UNIVERSITÀ DEGLI STUDI DI PARMA
Dipartimento di Matematica e Informatica

Corso di Laurea in Informatica

Refactoring per un'applicazione di reportistica: portabilità ed efficienza

Candidato: Jacopo Freddi

Relatore: Prof. Enea Zaffanella

Anno Accademico 2013/2014

Indice

Introduzione

Scopo della tesi è presentare il refactoring di un'applicazione basata su `SQLite` [11], effettuato per agevolare il porting verso `PostgreSQL` [9] ed altri sistemi, e valutarne l'impatto sull'efficienza. Vengono presentate alcune riflessioni sull'efficacia dell'astrazione della libreria e sull'effettiva necessità di una suite di funzionalità personalizzabile.

Si richiede che le modifiche intacchino il meno possibile il lato `SQLite` e le sue prestazioni: le trasformazioni strutturali devono dunque essere il più possibile marginali. Talvolta, soluzioni diverse da quella adottata nell'applicazione originaria e meglio sfruttabili dal lato `PostgreSQL` sono state scartate per non intaccare le prestazioni su `SQLite`. Nonostante l'attenzione spesa da questo punto di vista, si sono rese necessarie alcune modifiche per garantire l'interoperabilità dei sistemi.

Durante l'operazione di refactoring è stata prestata particolare attenzione a fornire un risultato finale pulito, seguendo i principi SOLID [3] e RRID [2] quando possibile.

La tesi si divide in quattro capitoli ed una conclusione:

- il primo capitolo fornisce una panoramica sul funzionamento e sull'utilizzo dell'applicazione. Vengono illustrate brevemente alcune nozioni preliminari relative all'utilizzo dei Database e del Google Protobuf [12], le principali tecnologie utilizzate dall'applicazione. Vengono presentati i principi di programmazione seguiti.

- il secondo capitolo illustra la struttura originale dell'applicazione e le modifiche che ad essa sono state apportate, motivandole. Vengono illustrati inoltre i pregi ed i difetti delle soluzioni adottate, quando ritenuto rilevante.

- il terzo capitolo mostra le differenze tra i sistemi di riferimento, esemplificandole e motivando le scelte effettuate per rendere i detti sistemi equivalenti.

- il quarto capitolo presenta un confronto delle prestazioni del sistema su database di medie e grandi dimensioni, nelle sue varianti:
  - sistema originario basato su `SQLite`;
  - sistema finale basato su `SQLite`;
  - sistema finale basato su `PostgreSQL`;
  - sistema finale basato su `MySQL` [4];
  - sistema ottimizzato basato su `PostgreSQL`.

  Vengono inoltre presentate alcune considerazioni sulla mole di lavoro necessaria per effettuare le modifiche e sull'efficacia del porting rispetto alla necessità di supportare altri sistemi.
- nella conclusione, oltre a riassumere il lavoro svolto, vengono raccolti alcuni suggerimenti per gli sviluppi futuri dell'applicazione.

Capitolo 1

Il contesto applicativo

In questa sezione si andranno a presentare i componenti principali dell'applicazione e si illustrerà come questi interagiscono. Per una breve descrizione dei principali termini utilizzati si rimanda al glossario.

1.1 L'ambiente di sviluppo

Eclair [15] è una piattaforma software di analisi e verifica del codice sviluppato dalla start-up BUGSENG s.r.l. presso il Dipartimento di Matematica e Informatica dell'Università degli Studi di Parma. Le applicazioni basate su Eclair sono varie, in alcuni casi basate fortemente su standard di codifica riconosciuti internazionalmente (ad esempio, MISRA-C [16]) e spesso personalizzate per l'utente.

Eclair si integra nel processo di compilazione e collegamento del codice in via di sviluppo, in modo completamente trasparente per l'utente. La configurazione del sistema fa in modo che i normali comandi per la costruzione del codice eseguibile (ad esempio `make`) invochino implicitamente Eclair. L'applicazione considerata in questo elaborato si occupa della generazione automatica dei rapporti di violazione degli standard di codifica. Questa operazione è utile nel processo di sviluppo di software che deve rispettare un certo standard di codifica.

L'applicazione effettua le sue analisi (in base alle configurazioni fornite) e produce come risultato una serie di messaggi di diagnostica. Tali messaggi hanno la stessa struttura dei normali messaggi di diagnostica forniti dal compilatore in uso e possono essere formattati in modo identico, così da poter essere visualizzati all'interno dell'IDE nel modo canonico. Se richiesto dalla configurazione, gli stessi messaggi possono essere serializzati usando la libreria `protobuf` di Google [12], ed essere quindi processati off-line da altri

strumenti di supporto del sistema Eclair. A partire dal file protobuf viene creato un database per i report di violazione, usando il DBMS `SQLite` [11]. Il database viene poi interrogato per ottenere numerosi tipi di rapporti puntuali e statistiche riassuntive sul programma analizzato. Durante il build di una applicazione una pluralità di unità di traduzione (in seguito chiamate "frame") vengono compilate indipendentemente, spesso condividendo larghe porzioni di codice (ad esempio, tramite inclusione di file header): come conseguenza, è possibile avere report duplicati che fanno riferimento alla stessa violazione.

Il lavoro qui presentato è stato svolto su `report`, il modulo che traduce il file protobuf nel database ed interroga quest'ultimo. `report` è per molti aspetti simile ad una comune applicazione di reportistica, ma se ne differenzia per due peculiarità:

- deve anche occuparsi della fase di caricamento del database, quando in generale le applicazioni di reportistica lavorano in sola lettura su un database già caricato. In aggiunta, deve rendere possibile caricare il database in modo incrementale (per supportare le ricompilazioni parziali).

- avendo importanti requisiti di efficienza, deve rinunciare alla 'generalità' fornita dalle altre applicazioni di reportistica. Il grado di configurazione e personalizzazione offerto è quindi minore, in quanto le operazioni possibili sono integrate nell'applicazione stessa.

1.1.1 L'applicazione

Nel codice originale di `report` l'elaborazione dei dati da parte dell'applicazione e la gestione del database coesistono senza alcuna distinzione. Le operazioni principali effettuate dall'applicazione sono:

- `-create-db`
  Crea lo schema del database senza dati, pronto per essere caricato con le informazioni dei report.

- `-load=FILE`
  Carica i dati contenuti nel file specificato nel database. Questa operazione è la più delicata dal punto di vista delle performance: come sarà possibile notare in seguito, il caricamento dei dati può richiedere molto tempo.

- `-show`
  Mostra le violazioni rilevate in formato testuale. L'output può essere

manipolato in modo da poter essere integrato e visualizzato in un qualsiasi IDE. È possibile mostrare differenti tipi di raggruppamento per le violazioni: la più comune raggruppa le violazioni prima per regola e poi per directory.

- `-annotate`
  Annota le violazioni direttamente sul codice sorgente.

- `-summarize=OPTION`
  Presenta un sommario della situazione attuale (numero di violazioni), usando il tipo di aggregazione specificata. Questa operazione può essere usata come strumento di controllo dei progressi del team di sviluppo o per ripartire il lavoro di adeguamento del codice allo standard di codifica scelto.

- `-diff=DIFF_DB`
  Permette di mostrare le differenze tra due database ottenuti da compilazioni diverse. Queste informazioni sono utili per avere una misura più precisa dei progressi ottenuti in un certo periodo di tempo.

1.2 Considerazioni preliminari

Prima di analizzare la struttura interna di `report` ed i cambiamenti ad essa apportati, è consigliabile trattare sinteticamente le dipendenze esterne dell'applicazione.

1.2.1 Database

La caratteristica più rilevante di un database relazionale è appunto la sua capacità di esprimere relazioni tra le informazioni. Tramite i metadati e le strutture definite al suo interno, con l'aiuto di un buon set di funzionalità ed un motore abbastanza potente è possibile estrapolare informazioni molto interessanti in tempi relativamente brevi e senza bisogno di gestire manualmente informazioni grezze. Un database garantisce inoltre stabilità e coerenza dei dati.

In questo caso particolare, al database è richiesta una proprietà particolare al di sopra delle altre: efficienza nel tempo di esecuzione. L'analisi di progetti software di dimensioni relativamente modeste può generare database di dimensioni considerevoli. È dunque essenziale che il DBMS offra meccanismi per limitare le inefficienze durante la fase di caricamento dati.

Anche semplici accorgimenti possono ridurre notevolmente il tempo totale di esecuzione.

Una funzionalità che durante il lavoro ha richiesto particolare attenzione è il supporto ai prepared statement. Essendo le query operazioni basilari di comunicazione con il database, è facile giungere alla conclusione che queste vengano eseguite molto spesso e che occupino la maggior parte del tempo di esecuzione: evitare l'esecuzione di un passaggio (in questo caso l'analisi della sintassi) può portare ad un vantaggio notevole. Si noti che il prepared statement è una funzionalità messa a disposizione dal database, non dalla libreria software: gli oggetti che una libreria può mettere a disposizione per manipolare un prepared statement non *contengono* lo statement, ma solo le informazioni necessarie per il suo utilizzo.

