Laboratorio di Basi di Dati e Programmazione Web

DBMS embedded – SQLite

Stefano Zacchiroli
zack@pps.jussieu.fr

DBMS più “facili”

- installare / configurare / istanziare un DBMS non è un task triviale ...
  - richiede competenze che esulano dalle normali competenze di utenti e sviluppatori
- ... ma SQL ed il modello relazionale sono diffusi
  - know-how degli sviluppatori in materia
  - la disponibilità di un motore di query SQL base è requisito di molti applicativi
    - e.g.: molte applicazioni LAMP
    - e.g.: … o più in generale web-based
- i DBMS embedded affrontano i problemi di deployment dei DBMS

DBMS embedded

- a differenza dei DBMS stand-alone, i DBMS embedded:
  - non vengono eseguiti in un processo separato
    - (in senso architetturale, `fork()` non è rilevante)
    - nessuna necessità di implementare nuovi canali di accesso
  - sono implementati come librerie linkate (dinamicamente o staticamente) nell'applicazione che li usa
    - nessuna necessità di configurare e istanziare un servizio di sistema
      - usuali competenze degli sviluppatori: uso di librerie
      - usuali compentenze degli utenti: configurazione
  - usano file ordinari come data storage
    - nessuna necessità di configurare realm aggiuntivi di permessi
    - ci si basa sui permessi del filesystem

SQLite

- SQLite è un DBMS embedded public-domain
  - il più diffuso TTBOMK
- scritto in C (65kloc)
  - codice elegante e ben commentato
  - API semplice
- utilizzabile come libreria (shared / static)
  - low memory footprint: 150-250 Kb (minimale-completo)
- ottime prestazioni
  - nei casi d'uso appropriati …
  - ... nei quali spesso è più veloce dei fratelli maggiori stand-alone !!

SQLite – feature

- self-contained
- zero-configuration
- DB su file singolo
- transazioni ACID
- scala a TB di dati
  - supporta campi di tipo “BLOB”
- è disponibile un top-level interattivo
- implementa SQL-92, eccezioni:
  - no foreign keys
  - no 100% ALTER TABLE
  - no 100% trigger
  - no RIGHT/OUTER join
  - no nested transaction
  - no VIEW scrivibili
  - no GRANT/REVOKE
    - … ma non servono!

SQLite – usi appropriati

- non è un DBMS “one size does fit all”
  - estremo sx nel trade-off: semplicità ↔ feature implementate
    - velocità e basso footprint sono conseguenze

- usi appropriati:
  - Web apps
  - Rimpiazzo per application-specific file format
  - sistemi embedded
  - db temporanei, analisi fire-and-forget da cmdline
  - sviluppo RAD
  - sperimentazione di estensioni
  - didattica

- usi *non* appropriati
  - grandi dataset (petabyte)
  - architetture inerentemente distribuite
  - alta concorrenza (locking)
  - query SQL molto complesse
    - i.e. molti join

SQLite – pacchetti

- Pacchetto sorgente `sqlite3`
- Pacchetti binari
  - `sqlite3`
    - top-level interattivo per query (“sqlite3”)
  - `libsqlite3-0` / `libsqlite3-dev`
    - librerie shared (dipendenze per i runtime) / librerie di sviluppo (dipendenze per la compilazione)
  - `sqlite3-doc`
    - documentazione (mirror del website)
    - ... ma (ovviamente) manca l'utilissimo wiki: http://www.sqlite.org/cvstrac/wiki
- reference: http://www.sqlite.org

SQLite – installation HOWTO

- facile (per il toplevel):

```
aptitude install sqlite3
```

- le librerie shared vengono installate automaticamente come dipendenze di pacchetti
- non c'è necessità di setup post-installazione
  - ogni applicazione crea il DB di cui necessità invocando la libreria shared
- le librerie -dev vanno installate on-demand:

```
aptitude install libsqlite3-dev
```

SQLite – gestione dei DB

- uso dei database

```
$ sqlite3 filename
```

- creazione di database
  - vengono creati in maniera lazy alla prima scrittura su di un database non esistente

```
$ sqlite3 foo.sqlite
sqlite> CREATE TABLE foo (id INT);
sqlite> CTRL-D
```

- rimozione di database

```
rm -f filename   # :-)
```

SQLite – shell

- `sqlite3` è il top-level interattivo per l'accesso a DB sqlite
  - GNU readline support
  - script-friendly
- convenzioni
  - i comandi extra-SQL iniziano con “ . ”
  - i comandi SQL sono terminati da “ ; ”

