto.jpg";
$fd = fopen ($filename, "rb");
$contents = fread ($fd, filesize ($filename));
fclose ($fd);

$conn = pg_connect ("host=localhost port=5432 dbname=foo user=bar password=baz");
pg_query ($conn, "begin");
$oid = pg_lo_create ($conn);
$rs = pg_exec($conn,"insert into foto(foto) values($oid);");//object field type must be "oid"
$handle = pg_lo_open ($conn, $oid, "w");
pg_lo_write ($handle, $contents);
pg_lo_close ($handle);
pg_query ($conn, "commit"); //OR END
pg_close();
?>
```

Script generico di estrazione e visualizzazione di un Large Object (oid) in postgreSQL:

```
<?php
header ("Content-type: image/jpeg");
$conn = pg_connect ("host=localhost port=5432 dbname=foo user=bar
password=baz");
$sql="select foto from foto where id = ".$id;
$rs = pg_exec($conn,$sql) or die($sql);
$row = pg_fetch_row($rs,0) or die($sql);
pg_query ($conn, "begin");
$loid = pg_lo_open($conn,$row[0],"r") or die($row[0]);
pg_lo_read_all ($loid);
pg_lo_close ($loid);
```

```
pg_query ($conn, "commit"); //OR END
pg_close();
?>
```

Script generico di inserimento di un'immagine in un campo bytea di postgresSQL:

```
<?php
//lettura file
$filename="./foto.jpg";
$fd = fopen ($filename, "rb");
$contents = fread ($fd, filesize ($filename));
fclose ($fd);
$conn = pg_connect ("host=localhost port=5432 dbname=foo user=bar
password=baz");
pg_query ($conn, "begin");
$rs = pg_exec($conn,"insert into fotobytea (foto)
values('".pg_escape_bytea($contents)."'::bytea);"); //tipo bytes
pg_query ($conn, "commit"); //OR END
pg_close();
?>
```

Script generico di estrazione e visualizzazione di un'immagine in un campo bytea di postgresSQL:

```
<?php
$conn = pg_connect ("host=localhost port=5432 dbname=foo user=bar
password=baz") or die ("connect");
$id=$_GET['id'];
$sql="select foto from fotobytea where id = ".$id;
$rs = pg_exec($conn,$sql) or die($sql);
$row = pg_fetch_row($rs,0) or die($sql);
pg_close();
header ("Content-type: image/jpeg");
echo pg_unescape_bytea($row[0]);
?>
```

A.1.1. Scripts per Test 1 e 3

Benchmark su db:

```
<?php
function getmicrotime(){
    list($usec, $sec) = explode(" ",microtime());
    return ((float)$usec + (float)$sec);
    }
srand((double)microtime()*1000000);
$totaltime=(float)0;
$max=1000; //100 per il test 3
for($i=0;$i<$max;$i++) {
    $id = rand(1,20000);//400 per il test 3
    $time_start = getmicrotime();
    //connessione al db
    //selezione record
    //fetch blob dentro una variabile
    //disconnessione
    $totaltime+= getmicrotime() - (float)$time_start;
}
print "total: $totaltime  average: ".$totaltime/$max;
?>
```

Benchmark su fs:

```php
<?php
function getmicrotime(){
    list($usec, $sec) = explode(" ",microtime());
    return ((float)$usec + (float)$sec);
    }
srand((double)microtime()*1000000);
$totaltime=(float)0;
$max=1000; //100 per il test 3
for($i=0;$i<$max;$i++) {
    $id = rand(1,20000);//400 per il test 3
    $time_start = getmicrotime();
    //connessione al db
    //selezione record
    //fetch nome file
    //apertura, lettura e chiusura file
    //disconnessione
    $totaltime+= getmicrotime() - (float)$time_start;
}
print "total: $totaltime  average: ".$totaltime/$max;
?>
```

A.1.2. Scripts per Test 2 e 4

Benchmark su db:

```php
<?php
function getmicrotime(){
    list($usec, $sec) = explode(" ",microtime());
    return ((float)$usec + (float)$sec);
    }
srand((double)microtime()*1000000);
$totaltime=(float)0;
$max=1000; //100 per test 4
//connessione al db
//selezione di TUTTI gli id
//fetch degli id dentro ad un array $records
for($i=0;$i<$max;$i++) {
    $id = rand(1,20000); //400 per test 4
    $time_start = getmicrotime();
    //connessione al db
    //selezione record corrispondente all'id casuale
    //fetch del blob dentro ad una variabile
    //disconnessione
    $totaltime+= getmicrotime() - (float)$time_start;
}
print "total: $totaltime  average: ".$totaltime/$max;
?>
```

Benchmark su fs:

```php
<?php
function getmicrotime(){
    list($usec, $sec) = explode(" ",microtime());
    return ((float)$usec + (float)$sec);
    }
srand((double)microtime()*1000000);
$totaltime=(float)0;
$max=1000; //100 per il test 4
//connessione al db
//selezione di TUTTI gli id e nomi files
//fetch dei record dentro ad un array $records
for($i=0;$i<$max;$i++) {
    $id = rand(1,20000);//400 per il test 4
    $time_start = getmicrotime();
    //niente più connessione al db
    $fp=fopen($records[$id][1],"r");
    $content=fread($fp, filesize($records[$id][1]));
    fclose($fp);
    $totaltime+= getmicrotime() - (float)$time_start;
}
print "total: $totaltime  average: ".$totaltime/$max;
?>
```

A.2. Qualche riferimento

- Introduzione tecnica ai blob di firebird/interbase
  http://www.ibphoenix.com/main.nfs?a=ibphoenix&page=ibp_blobs
- Post di L. Serni sui controlli di integrità per immagini su filesystem
  <vqp6auorfhoi904dddmruarj49713ojqbr@L.Serni>
- Precedente test (poco raccomandabile) effettuato dal sottoscritto:
  <Xns917C973992D17punkarruso@10.16.96.46>
- Post iniziale del thread:"read/write slows down because of too many files" per la gestione di molti file su filesystem:
  <e500ed91.0208121313.1d8e4780@posting.google.com>
- Post iniziale del thread: ""too many files" in an FFS dir? " per la gestione di molti files su filesystem:
  <20001215205713.A19404@jake.capybara.org>
- "Images in the Database": pro e contro per il dbms Sybase:
  <V$pczbNJBHA.202@forums.sybase.com>
- "too many files in one directory" considerazioni sui fs di linux
  <slrna9ksaa.5ok.danceswithcrows@samantha.crow202.dyndns.org>
- Quando la shell non riesce a gestire troppi files...
  http://www.gnu.org/software/fileutils/doc/faq/#Argument%20list%20too%20long
- Documenti sulla gestione dei files e inodes di NFS:
  http://www.netapp.com/tech_library/3006.html
  http://www.netapp.com/tech_library/3002.html
- Benchmarks e considerazioni sulla gestione dei 2 tipi di blob su postgres:
  <HEECIHEEJDBMCCGMGIOBCEBOCHAA.jshevland@j-elite.com>
- Links su PHP:
  http://www.php.net/manual/it/function.ibase-blob-create.php (interbase)
  http://www.php.net/manual/it/function.pg-lo-open.php (postgres oid)
  http://www.php.net/manual/it/function.pg-escape-bytea.php (postgres e bytea)
  http://www.php.net/manual/it/function.pg-unescape-bytea.php (nuova)