Altre operazioni sono dispendiose e vanno quindi limitate il più possibile: ad esempio il "`commit`", query particolare che segnala la fine di una sequenza di operazioni logicamente correlate. Ad un `commit` segue (in molti DBMS) una scrittura sul disco rigido delle modifiche effettuate: questa operazione è drammaticamente lenta a causa dei limiti hardware della macchina, pertanto va razionalizzata il più possibile.

Lo schema del database

Le tabelle principali da cui vengono estratte informazioni sono `hreport` ed `area`. `hreport` sta per 'hashed report' e fattorizza le violazioni duplicate. Mantiene informazioni su:

- regola infranta (identificata da un codice, tramite il quale è possibile risalire al suo testo);
- tipo di frame (può essere un comando, una unità di traduzione o l'intero programma);
- codice hash del rapporto di violazione, utilizzato per distinguere i report duplicati.

`area` identifica le aree di testo a cui si riferiscono i report (un esempio di area viene fornito più avanti). Le informazioni sulle aree vengono arricchite con:

- le posizioni di inizio e fine dell'area (file, riga e colonna);
- il tipo di area:

  `culprit`: l'area contiene l'entità responsabile della violazione;

  `evidence`: l'area contiene prove dell'avvenuta violazione;

  `context`: l'area contiene il contesto in cui è avvenuta la violazione;

- messaggio della violazione.

Le informazioni collegate ad un report possono essere anche più elaborate (ad esempio se la violazione è stata rilevata in una porzione di codice prodotta dall'espansione di una macro).

In totale lo schema comprende 34 tabelle ed una vista, sulle quali agiscono 35 trigger. Di queste tabelle, 5 fungono da 'enumerazioni' per i dati immutabili (come i tipi di frame o le regole dello standard di codifica). Sono presenti altre tabelle per memorizzare i frame, i file (anch'essi necessitano di un codice hash a causa di potenziali nomi duplicati) ed altre informazioni rilevanti. La tabella con il maggior numero di riferimenti verso altre tabelle è `area`, che mantiene 7 vincoli di chiave esterna. In totale i vincoli di chiave esterna sono 37.

1.2.2 Google Protocol Buffer

Google Protocol Buffer [12] (abbreviato in Google Protobuf, o semplicemente protobuf) è un meccanismo di serializzazione dati molto simile ad XML: vengono definite delle 'classi' costituite da attributi strutturati, le quali vengono poi utilizzate per organizzare i dati ed arricchirli. Il vantaggio dei protobuf consiste nel fatto che la libreria che li gestisce genera automaticamente il codice sorgente necessario per poterli utilizzare, direttamente da un file di descrizione. Inoltre, XML è molto costoso in termini di tempo e spazio a causa del numero di caratteri necessari per ogni tag, mentre i messaggi del protocollo protobuf sono stati progettati per essere leggeri e veloci.

1.2.3 Analisi di una violazione

Viene qui riportato un esempio di rapporto di violazione preso dalla versione 5.2.2 della libreria Open Source `Lua` [14]. Questa libreria è stata sviluppata senza aderire agli standard di codifica adottati in alcuni contesti a livello industriale, pertanto è facile ottenere una discreta varietà di report. Il rapporto di violazione qui presentato segnala delle parentesi mancanti in un'espressione booleana.

```
153:    LUA_API int lua_absindex (lua_State *L, int idx) {
154:      return (idx > 0 || ispseudo(idx))
155:             ? idx
156:             : cast_int(L->top - L->ci->func + idx);
157:    }
```

```
/lua-5.2.2/src/lapi.c:154.11-154.17: violated rule LP1.158 (The operands
of a logical ‘&&’ or ‘||’ shall be parenthesized if the operands
contain binary operators.) Loc #1 [culprit: child expression is not
in parentheses]
/lua-5.2.2/src/lapi.c:154.19-154.20: Loc #2 [evidence: main operator]
/lua-5.2.2/src/lapi.c:154.11-154.34: (MACRO) Loc #3 [context: main
expression]
/lua-5.2.2/src/lapi.c:44.58: Loc #4 [context: expanded from macro
‘ispseudo’]
```

Possono essere messi in evidenza i campi che compongono il rapporto:

- `/lua-5.2.2/src/lapi.c:154.11-154.17`: il nome del file, comprensivo di directory, in cui è stata rilevata la violazione, insieme all'area di testo in cui si verifica, caratterizzata dalle coordinate di inizio e fine della violazione. Un IDE può sfruttare queste informazioni per evidenziare l'area in cui avvengono le violazioni. Un'area può iniziare in un file e finire in un altro file (ad esempio in caso di direttive di preprocessore che iniziano in un file e finiscono in un altro tramite inclusione).
- `violated rule LP1.158`: il nome della regola violata (in questo caso: LP1 (codice che identifica il Joint Strike Fighter Air Vehicle C++ Coding Standard [13] della Lockeed Martin), regola 158);
- `The operands of a logical ‘&&’ or ‘||’ shall be parenthesized if the operands contain binary operators`: il testo della regola violata;
- `Loc #1 [culprit: child expression is not in parentheses]`: l'area evidenziata identifica il 'culprit', l'entità che ha violato la regola (in questo caso una sottoespressione);
- `/lua-5.2.2/src/lapi.c:154.19-154.20: Loc #2 [evidence: main operator]`: possono seguire una o più prove della violazione all'interno dell'area definita prima;
- `/lua-5.2.2/src/lapi.c:154.11-154.34: (MACRO) Loc #3 [context: main expression]`: possono seguire uno o più contesti di violazione;
- `/lua-5.2.2/src/lapi.c:44.58: Loc #4 [context: expanded from macro ‘ispseudo’]`: se l'area precedentemente indicata è stata ottenuta espandendo una macro, viene indicata la macro generatrice (se una macro espansa presenta altre macro, queste vengono tracciate ricorsivamente).

1.3 Principi di progettazione

Durante la progettazione delle soluzioni software si è deciso di seguire, per quanto possibile, alcune 'buone norme' che aumentano la qualità del codice e ne riducono i rischi di malfunzionamento. Le regole seguite sono qui presentate e spiegate sommariamente.

1.3.1 SOLID

L'acronimo SOLID [3] copre 5 regole fondamentali per la buona programmazione ad oggetti:

- SRP - Single Responsibility Principle
  Una classe (o un metodo) deve avere una sola funzione: se una classe può essere utilizzata per più cose diverse occorre dividerla affinché per ogni necessità esista una classe dedicata.
- OCP - Open Closed Principle
  Una classe deve essere aperta alle estensioni ma chiusa rispetto alle modifiche. In altre parole, deve essere facile modificare la classe per migliorarla, ma non deve essere necessario aggiungere, togliere o alterare le sue funzionalità.
- LSP - Liskov Substitution Principle
  Una sottoclasse deve poter essere sostituibile alla sua classe base, senza alterare il funzionamento del programma.
- ISP - Interface Segregation Principle
  È preferibile avere più interfacce specifiche rispetto ad una singola interfaccia generica.
- DIP - Dependency Inversion Principle
  Bisogna dipendere dalle astrazioni, non dalle concretizzazioni. In altre parole, bisogna adattare il codice al progetto e non il progetto al codice.

1.3.2 RAII/RRID

Nella programmazione `C/C++` è frequente l'utilizzo dei puntatori, specialmente quando si desidera sfruttare il polimorfismo dinamico della programmazione a oggetti. Questa categoria di oggetti è molto delicata da usare, in quanto potenzialmente dannosa: si rischia di scrivere in zone di memoria non lecite, di leggere da zone non più valide o non ben formattate, e se usati male i puntatori possono generare zone teoricamente vuote ma ancora segnate come in

uso, che accumulandosi esauriscono rapidamente la memoria disponibile. Uno dei metodi più efficaci per scongiurare questi pericoli consiste nell'adozione degli idiomi RAII/RRID [2]: Resource Acqisition Is Initialization, Resource Release Is Destruction.

- RAII
  Tutte le risorse necessarie per il puntatore devono essere acquisite nel costruttore.

- RRID
  Tutte le risorse acquisite dal puntatore devono essere rilasciate quando questo viene distrutto.

Questi idiomi, se usati correttamente, garantiscono che le risorse vengano rilasciate anche nei cammini di esecuzione eccezionali. Un esempio pratico dell'idioma RRID viene presentato per la classe `R_Stmt`, più avanti.

Capitolo 2

Modifiche Strutturali

2.1 La situazione iniziale

Per evitare il degrado di performance, non è stato ritenuto conveniente utilizzare una libreria di interfacciamento generico, preferendo invece convertire manualmente le parti interessate. Si noterà poi come questa scelta abbia avuto conseguenze sull'evoluzione della struttura.
La modifica del codice ha interessato essenzialmente due entità dell'applicazione:

- `Stmt`
- `DB_Utils`

Il capitolo inizia con una rassegna generale delle caratteristiche principali di ogni libreria di interfacciamento utilizzata, evidenziandone i tratti caratteristici. Vengono poi mostrate le evoluzioni subite dalle entità sopra citate, motivando le scelte effettuate. Infine, si presentano gli altri aspetti dell'applicazione, mostrando l'impatto derivante dai cambiamenti apportati alle sezioni principali.