SQLite – shell (cont.)

```
sqlite> .help    -- riformattato / accorciato
.dump ?TABLE? ...      Dump the database in an SQL text format
.explain ON|OFF        Turn output mode suitable for EXPLAIN on or off.
.help                  Show this message
.import FILE TABLE     Import data from FILE into TABLE
.indices TABLE         Show names of all indices on TABLE
.load FILE ?ENTRY?     Load an extension library
.mode MODE ?TABLE?     Set output mode where MODE is one of:
                         csv, column, html, insert, line, list, tabs, tcl
.output FILENAME       Send output to FILENAME
.read FILENAME         Execute SQL in FILENAME
.schema ?TABLE?        Show the CREATE statements
.tables ?PATTERN?      List names of tables matching a LIKE pattern
```

Esercizi

- Ricreate in SQLite i database visti in precedenza
- Cosa non è enforced dei vincoli espressi nello schema?
  - come si confronta SQLite a livello di vincoli supportati con MySQL e Postgres
- Effettuta un dump da MySQL di un database di dimensioni significative
  - Micro-benchmark time
    - Quanto è il tempo di import in SQLite?
    - Come si confronta con Postgres?
    - … e i tempi di query?

SQLite – utenza

- il controllo di accesso è delegato al file system
- non esistono quindi equivalenti degli statement GRANT / REVOKE
- locking lettori / scrittori implicito ed effettuato a livello di transazione

SQLite – miscellanea

Alcuni link come spunti di discussione ...

- benchmark / conronti di performance
  - http://www.sqlite.org/cvstrac/wiki?p=SpeedComparison
- success stories
  - http://www.sqlite.org/cvstrac/wiki?p=BigNameUsers
- amalgamation
  - http://www.sqlite.org/cvstrac/wiki?p=TheAmalgamation
- supporto per full text search
  - http://www.sqlite.org/cvstrac/wiki?p=FtsTwo

SQLite – core API

```
#include <stdio.h>
#include <sqlite3.h>
int main(int argc, char **argv){
  sqlite3 *db;
  char *zErrMsg = 0;
  int rc;
  if( argc!=3 ){
    fprintf(stderr, "Usage: %s DATABASE SQL-STATEMENT\n", argv[0]);
    exit(1);
  }
  rc = sqlite3_open(argv[1], &db);
  if( rc ){
    fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(db));
    sqlite3_close(db);
    exit(1);
  }
  rc = sqlite3_exec(db, argv[2], callback, 0, &zErrMsg);
  if( rc!=SQLITE_OK ){
    fprintf(stderr, "SQL error: %s\n", zErrMsg);
    sqlite3_free(zErrMsg);
  }
  sqlite3_close(db);
  return 0;
}
```

SQLite – core API (cont.)

```
static int callback(void *NotUsed, int argc, char **argv, char **azColName){
  int i;
  for(i=0; i<argc; i++){
    printf("%s = %s\n", azColName[i], argv[i] ? argv[i] : "NULL");
  }
  printf("\n");
  return 0;
}
```

(fonte: http://www.sqlite.org/quickstart.html)

- riferimenti
  - http://www.sqlite.org/capi3.html (manuale)
  - http://www.sqlite.org/capi3ref.html (API reference)

SQLite – estensioni

- dalla versione 3.3.6 SQLite permette di caricare estensioni da shared library
  - sintassi (alternative):

```
sqlite> .load filename
sqlite> SELECT load_extension('filename');
```

- una estensione può definire
  - nuove funzioni (aggregate o meno)
  - nuove collation sequences
- al momento del caricamento la funzione entry point `sqlite3_extension_init` delle estensioni viene invocata
- riferimenti:
  http://www.sqlite.org/cvstrac/wiki?p=LoadableExtensions

SQLite – estensioni (cont.)