2.2 Le librerie utilizzate

Ogni backend implementato si appoggia ad una libreria specifica per interfacciarsi al database e portare a termine i suoi compiti. Il requisito fondamentale è che la parte di interfacciamento a `SQLite` sia intaccato il meno possibile, eventualmente anche a discapito dell'efficienza (e semplicità) degli altri backend. Il punto interessante di quest'argomento è che ogni libreria, pur avendo costrutti più o meno simili, è strutturata in modo diverso: svolgere

un compito nello stesso modo con strumenti diversi si è rivelata una sfida non banale, che spesso ha richiesto complicazioni notevoli per implementare un'applicazione equivalente a quella di partenza ed a volte ha reso necessario modificarne leggermente il comportamento.

2.2.1 SQLite

L'applicazione originale si basa sulla libreria `C` [11] di gestione di `SQLite`. Le entità utilizzate sono principalmente puntatori a `sqlite3_db`, una struttura di gestione del database, e `sqlite3_stmt`, che gestisce invece i prepared statement. Le interfacce degli oggetti non forniscono metodi: le operazioni vengono svolte da funzioni statiche a cui questi vengono passati come parametri, seguendo il paradigma `C`.

Le query non parametriche vengono eseguite direttamente sulla struttura di gestione del database, senza bisogno di creare strutture particolari. Anche i parametri di sessione vengono modificati direttamente su `sqlite3_db`. I prepared statement vengono invece creati come strutture a sé stanti: i parametri possono essere assegnati allo statement senza necessità di coinvolgere il database, così come l'operazione di `reset`, che permette di riutilizzarli con parametri diversi. Quando un parametro viene legato, ne viene specificato l'indice all'interno della query: questa possibilità garantisce un'elevata flessibilità all'interno dell'applicazione.

È disponibile un solo comando per l'esecuzione di una query, ma viene usato in due modi diversi dall'applicazione. In caso di query che producono risultati viene usato direttamente `sqlite3_step()`, un comando della libreria che produce una nuova tupla di risultato ad ogni invocazione (comportamento tipico di un iteratore). La funzione `exec`, presente nell'interfaccia di `DB_Utils`, viene invece utilizzata per le query che non ritornano alcun valore (ad esempio, gli inserimenti) e si limita ad invocare `sqlite3_step()` assicurandosi che l'esecuzione vada a buon fine. `SQLite` non mette a disposizione, nella sintassi, un costrutto per recuperare informazioni sulle righe inserite: la libreria offre `last_insert_rowid()` per ottenere il valore identificativo dell'ultima tupla inserita.
Ogni esecuzione, a prescindere dalla presenza o meno del risultato, restituisce un result code che fornisce informazioni sullo stato della connessione e sull'esecuzione della query. Non vengono generate eccezioni dalla libreria, ma `report` può lanciarne in seguito al verificarsi di condizioni di errore.

2.2.2 PostgreSQL

La libreria scelta per PostgreSQL è libpqxx [18], una libreria Object-Oriented per il C++ che si appoggia sulla libreria C ufficiale, libpq [10]. libpqxx fornisce supporto a vari tipi di connessioni e transazioni, eccezioni personalizzate e supporto alle funzionalità specifiche di PostgreSQL. La maggior parte delle funzionalità si basa sui due oggetti principali della libreria, che implementano rispettivamente la connessione al database e la transazione. Pressoché ogni elemento della libreria è un oggetto e presenta un'interfaccia completa e funzionale.

Le query non parametriche sono semplici stringe SQL che vengono passate come parametro di funzione ed eseguite. I prepared statement, prima di essere utilizzati, devono essere preparati dal database attraverso una sequenza di chiamate di funzione: durante la definizione dello statement vengono specificati i tipi dei parametri e viene assegnato allo statement un nome univoco. Perché uno statement parametrico sia eseguito occorre utilizzare il nome ad esso assegnato durante la definizione e presentare la lista di valori da assegnare ai parametri.

Ogni esecuzione di query produce un result set, che oltre ad implementare le funzionalità tipiche di un vector contiene le informazioni sull'esecuzione e gli eventuali risultati. Il reset non è contemplato dall'interfaccia della libreria in quanto implicitamente automatico: ogni nuova invocazione è accompagnata dal suo assegnamento di parametri, che viene eseguito su una nuova istanza della struttura preposta a gestirli. In caso di errore vengono lanciate le eccezioni rilevanti. Come sarà evidenziato più avanti, questa libreria presenta, pur nella pulizia dell'interfaccia, alcune rigidità che hanno influito negativamente sulla struttura del backend.

2.2.3 MySQL

Alla fine del refactoring è stato ritenuto opportuno valutare quanto il lavoro di implementazione di altri backend fosse stato semplificato. È stato dunque implementato parzialmente il supporto a MySQL [4]. La libreria scelta per implementare il supporto al nuovo backend è mysqlpp [5], libreria C++ ufficiale. MySQL è stato di recente acquisito da Oracle, mantenendo però la sua semplicità e gratuità.
Anche mysqlpp presenta una forte connotazione Object-Oriented ed una buona integrazione con la STL. La struttura principale è Connection, l'oggetto di gestione della connessione al database.

Per eseguire una query occorre creare un oggetto Query legato alla connessione. Gli statement parametrici sono particolari sottoclassi di Query. La

procedura di assegnamento dei parametri può essere svolta in più modi, tra cui quello utilizzato nella versione originale dell'applicazione: questa caratteristica ha semplificato notevolmente lo sviluppo del backend. L'oggetto responsabile della gestione dei parametri mette a disposizione dello sviluppatore un metodo apposito che si prende cura di gran parte dell'operazione di reset. Lo stato di esecuzione dello statement può essere interrogato tramite metodi appositi, mentre le situazioni di errore portano ad eccezioni personalizzate. Sono inoltre a disposizione tre tipi di oggetti in cui è possibile memorizzare i risultati dell'esecuzione di una query, differenziati a seconda dell'utilizzo previsto per le informazioni restituite.

2.3 Modifiche a `Stmt`

Originariamente, una struct `Stmt` era poco più di un wrapper per un puntatore alla struttura `sqlite3_stmt`, contenente tutte le informazioni sulla query di riferimento. Le operazioni principali che le varie strutture di `report` eseguono sullo statement sono:

- **assegnazione dei parametri**: nel caso di query parametriche, prima di eseguire è necessario assegnare un valore ad ogni parametro libero. La procedura di binding viene eseguita dal `DB_Utils` tramite funzioni templatiche rispetto al tipo di parametro inserito. Nonostante l'indipendenza dall'oggetto di connessione al database, le operazioni di binding passano attraverso metodi di `DB_Utils`: come conseguenza, la dipendenza logica degli statement rispetto alla connessione si traduce in dipendenza effettiva degli oggetti.

- **esecuzione dello statement**: anche in questo caso, nonostante l'esecuzione dello statement non dipenda dalla struttura di controllo del database, il metodo effettivamente invocato fa parte dell'interfaccia di `DB_Utils`. Ci sono due tipi diversi di statement dal punto di vista dell'esito dell'esecuzione.

  - *Statement che ritornano risultati*
    Le query di selezione sono tipici esempi di statement che ritornano risultati. La funzione `sqlite3_step` viene chiamata ripetutamente per ottenere, una alla volta, tutte le righe del risultato. Notare che queste query possono anche ritornare un set vuoto (ad esempio quando si vuole accertare l'esistenza o meno di una particolare tupla in una tabella).

  - *Statement che non ritornano niente, o il cui risultato non viene utilizzato*
    Le query di inserimento, eliminazione, modifica o definizione dei dati (ad esempio, la creazione di indici o tabelle) non hanno valore di ritorno, pertanto la loro gestione è molto più semplice. In caso sia necessario un feedback dall'operazione svolta (ad esempio, il numero di righe modificate da una query di `update`) sono a disposizione metodi accessori per rilevare queste informazioni. Altri database hanno implementato estensioni non standard di `SQL` per rendere possibile a queste query ritornare informazioni allo stesso modo di una query di selezione.

- **rilevamento dello stato dello statement**: la libreria di `SQLite` mette a disposizione un meccanismo di result codes per monitorare lo stato di esecuzione della query. I result code significativi per l'applicazione sono:

  - `SQLITE_OK` - indica che l'esecuzione della query ha avuto successo;
  - `SQLITE_DONE` - indica che la query di selezione non ha prodotto risultati. In alternativa, per query che producono righe di risultato, indica che tutte le righe richieste sono state scaricate.
  - `SQLITE_ROW` - indica che ci sono altre righe di risultato dopo quella appena scaricata.
  - qualsiasi altro result code viene interpretato come errore.

- **reset**: uno `Stmt` parametrico può essere resettato per permettere alla query di essere eseguita nuovamente (cambiando parametri).

2.3.1 Le evoluzioni dell'architettura

Gli `Stmt` sono una parte fondamentale della struttura di funzionamento di `report`, pertanto la loro reimplementazione ha richiesto uno sforzo particolarmente intenso. Gran parte della difficoltà di traduzione consisteva nella necessità di adattare due (in seguito tre) librerie diverse ad una interfaccia comune. Per raggiungere questo obiettivo sono stati necessari vari passaggi di modifica, coinvolgendo gran parte del meccanismo di utilizzo degli `Stmt`.

La gerarchia delle classi

Gli `Stmt` generati da `Reusable_Query` vengono manipolati in modo leggermente diverso da quelli non riutilizzabili: la prima ipotesi di refactoring mi-

rava a sfruttare l'ereditarietà multipla per modellare queste differenze. La struttura risultante è visibile nella figura 2.1.

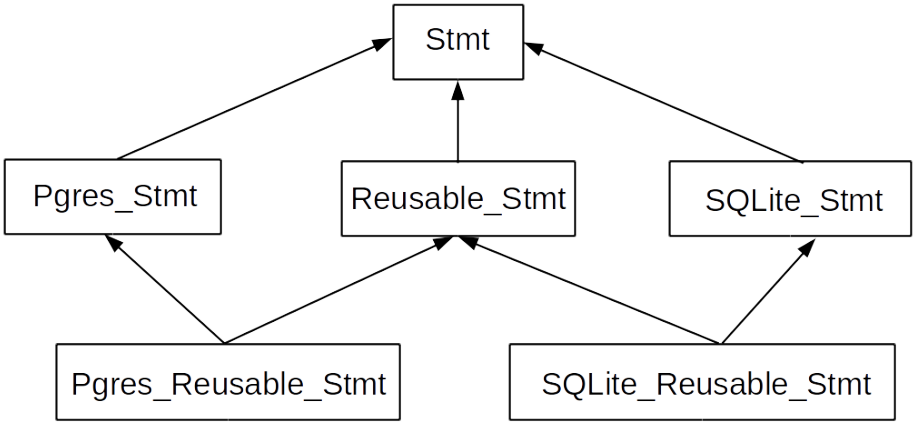

Figura 2.1: Diamond Problem nella prima bozza di ristrutturazione

In questa struttura, tramite overriding era possibile ridefinire facilmente il comportamento delle classi a seconda del database di riferimento e del tipo di statement. La struttura soffriva però di una complessità molto elevata e del famigerato Diamond Problem [1]. Il Diamond Problem emerge nell'utilizzo dell'eredità multipla, che in `C++` è particolarmente delicata da trattare: sono possibili almeno 4 tipi diversi di ereditarietà, le combinazioni di ereditarietà `public` e `private` con eredità semplice e virtuale. Quando si utilizza l'ereditarietà multipla si corre il rischio di avere due oggetti della classe base (nel nostro caso `Stmt`) su cui possono essere invocati i metodi e nessuno strumento di controllo per decidere su quale dei due il metodo debba essere invocato. Il problema viene risolto utilizzando l'ereditarietà virtuale, grazie alla quale l'oggetto base viene istanziato una sola volta. Restano però irrisolti altri problemi: se una o più classi nel percorso dalla classe più derivata (ad esempio `Pgres_Reusable_Stmt`) alla classe base (`Stmt`) ridefiniscono un metodo della classe base e la classe più derivata non lo ridefinisce ulteriormente, quando questo metodo viene invocato sulla classe derivata non è possibile prevedere quale versione del metodo verrà chiamata.

Si è concluso che l'ereditarietà multipla non era necessaria: l'unica cosa che distingueva gli statement riutilizzabili dai non riutilizzabili era la procedura di creazione e distruzione, che nel caso di statement riutilizzabili coinvolgeva anche la cache di `DB_Utils`. Per questo motivo si è deciso di eliminare la distinzione gerarchica tra `Stmt` e `Reusable_Stmt` (figura 2.2) so-

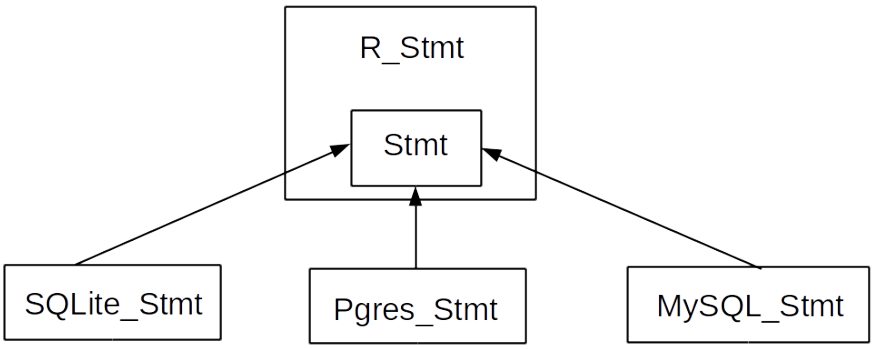

Figura 2.2: Semplificazione della struttura con rimozione del Diamond Problem

stituendola con un campo booleano. Non essendo previste altre tipologie di utilizzo degli `Stmt` (osservazione confermata dalla stabilità di questa struttura durante il periodo di utilizzo dell'applicazione) questa scelta risulta essere ragionevolmente priva di controindicazioni.

Polimorfismo dinamico

Per poter sfruttare le proprietà dinamiche del polimorfismo, è necessario che il tipo statico di uno statement possa essere diverso da quello dinamico. In `C++` questo obiettivo viene comunemente raggiunto utilizzando i puntatori. Utilizzando un puntatore a `Stmt`, è possibile avere un tipo di dato uniforme in tutto il codice il cui comportamento differisce a seconda del backend selezionato. Lato negativo: avere puntatori liberi nel codice è una pessima pratica di programmazione, prona a memory leak e dangling pointers. La soluzione più comune è lo smart pointer, un wrapper per il puntatore che gestisce automaticamente costruzione e distruzione. Gli smart pointer sono entrati nello standard solo con `C++11`, mentre l'applicazione è basata su `C++03`. Per una serie di motivi è stato valutato prematuro passare a `C++11`, dunque non sono state sfruttate queste funzionalità aggiuntive. Alcune librerie esterne (ad esempio `Boost` [6]) offrono le stesse funzionalità, ma è stato ritenuto sconveniente dover dipendere da una intera libreria solo per utilizzare uno smart pointer. L'alternativa adottata è un wrapper personalizzato, scritto secondo le regole dell'idioma RAII/RRID [2]. `R_Stmt` (che sta per RRID Statement) è un wrapper per puntatori a `Stmt`, completo di distruttore exception safe, che offre all'esterno le funzionalità essenziali per manipolare lo `Stmt` contenuto.

Al termine del refactoring gli Stmt originali sono stati sostituiti da R_Stmt, inizializzati con un riferimento ad uno Stmt sull'heap. Questo riferimento, fornito dal DB_Utils stesso, è un puntatore alla specifica implementazione di Stmt per quel backend. Su R_Stmt possono essere eseguite le stesse operazioni che prima venivano eseguite sugli Stmt, a parte l'interrogazione dei codici di risultato (sostituiti da più eleganti metodi booleani).
Le varie implementazioni di Stmt contengono, oltre alla ridefinizione delle funzioni virtuali della classe base, alcuni metodi aggiuntivi utilizzati dalla struttura di gestione del database. Raggiungere tali metodi richiede uno static_cast esplicito verso la classe derivata di Stmt desiderata. Questa procedura, benché 'sporca', è resa sicura dal fatto che questi cast vengono effettuati unicamente all'interno di DB_Utils, la stessa classe che genera gli Stmt. Operando con lo stesso DB_Utils, si è certi che gli static_cast definiti siano leciti. È impossibile che uno Stmt di un backend venga interrogato da un DB_Utils operante su un altro DBMS: il backend viene scelto a runtime e non viene cambiato per tutta la durata delle operazioni richieste, al termine delle quali sia il DB_Utils che gli Stmt eventualmente rimasti attivi vengono distrutti.

2.3.2 Prepared Statement

I prepared statement sono query particolari, che permettono di essere analizzate una sola volta ed eseguite più volte. Molti (se non tutti i) DBMS offrono un certo grado di supporto ai prepared statement. In sostanza, un prepared statement è una query in cui alcuni valori vengono sostituiti da 'segnaposti' identificati da caratteri particolari. Questi vengono successivamente sostituiti dai valori specificati e la query viene eseguita. Alcune considerazioni sulle differenze tra SQLite e PostgreSQL riguardo i prepared statement e sulle soluzioni adottate possono essere trovate nel paragrafo 2.5.1.