- prototipo dell'entry point:

```
int sqlite3_extension_init(
    sqlite3 *db,           /* database connection */
    char **pzErrMsg,       /* Write error msgs here */
    const sqlite3_api_routines *pApi  /* API methods */
);
```

  - *db* è analogo a quanto ritornato da sqlite3_open, è da passare a `sqlite3_create_{function,collation}`
  - *`pzErrMsg`* può essere riempito con `sqlite3_mprintf` per ritornare messaggi di errore
  - *`pApi`* è il punto di accesso alla API di SQLite per l'estensione

SQLite – ext. & sicurezza

- problematiche di sicurezza
  - (purtroppo) non sono infrequenti applicazioni che permettono all'utente finale di inserire statement SQL che vengono passati verbatim ad un DBMS
- ciò non è particolarmente problematico con SQL standard ...
  - si rischia di compromettere “solo” il database
  - esistono meccanismi di blacklisting statement per statement
- … ma è più problematico per le estensioni
  - una estensione può eseguire codice C arbitrario
  - è possibile bypassare il blacklisting

SQLite – ext. & sicurezza (cont.)

- Ancora problematiche di sicurezza
  - SQLite prevede un master lock per le estensioni
    - è attivo di default (i.e. non è possibile caricarne)
    - è disattivabile via `sqlite3_enable_load_extension();`
  - Nel *top-level* distribuito nei pacchetti Debian è disattivato per default
    - se usate altre distribuzioni di SQLite può essere necessario ricompilarlo per potere caricare le estensioni dal top-level
  - maggiori info su http://www.sqlite.org/cvstrac/wiki?p=LoadableExtensions

SQLite – es. di ext. function

```
// file ext-half.c
#include <sqlite3ext.h>            // Notazione: boilerplate, highlight
SQLITE_EXTENSION_INIT1

/* The half() SQL function returns half of its input value. */
static void halfFunc(sqlite3_context *context, int argc,
  sqlite3_value **argv)
{
  sqlite3_result_double(context, 0.5*sqlite3_value_double(argv[0]));
}
int sqlite3_extension_init(sqlite3 *db, char **pzErrMsg,
  const sqlite3_api_routines *pApi)
{
  SQLITE_EXTENSION_INIT2(pApi)
  sqlite3_create_function(db, "half", 1, SQLITE_ANY, 0, halfFunc, 0, 0);
  /* ... other user-defined functions/collation here ... */
  return 0;
}
// Fonte: http://www.sqlite.org/cvstrac/wiki?p=LoadableExtensions
```

SQLite – es. di ext. function (cont.)

- compilazione

```
$ ls -l ext-half.c
-rw-r--r-- 1 zack zack 548 2007-07-06 15:03 ext-half.c
$ gcc -shared -fPIC ext-half.c -o ext-half.so
$ ls -l ext-half.so ; file ext-half.so
-rwxr-xr-x 1 zack zack 5629 2007-07-06 15:04 ext-half.so
ext-half.so: ELF 32-bit LSB shared object, Intel 80386, version 1 (SYSV), not
   stripped
```

- loading & testing

```
$ export LD_LIBRARY_PATH=`pwd`:$LD_LIBRARY_PATH
$ sqlite3
SQLite version 3.3.17
Enter ".help" for instructions
sqlite> select half(3);
SQL error: no such function: half
sqlite> .load ext-half.so
sqlite> select half(3);
1.5
```

SQLite – extension API

- punti di partenza per sopravvivere alla API reference:
  - `sqlite3_result_*`
    - e.g. `sqlite3_result_text`
    - insieme di funzioni usabili da un'estensione per restituire valori all'engine SQLite
    - effettuano la conversione da tipi C a tipi SQLite
  - `sqlite3_value_*`
    - e.g. `sqlite3_value_text`
    - duali alle funzioni di cui sopra
    - insieme di funzioni per ottenere dall'engine SQLite i parametri passati ad una funzione
    - effettuano la conversione da tipi SQLite a tipi C

Esercizi

- Implementate le estensioni SQLite che aggiungano supporto per le seguenti *funzioni*:
  - `crypt(s)` – hash crittografico di una stringa, random salt
    - Simile: `md5sum(s)` – checksum di una stringa
  - pwdcheck(s, p) – verifica se la stringa s corrisponde alla password crittata p
  - `vercmp(v1, v2)` – ordinamento lessicografico tra versioni, ritorna -1, 0, 1 (rispettivamente: v1 < v2, v1 = v2, v1 > v2)
- Implementate un'estensione che fornisca una libreria di funzioni core per manipolare date ed orari in SQLite, usando STRING come tipo di supporto

Collation sequences

- Una “collation sequence” (o “collation) è un (*pre)ordine totale* su un insieme di elementi
  - e.g.: l'ordine alfabetico su stringhe (cfr. `sort`)
  - e.g: l'ordine numerico su stringhe (cfr. `sort -n`)
  - e.g.: Unicode collation algorithm
  - Nota: collation sequence ≠ collation algorithm
  - È un pre<u>ordine</u> nel caso di valori numerici
    - e.g. 30 vs 30.0 vs 3e1
- Gli ordinamenti sono rilevanti per i DBMS
  - Clausole del DQL: WHERE / SORT BY / GROUP BY / ...
  - Indicizzazione
  - ...