La libreria di SQLite prevede un sistema di assegnamento dei parametri molto flessibile: in ogni istante compreso tra la definizione e l'esecuzione dello statement è possibile legare un singolo parametro al corrispondente segnaposto specificandone la posizione ordinale. È quindi possibile scrivere codice del tipo:

```
Stmt r(db, "select ? as Col1, ? as Col2, ? as Col3");
db.bind(r, 2, "b");
db.bind(r, 3, "c");
db.bind(r, 1, "a");
sqlite3_step(r);
```

Il risultato della query è (a b c), nonostante l'ordine con cui i parametri sono stati legati sia diverso. Una tale flessibilità dipende dalla possibilità offerta dalla libreria di legare i parametri in modo indicizzato, specificando esplicitamente a quale segnaposto andava legato il parametro. In libpqxx questa possibilità non è stata contemplata: i parametri vengono legati ai segnaposto nell'ordine in cui vengono forniti al prepared statement. Si porta come esempio il seguente pezzo di codice, che secondo la libreria libpqxx è il modo standard per raggiungere l'obiettivo sopra illustrato:

```
transaction.prepare("mystatement",
    "select $1 as Col1, $2 as Col2, $3 as Col3")
    ("varchar", pqxx::prepare::treatstring)
    ("varchar", pqxx::prepare::treatstring)
    ("varchar", pqxx::prepare::treatstring);
connection.prepared("mystatement")("a")("b")("c").exec();
```

È possibile osservare come subito dopo la definizione dello statement (con il nome univoco e la query) occorra dichiarare il tipo di ogni parametro (in questo caso tre valori testuali). Non sembra possibile eseguire l'esecuzione della query in un momento diverso dal binding dei parametri: i parametri devono essere legati chiamando una funzione sul valore di ritorno di pqxx::connection::prepared() ed il metodo di esecuzione dev'essere chiamato sul valore di ritorno di questa funzione. Questo modo di procedere è sicuramente comodo in certi casi, ma considerando il vincolo di modificare il meno possibile la struttura software lo 'standard' libpqxx risulta totalmente inutilizzabile. È stato necessario identificare, all'interno del codice, i punti in cui la flessibilità di SQLite veniva sfruttata e modificare queste porzioni di codice per uniformarli a PostgreSQL. Il numero di metodi modificati è esiguo.

Risolto il problema dell'ordine di binding dei parametri, restava da implementare il binding effettivo, che doveva essere in grado di legare un singolo parametro alla volta. È stato necessario manipolare esplicitamente alcune parti di libpqxx progettate per rimanere trasparenti agli occhi dello sviluppatore: le classi pqxx::prepare::declaration e pqxx::prepare::invocation. I limiti presentati da queste classi, utilizzate in un modo che chiaramente gli sviluppatori di libpqxx non avevano previsto, hanno rivelato un certo grado di rigidità nell'interfaccia della libreria. declaration e invocation vengono utilizzate per effettuare rispettivamente la dichiarazione e l'assegnamento dei parametri da legare agli statement e sono nascoste nei valori di ritorno delle chiamate pqxx::connection::prepare() e pqxx::work::prepared(). Per poter effettuare l'assegnamento dei parametri è stato necessario incapsulare

invocation nella classe Pgres_Stmt. Su invocation viene invocato il metodo operator()(), per ogni parametro da assegnare, nell'ordine specificato. Mancando di operatore di assegnamento (privato e non implementato) non era però possibile, una volta terminato l'utilizzo dello statement con determinati parametri, generare una nuova invocation. Questo limite impediva di ripetere le operazioni di bind con un altro set di parametri, rendendo la classe di fatto utilizzabile una volta sola. L'ostacolo è stato risolto sostituendo a invocation un opportuno wrapper per un puntatore alla stessa, il quale viene resettato e inizializzato con una nuova istanza di invocation durante il reset. Questa operazione, come si può evincere dal codice sopra, necessita di un riferimento all'oggetto connessione per poter essere effettuata indipendentemente dal DB_Utils.

declaration è stata invece utilizzata per dichiarare il tipo dei parametri. Non essendo necessario resettarla, è stato sufficiente salvarsi un riferimento ad essa. Per avere informazioni più precise sul contesto d'uso di declaration si fa riferimento alla sottosezione 2.5.1.

Gli accorgimenti sopra presentati hanno molto complicato le procedure di creazione, binding e reset degli statement sul lato PostgreSQL, ma sono stati necessari per garantire una efficace traduzione.

Lati negativi

- L'applicazione risultante ha dovuto perdere flessibilità: perché PostgreSQL potesse legare i parametri nel modo corretto, tutti i casi in cui i parametri non venivano legati nell'ordine in cui comparivano nella query hanno dovuto essere modificati. Nell'applicazione attuale, i parametri vengono legati in ordine sequenziale. La possibilità di legare i parametri esplicitando gli indici esiste ancora (è necessaria e sfruttata sia nel backend SQLite che nel backend MySQL), ma la flessibilità che questa funzionalità offre non può più essere utilizzata fuori dal modulo DB_Utils.

- La struttura descritta sopra, con la manipolazione esplicita di invocation e declaration, può essere usata solo fino alla versione 3.1 della libreria libpqxx: dalla versione 4.0, infatti, la chiamata pqxx::connection::prepare() non ha valore di ritorno e la struttura elaborata perde consistenza.

Probabilmente usare libpq invece di libpqxx avrebbe reso la traduzione meno semplice ma più fedele. La minore flessibilità potrebbe essere dovuta al fatto che, essendo libpq implementato in C come PostgreSQL, la maggior

parte degli sviluppatori usa questa libreria. libpq, avendo un bacino di utenza molto vasto, resta dunque aperta a un largo ventaglio di tipi di utilizzo. Al contrario libpqxx, venendo usata da meno sviluppatori, ha sicuramente ricevuto meno feedback e gli sviluppatori non hanno tenuto in considerazione utilizzi di questo tipo. Si fa comunque notare che anche nella libreria libpq i tipi dei parametri dei prepared statement devono essere dichiarati ed i valori vengono assegnati tutti in una volta al momento dell'esecuzione.

2.4 Modifiche al DB_Utils

La struct DB_Utils originaria conteneva semplicemente una serie di metodi operanti su un database SQLite per finalità di esecuzione di query (con relativa gestione degli errori), fetching dei risultati e reportistica sullo stato del database. Per permettere il supporto a più DBMS anche questa entità è stata trasformata in una classe, seguendo lo stesso approccio adottato per gli statement (figura 2.3). La versione per SQLite conteneva sostanzialmente gli stessi metodi della classe originaria. PostgreSQL oltre a tradurre la sintassi SQL ha dovuto adottare alcuni accorgimenti per poter lavorare con la stessa metodologia della classe originaria.
La classe base virtuale (DB_Utils) che ha preso il posto dell'originale struct

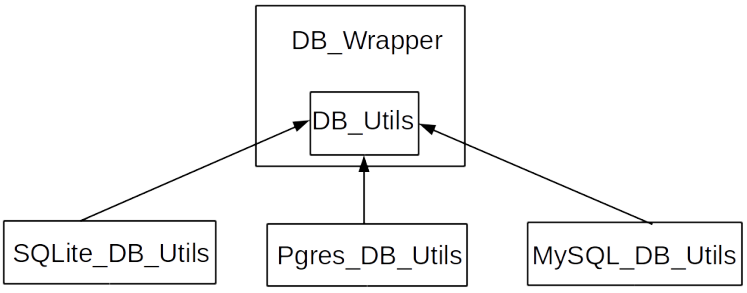

Figura 2.3: Gerarchia di DB_Utils

è stata a sua volta incapsulata in una classe wrapper, che ha il solo scopo di nascondere l'implementazione effettiva ed evitare di dover cambiare l'invocazione dei metodi in tutto il codice (senza wrapper, sarebbe stato necessario invocare i metodi di DB_Utils tramite puntatore). Questa scelta ha permesso di evitare il più possibile di toccare codice in zone sparse, concentrando eventuali debugging in un'unica unità di traduzione, ed ha come side-effect un più rigido controllo sul puntatore alla classe base (necessario per la riso-

luzione corretta dell'overriding) e di conseguenza una migliore gestione della classe stessa.

2.4.1 La creazione di uno statement

Come già accennato precedentemente (si veda la sezione 2.3.1, la creazione di un `R_Stmt` dipende dal `DB_Utils`, in quanto è quest'ultimo a fornire il puntatore a `Stmt` da incapsulare. Di seguito viene fornita una descrizione abbastanza dettagliata del processo di creazione nel backend `PostgreSQL`.

1. Viene creata una query: questa può consistere di una semplice stringa testuale o di una `Reusable_Query`.
2. Viene invocato il metodo che crea lo statement a partire dalla query. Tramite overloading, viene chiamata una delle 4 varianti di questo metodo (le combinazioni di query parametrica o non parametrica, derivante da una semplice stringa o da una `Reusable_Query`). Al suo interno:
   - se si sta creando uno statement a partire da una `Reusable_Query` già incontrata, viene restituito lo `Stmt`* corrispondente nella cache.
   - se la query è parametrica, il testo della query viene processato e la sintassi dei parametri viene corretta: per approfondire vedere la sezione 2.5.1.
   - se la query presenta parametri, viene effettuata la dichiarazione dei parametri tramite la classe `invocation`.
   - viene richiesto al DBMS di preparare un nuovo statement, specificando la query in questione ed il nome con cui questo viene identificato. Il nome dello statement è obbligatorio se si vogliono mantenere più statement utilizzabili nello stesso momento: lo statement senza nome può essere definito, ma viene rimpiazzato ad ogni successiva definizione. Per velocizzare l'indicizzazione dello statement, il nome è stato ridotto ad una codifica alfanumerica crescente.
   - viene creato un oggetto di tipo `Pgres_Stmt` contenente i campi necessari: il nome dello statement, il wrapper di `declaration` necessario per legare i parametri, i riferimenti agli oggetti di `DB_Utils` per permettere di reinizializzare il wrapper in caso di reset.
3. il puntatore viene restituito all'esterno e incapsulato in un `R_Stmt` che ne nasconde il tipo dinamico.

2.5 Limitare la dipendenza dal DBMS

La sintassi delle query cambia a seconda del DBMS su cui queste vengono eseguite: alcune query possono essere accettate su un sistema, ma rifiutate in quanto sintatticamente scorrette su un altro. Queste problematiche interessano in particolar modo le query che usano una sintassi diversa da quella definita nello standard `SQL`. Finché queste query si trovano all'interno del `DB_Utils` (parte dinamica dell'applicazione), la compatibilità viene realizzata in modo piuttosto naturale: è possibile riscrivere a piacimento le query in ogni backend in modo da poterle eseguire sul DBMS di riferimento senza problemi di sintassi. Dato che nell'applicazione originale la sintassi `SQL` non standard veniva usata anche nei moduli non legati alla gestione del database (principalmente il modulo di caricamento e quello di confronto), in teoria anche questi avrebbero dovuto diventare virtuali ed essere implementati per ogni backend. Le differenze specifiche vengono discusse nel capitolo 3, in questo paragrafo si fa piuttosto notare come anche in questo caso il risultato finale sia un compromesso tra pulizia ed efficienza: i metodi contenenti query non standard sono stati trasferiti nell'interfaccia del `DB_Utils` ed ogni backend li ha implementati in modo compatibile per il DBMS corrispondente. In alcuni casi è stato necessario aggiungere parametri alle chiamate, in quanto all'interno dei metodi venivano utilizzate strutture dati contenute dai moduli ed estranee al `DB_Utils`. In totale, circa una decina di metodi sono stati inglobati dal `DB_Utils` ed utilizzati dai moduli di caricamento, sommario e confronto.

2.5.1 Compatibilità dei prepared statement

Un altro problema di portabilità riguardava la sintassi dei prepared statement. Come già accennato, le query parametriche vengono definite in pressoché ogni modulo dell'applicazione. Ogni DBMS implementa i prepared statement senza seguire uno standard, come spesso capita con le entità `SQL`: ne consegue che, a seconda del backend utilizzato, una data sintassi per un prepared statement può essere accettata o rifiutata.

Codifica

Di default `SQLite` utilizza una notazione dei parametri dei prepared statement non numerale, in cui ogni parametro è sostituito da un punto di domanda ('`?`'). La richiesta di bind viene effettuata specificando il numero del parametro da legare al valore richiesto, come illustrato sopra.
In `PostgreSQL` viene invece utilizzata una notazione numerale del tipo '`$x`'.

In questo caso risulta possibile usare lo stesso parametro più volte all'interno della query (esempio: `select $1 where $1 = $2`), ma dato che nell'applicazione si è deciso di non sfruttare tale proprietà (essendo impossibile distinguere i parametri nella notazione `SQLite`) tale decisione è rimasta anche nell'altro backend.

Una sintassi alternativa di `SQLite` prevede che i parametri vengano espressi nella forma '`$x`'. Volendo sarebbe stato possibile riscrivere tutte le query dell'applicazione evitando le problematiche (principalmente di performance) relative alla conversione del testo della query. Tale iniziativa presenta comunque scarsa rilevanza dal punto di vista semantico ed è stata ritenuta opzionale. Inoltre, la sintassi '`$x`' non è generale (la sintassi equivalente per `MySQL` è '`%x`'), dunque al momento di implementare il supporto di un nuovo backend il problema della conversione avrebbe comunque dovuto essere risolto.

Il problema di compatibilità viene risolto internamente al `DB_Utils`. A livello di applicazione si mantiene come 'standard' la sintassi di `SQLite`: ogni backend può cambiare la codifica dei parametri per adattarla alla sintassi del DBMS corrispondente. Senza questo accorgimento, ogni metodo contenente un prepared statement avrebbe dovuto essere spostato all'interno di `DB_Utils`, rendendo l'applicazione quasi completamente dipendente dal backend (violando il primo e l'ultimo dei principi SOLID).

Esempi di sintassi di prepared statement

In `SQLite`:

```
update rule set (summary, sortkey) = (?, ?) where id = ?;
```

In `PostgreSQL`:

```
update rule set (summary, sortkey) = ($1, $2) where id = $3;
```

Dichiarazione dei parametri

Altro aspetto a cui è stato necessario apportare modifiche in questo campo è la dichiarazione dei parametri. `SQLite` non richiede che il tipo dei parametri venga specificato, mentre `PostgreSQL` (e `libpqxx` di conseguenza) è più rigido al riguardo. L'applicazione non effettuava nativamente alcuna dichiarazione dei parametri, dunque è stato necessario aggiungere questa funzionalità e fare in modo che `SQLite` la ignorasse. Una prima soluzione consisteva nell'utilizzo di `etc()`, un metodo di `libpqxx` che permette di dichiarare un numero arbitrario di parametri di tipo arbitrario (sostanzialmente chiedendo a `libpqxx` di cercare di inferire automaticamente tipo e numero di parametri). Questo approccio funziona solo finché parametri di tipo stringa - varchar

Tabella 2.1: Tabella di conversione. le lettere vengono passate come un parametro stringa e la dichiarazione dei tipi viene effettuata durante la creazione dello statement nello stesso ordine in cui le lettere vengono presentate.

Lettera	Tipo di dato corrispondente
v	varchar
i	integer
t	timestamp
b	boolean

erano coinvolti. La soluzione definitiva ha richiesto interventi più delicati. Ai costruttori di `Reusable_Query` e `Stmt` è stato aggiunto un parametro di tipo stringa, contenente la dichiarazione in forma breve dei parametri. Alla creazione dello statement effettivo i parametri vengono dichiarati uno alla volta tramite un semplice switch case (vedi tabella 2.1) utilizzando la classe `declaration` a cui si è accennato sopra. Questo meccanismo permette di migliorare la stabilità del codice in due modi:

- permette di effettuare un controllo incrociato, verificando che il numero di parametri dichiarato nella stringa sia uguale al numero di parametri trovato durante la traduzione della query parametrica;
- permette di riconoscere le query parametriche (che richiedono traduzione) da quelle non parametriche (sostanzialmente standard): ciò permette di ottimizzare leggermente i tempi di esecuzione riducendo il numero di invocazioni della funzione di traduzione ed evita al contempo che si dichiarino query non parametriche in modo parametrico e viceversa.

Segue un esempio di utilizzo della dichiarazione parametri durante la costruzione di una `Reusable_Query`: notare il secondo parametro del costruttore, che specifica il tipo dei tre parametri.

```
static Reusable_Query query("update rule set (summary, sortkey) =
        ($1, $2) where id = $3", "vvv");
```

2.5.2 Limitare la doppia manutenzione

Dove possibile, si è cercato di ridurre il principale problema dovuto alla procedura di trasferimento delle query all'interno del `DB_Utils`: la doppia

manutenzione. Alcune query contenute nel `DB_Utils` si sono rivelate infatti abbastanza standard da poter essere utilizzate da tutti i backend (`MySQL` compreso) senza necessità di modifiche. Queste query (circa una ventina) sono state raggruppate nell'unità di traduzione della classe base ed utilizzate dai backend come costanti. Si è quindi ottenuto un parziale 'single point of failure, single point of control': eventuali manutenzioni possono essere eseguite una volta sola ed in una singola, definita area del codice.

Alcune query, anche se non comuni nei DBMS, condividevano alcune strutture comuni. È il caso, ad esempio, delle query di definizione di indici e viste. `SQLite` supporta, per queste entità, la definizione condizionale (l'indice o vista viene creato solo se non esiste già), mentre `PostgreSQL` no. Per ottenere lo stesso effetto occorre eseguire una chiamata alle tabelle di amministrazione (dove le informazioni su queste entità sono mantenute) ed eseguire la query solo se l'entità che si vuole creare non è ancora stata definita. Per evitare di spostare anche questa procedura nel `DB_Utils`, sono state create due funzioni per la definizione di indici e viste. I parametri di queste funzioni sono le informazioni essenziali per la definizione delle entità; al loro interno le funzioni svolgono i controlli dovuti ed organizzano le informazioni in modo compatibile con il DBMS. Di seguito è possibile osservare una chiamata di funzione per la creazione di indici con le query effettuate da ogni backend.