SQLite – collations

- SQLite usa collations ogni volta che 2 valori *testuali* vengono confrontati
- Collations built-in
  - BINARY – usa `memcmp()`, collation di *default*
  - NOCASE – come binary, ma case insensitive su ASCII
  - RTRIM – come binary, ma rimuove trailing white space
  - Nuove collation user-defined possono essere definite con il meccanismo delle estensioni (hold your breath …)
- Uso di collation: si richiede a CREATE TABLE time

```
CREATE TABLE t (
  a STRING,                  -- collation: BINARY (default)
  b STRING COLLATE BINARY,   -- collation: BINARY
  c STRING COLLATE NOCASE,   -- collation: NOCASE
  d STRING COLLATE RTRIM     -- collation: RTRIM );
```

- http://www.sqlite.org/datatype3.html#collation

SQLite – user-defined collations

- Extension API per definire nuove collation sequences
  - API di base:

```
int sqlite3_create_collation(sqlite3*,
  const char *zName, int eTextRep, void*,
  int(*xCompare)(void*,int,const void*,int,const void*));
```

<len, string> <len, string>

  - Argomenti
    - Connessione al DB
    - Nome della nuova collation sequence
    - Char. encoding atteso (SQLITE_{UTF8,16LE,16BE})
    - User data (passato ad ogni invocazione, primo arg.)
    - Funzione di confronto user-defined
      - Ritorna <0, 0, >0 (risp.: primo arg { <, =, > } secondo arg)

SQLite – es. di user collation

```
// file ext-revcoll.c
#include <sqlite3ext.h>              // Notazione: boilerplate, highlight
SQLITE_EXTENSION_INIT1
/* user defined collation which simply reverse the order of memcmp() */
static int myRevCmp(void *userdata,
    int len1, const void *s1, int len2, const void *s2)
{
  int cmp, minlen;
  minlen = len1 < len2 ? len1 : len2;
  cmp = memcmp(s1, s2, minlen);
  if (cmp == 0) return 0;
  return cmp < 0 ? 1 : -1;
}
int sqlite3_extension_init(sqlite3 *db, char **pzErrMsg,
  const sqlite3_api_routines *pApi)
{
  SQLITE_EXTENSION_INIT2(pApi)
  sqlite3_create_collation(db, "MYREV", SQLITE_UTF8, NULL, &myRevCmp);
  return 0;
}
```

SQLite – es. di user collation (cont.)

```
$ gcc -shared -fPIC ext-revcoll.c \
    -o ext-revcoll.so
$ export LD_LIBRARY_PATH=`pwd`
$ sqlite3 test.sqlite
SQLite version 3.6.11
Enter ".help" for instructions
Enter SQL statements terminated with a
";"
sqlite> CREATE TABLE t (s STRING);
sqlite> CREATE TABLE t2 (s STRING
COLLATE MYREV);
SQL error: no such collation sequence:
MYREV
sqlite> .load ext-revcoll.so
sqlite> CREATE TABLE t2 (s STRING
COLLATE MYREV);
sqlite>
```

```
sqlite> INSERT INTO t VALUES ("foo");
sqlite> INSERT INTO t VALUES ("bar");
sqlite> INSERT INTO t VALUES ("baz");
sqlite> INSERT INTO t2 VALUES ("foo");
sqlite> INSERT INTO t2 VALUES ("bar");
sqlite> INSERT INTO t2 VALUES ("baz");
sqlite>
sqlite> SELECT * FROM t ORDER BY s;
bar
baz
foo
sqlite> SELECT * FROM t2 ORDER BY s;
foo
baz
bar
sqlite>
```

Esercizi

- Implementate le estensioni SQLite che aggiungano supporto per le seguenti *collation*:
  - `DATE` – ordinamento di date nella forma YYYY/MM/DD
  - ACCLESS – ordinamento alfabetico, ignorando gli accenti
  - `VERCMP` – ordinamento lessicografico tra versioni
    - … lo stesso, con la semantica di versioni Debian Policy ...