In un modulo di `report`:

```
db.create_index("cache.",
                "phr_rule",
                "phr(rule)");
```

In `SQLite`:

```
create index if not exists cache.phr_rule
    on phr(rule);
```

In `PostgreSQL`:

```
select 1
from pg_class c, pg_namespace n
where c.relnamespace = n.oid
    and relname = 'phr_rule'
    and nspname = 'cache' and relkind = 'i';
// se la prima query ritorna 0 righe,
// l'indice non esiste.
create index phr_rule on cache.phr(rule);
```

Le `Reusable_Query`, apparentemente legate agli `Stmt`, contengono solo il codice sql della query da riutilizzare ed un indice. `DB_Utils` contiene una cache che memorizza gli statement ottenuti da `Reusable_Query` man mano che questi vengono dichiarati, velocizzandone la fruizione: quando occorre generare uno `Stmt` a partire da una `Reusable_Query` già utilizzata, invece di essere creato da zero lo `Stmt` viene recuperato dalla cache. Dato che le `Reusable_Query` non dipendono dal database, a queste strutture non è stato applicato nessun cambiamento.

2.6 I design pattern utilizzati

Il refactoring ha comportato l'applicazione di una serie di design pattern [8], di cui i principali sono:

- Adapter: In qualche caso, i metodi delle concretizzazioni di `DB_Utils` svolgono semplicemente la funzione di passacarte" tra l'esterno e la libreria su cui si appoggiano. Un esempio lampante è dato dai metodi `step` e `bind`, che nella classe `SQLite_DB_Utils` chiamano le omonime funzioni della libreria.

- Bridge: La classe `DB_Wrapper` implementa un bridge tra l'interfaccia e le diverse concretizzazioni, fungendo da passacarte per i metodi pubblici del `DB_Utils` sottostante. Il backend viene deciso a runtime.

- Factory: Ogni implementazione di `DB_Utils` crea delle sottoclassi di `Stmt` con i quali il `DB_Utils` corrente è in grado di lavorare. Occorre passare per il `DB_Utils` per creare gli `Stmt`: i moduli esterni non conoscono nulla dell'implementazione interna degli `Stmt` e li usano come scatola nera.

Capitolo 3

Differenze tra i DBMS

3.1 Query non standard

La maggior parte delle query definite dallo standard `SQL` sono utilizzabili su ogni database. Alcuni DBMS hanno arricchito l'offerta con altre istruzioni che aumentano la potenza espressiva del linguaggio. Queste istruzioni sono utilizzabili solo su una selezione ridotta di database, quindi durante il porting è stato necessario tradurle in modo da ottenere gli stessi risultati anche sui DBMS che non le supportano.

3.1.1 `insert or ignore`

L'applicazione originaria utilizzava piuttosto pesantemente un tipo di inserimento non standard: `insert or ignore`. In questa variante della canonica `insert` le ennuple che dovrebbero essere inserite ma violerebbero il vincolo di chiave primaria non vengono inserite. Il mancato inserimento evita la generazione di un errore e, di conseguenza, il fallimento dell'intera transazione. Per emularne il comportamento in `PostgreSQL` è stato necessario spostare all'interno dell'interfaccia i metodi che le utilizzavano (per un totale di circa 20 query e 3 metodi) e riscrivere tali query come canoniche insert con la clausola `where not exists`, in modo da verificare che le tuple inserite non violassero il vincolo. Questa modifica è stata efficace, ma ha ridotto la leggibilità delle query interessate.
Esempio di query '`insert or ignore`' in `SQLite`:

```
insert or ignore
into tag(domain, name, sortkey, comment)
values ('mainfile', $1, $2, '-load');
```

Query equivalente in PostgreSQL:

```
insert into tag(domain, name, sortkey, comment)
select 'mainfile', $1, $2, '-load'
where not exists (select 1 from tag
    where name = $3 and domain = 'mainfile');
```

3.1.2 insert or update

Nell'applicazione viene utilizzato un altro inserimento non standard di SQLite, `insert or update`, generalmente chiamato `upsert` nella letteratura della rete. Questa seconda variante prevede che le tuple che violano il vincolo di chiave primaria non vengano inserite, ma sovrascritte a quelle già presenti. In questo caso l'equivalente PostgreSQL non è banale: la soluzione più semplice sarebbe una query di `delete` delle tuple presenti in tabella che violano il vincolo di chiave primaria seguita dalla canonica `insert`, ma questo modo di procedere è incline a violare vincoli di chiave esterna (cancellare tuple contenenti valori riferiti da altre tabelle potrebbe provocare una cancellazione in cascata, operazione non equivalente alla query originale).

La soluzione in questo caso consiste nell'invertire l'ordine: si tenta di eseguire l'`update` della tupla e si controlla se è stata apportata la modifica. Se ciò non accade, la tupla che presenterebbe la stessa chiave primaria non esiste nella tabella e può essere inserita senza problemi. Notare che questa soluzione può presentare comunque problemi in caso di esecuzione parallela: due thread potrebbero tentare la update contemporaneamente notando che la tupla non esiste e nella successiva inserimento uno dei due fallirebbe a causa di chiave primaria duplicata.
Esempio di query `insert or update` in SQLite:

```
insert or update into rule(id, summary, sortkey) values (?, ?, ?);
```

Query eqivalenti in PostgreSQL:

```
update rule set (summary, sortkey) = ($1, $2) where id = $3;
// Se l'update modifica 0 righe, nessuna tupla
// corrisponde a quella specificata.
insert into rule(id, summary, sortkey) values ($1, $2, $3);
```

3.2 Timestamp, timezone e millisecondi

Il timestamp è un tipo di dato complesso e molto importante, che implementa giorno ed ora con precisione arbitraria e supporta una certa varietà di formati

e fusi orari. SQLite non comprende tra i suoi tipi di dato il timestamp, ma fornisce una funzione che formatta un intero (unix time, il numero di secondi passati dalla mezzanotte del 1/1/1970) in una data. Inoltre, SQLite non offre supporto per i fusi orari, vale a dire che il timestamp inserito viene formattato in riferimento al meridiano di Greenwich, senza considerazioni relative all'effettivo luogo di esecuzione. PostgreSQL, al contrario, ha una variabile globale timezone che di default ha valore 'localtime': i timestamp inseriti vengono automaticamente convertiti nel fuso orario locale. Tale variabile può essere modificata: ad esempio, impostando il fuso orario a UTF+00 e chiedendo a PostgreSQL di restituire il timestamp attuale (con la funzione now()) alle ore 12:00 su una macchina situata a Roma, PostgreSQL risponderà che sono le ore 11:00.

Per rendere il backend di PostgreSQL equivalente con quello di SQLite è stato necessario rendere i timestamp generati uguali rispetto ai fusi orari. Modificare i file di configurazione di PostgreSQL per uniformare il fuso orario è sconsigliabile in quanto:

- richiede privilegi da superutente;
- è un'operazione complessa;
- apporta modifiche persistenti;
- richiede un riavvio del database.

La soluzione alternativa consiste nell'impostare il valore di tale variabile a runtime: tale modifica può essere fatta da qualsiasi utente, coinvolge solo la sua connessione ed i tempi inseriti con il fuso orario personalizzato mantengono lo stesso valore anche in altri timezone. La sintassi è semplice: si tratta di modificare una variabile, nello stesso modo con cui si modifica lo schema. Apportare questa modifica appena connessi al database risolve i problemi di allineamento con SQLite.

Altra differenza, dovuta alla carenza di tipi di dato temporali, è la precisione del timestamp. PostgreSQL offre controllo anche sulle frazioni di secondo (fino al microsecondo), mentre SQLite ha una precisione limitata al secondo. Nel confronto dei risultati di esecuzione dei due backend anche queste piccole differenze portano ad avere risultati diversi. Per risolvere, è sufficiente dichiarare i campi timestamp di PostgreSQL come timestamp(0), senza esplicitare il fuso orario. Una funzionalità molto utile di PostgreSQL è la possibilità di decidere il fuso orario anche al momento di generare o inserire un timestamp.

3.3 Codifica dei booleani

Tra i tipi di dato non supportati da `SQLite` compare anche il booleano. La codifica dei booleani nell'applicazione originale viene implementata utilizzando il più piccolo intero disponibile (`char`). Il backend per `PostgreSQL` ha dovuto adattare la codifica utilizzata ed il relativo tipo di dato. Fortunatamente, la codifica da intero a booleano viene interpretata correttamente da `PostgreSQL`, evitando problemi di conversione. Nonostante tra le funzioni di binding quella con parametro booleano sia assente, tra i parametri dichiarati per i prepared statement (tabella 2.1) compare anche il tipo booleano. È questo accorgimento a permettere che la conversione funzioni: durante la definizione dello statement il parametro viene dichiarato booleano e durante il binding il parametro intero viene correttamente convertito.

3.4 Collating

Collating è un termine che indica il criterio adottato dal DBMS per ordinare le stringhe. `PostgreSQL` può adottare diversi collating, basate sugli alfabeti delle varie lingue (`IT_it`, `EN_en`, ...) e sulla codifica adottata (`ASCII`, `UTF-8`, `UTF-16`, ...). Per contro, il collating di default di `SQLite` è piuttosto grezzo: si esegue una semplice `memcmp` (memory compare) sulle due stringhe ed in base al risultato le stringhe vengono ordinate. Per realizzare la completa uniformità tra i due backend è opportuno far sì che i risultati siano ordinati nello stesso modo. In `PostgreSQL` si può emulare il collating di `SQLite` specificando nelle opzioni di creazione del database '`lccollate=C`'. In caso non si voglia cambiare database utilizzato, per ottenere lo stesso ordinamento sarebbe necessario specificare `collate 'C'` nelle query in cui viene richiesto di ordinare secondo stringhe. Tale accorgimento implicherebbe un sostanzioso trasferimento di query (se non di metodi) dai moduli esterni verso l'interfaccia, operazione sconsigliata per i già citati motivi di portabilità ed efficienza. Notare che l'opzione riguarda la creazione del database, non dello schema.

3.5 Identificare il database: file vs schema

`SQLite` opera su database memorizzati su file. Tali file vengono specificati nelle opzioni di caricamento dell'applicazione e vengono gestiti (montati, smontati, creati, ...) da una piccola varietà di metodi ed in contesti distinti del codice. `PostgreSQL` implementa il protocollo client/server e presenta sia database che schema, entrambi gestiti direttamente dal DBMS. Le due visio-

ni riguardo al meccanismo di memorizzazione dei dati risultano fin dall'inizio scarsamente compatibili.

Il divario viene risolto mappando il concetto di "file database" di SQLite nel concetto di "schema" di PostgreSQL. In PostgreSQL lo schema è una porzione di database, più che un database a sé stante. La distinzione tra database e schema è simile a quella tra libreria e namespace in C/C++: una libreria è composta da più namespace, ognuno dei quali identifica un ambiente separato. Nonostante ogni schema abbia le sue tabelle e la sua gerarchia di autorizzazioni, tra essi non ci sono particolari distinzioni: è possibile agire su più schemi contemporaneamente, esattamente come in un'applicazione è possibile invocare due metodi da due namespace differenti. Allo stesso modo, in SQLite si può agire su più database (quando questi vengono 'montati'). Usare gli schema in PostgreSQL permette la separazione di caricamenti diversi mantenendo le funzionalità necessarie per operazioni di confronto. Qualche differenza emerge nella procedura di controllo di esistenza del file, in cui il controllo sul file system viene sostituito da una query sulle tabelle di amministrazione, e nella gestione di montaggio e smontaggio del file (operazione senza significato per PostgreSQL).

3.6 Tablewriter vs :memory:

Una differenza sostanziale dal punto di vista delle prestazioni riguarda il file speciale :memory:. Per un buon numero di operazioni, SQLite usa un database speciale che viene caricato direttamente in RAM per migliorare le performance. sebbene in PostgreSQL sia possibile dichiarare tabelle temporanee (mediante la query create temp table, con la quale è possibile specificare che tali tabelle non devono essere rese persistenti) queste sono comunque memorizzate all'interno dello spazio di memoria del DBMS e quindi l'ottimizzazione ottenuta è marginale rispetto a quella che si ha con :memory:. Per migliorare le performance del backend PostgreSQL è stato valutato necessario attingere ad alcune caratteristiche che libpqxx metteva a disposizione e che ancora non erano state sfruttate. In PostgreSQL è presente una istruzione di inserimento e selezione particolarmente utile durante le *batch load*, ovvero i caricamenti massivi in un database dovuti all'inizializzazione dello stesso. Il comando copy può spostare dati da un file in una tabella, da una tabella in un file o tra due tabelle, ed è ottimizzato per la manipolazione di grandi numeri di dati, a discapito della verifica di vincoli (che viene comunque effettuata, ma solo a caricamento ultimato). La struttura dati che in libpqxx supporta questo tipo di query è pqxx::tablewriter: accetta come input un vector<vector<std::string» e lo carica in una tabella specifica utilizzan-

do internamente `copy`. Una caratteristica molto importante di questa query è la possibilità di ridurre lo scambio di messaggi tra client e server, che in caso di database remoti diventa particolarmente oneroso. Per questo motivo utilizzare il tablewriter fornisce risultati apprezzabili solo se l'inserimento viene fatto una volta ogni tanto: ad ogni modo, nonostante fosse preferibile inserire i dati una volta sola (alla fine del caricamento), per motivi di vincoli di chiave esterna è stato necessario dosarne l'utilizzo alla fine di ogni frame del protobuf. I risultati sono comunque visibili ed apprezzabili (vedere la sezione dedicata alle performance per ulteriori informazioni). Le tabelle su cui si è deciso di utilizzare il tablewriter sono 3: `hreport`, `freport` e `area`. Queste tabelle sono le candidate ideali, in quanto sostengono un traffico rilevante e di conseguenza contano un numero dignitoso di righe. Altre tabelle, pur avendo una dimensione simile, selezionano i dati dal database stesso: in questo caso il tablewriter è inutile, mentre l'operazione di `copy` può essere ancora considerata utile. Per evitare gli errori di chiavi duplicate, l'identificativo `hreport(id)`, comune alle tabelle, viene inserito in una tabella hash e controllato ad ogni inserimento lato applicazione.

Nell'applicazione originale veniva effettuata una query per controllare l'esistenza del codice hash del report da inserire: solo se il codice non era ancora stato inserito si procedeva quindi ad inserire il report insieme alle sue aree. Il numero medio di aree per report nei casi studiati varia da 1.5 a 3.5:* il numero medio di query originale si aggira dunque tra 3.5 e 5.5 (una query di selezione, una query di inserimento in `hreport` e le query di inserimento in `area`). Con il tablewriter, il numero di query per queste due tabelle si riduce in modo consistente: nel primo caso si risparmia circa il 70% delle query, nel secondo caso si arriva all'80%. Le query di inserimento vengono gestite dal tablewriter utilizzando metodi molto più efficienti. Il capitolo 4 offre un'idea più precisa delle grandezze numeriche in gioco.

Lati negativi

Il funzionamento interno dei metodi di `libpqxx` che implementano il tablewriter non è stato indagato, ma è probabile che venga creato un file e su di esso venga eseguita una `copy`. Il file non è gestito direttamente dall'applicazione, dunque è possibile che alcune strategie che in questo caso possono essere applicate non vengano utilizzate dal meccanismo di tablewriter. Ad esempio, potrebbe non essere necessario tradurre i dati in stringhe letterali.

*1,5 per database di piccole dimensioni, 3.5 per database di grandi dimensioni. Vedere il capitolo 4 per ulteriori informazioni.

Capitolo 4

Valutazioni di costo

In questo capitolo vengono considerati i costi dell'operazione di refactoring sia in termini di performance (perdita di velocità nella versione finale dell'applicazione rispetto alla versione iniziale) che in termini di portabilità (tempo di sviluppo speso per rendere il sistema aperto al supporto di ulteriori database). Se il refactoring avesse avuto come obiettivo l'ottimizzazione di ogni backend, il risultato finale avrebbe presentato due applicazioni distinte (ognuna ottimizzata per il DBMS di riferimento e fortemente dipendente dalla propria libreria). Viceversa, se il refactoring fosse stato un semplice esercizio di pulizia il codice risultante sarebbe stato semplicemente riscritto utilizzando una libreria di interfacciamento generica (un esempio è SOCI[7]), con gravissime perdite di performance.

Il refactoring è stato effettuato con l'intenzione di aderire il più possibile ai principi di buona programmazione, ma l'applicazione è realmente utilizzata e, come sarà evidente notare in questo capitolo, la possibilità di sfruttare funzionalità specifiche di ogni backend è fondamentale per ottenere tempi di esecuzione accettabili. Il risultato del refactoring è un compromesso tra portabilità ed efficienza.

4.0.1 Aspettative su `MySQL`

Per misurare l'efficacia del refactoring rispetto alla portabilità, è stato deciso di implementare un terzo backend per il DBMS `MySQL`, seguendo la struttura risultante dalle modifiche presentate nel capitolo 2. Su `MySQL` non sono stati implementati meccanismi di ottimizzazione di nessun tipo: la traduzione è stata pressoché letterale, volta unicamente a rendere il backend operativo (almeno sulle operazioni principali).

4.1 Performance

Di seguito vengono presentati i risultati dei test di esecuzione per le varie versioni dei differenti backend. Il modulo di caricamento è stato testato sull'applicazione originaria e sull'applicazione finale, per tutti e tre i backend e per entrambe le varianti di `PostgreSQL`. Il modulo di sommario è stato testato in ogni sua parte sulla versione originale e per i backend `PostgreSQL` e `SQLite`. Per questo modulo non è necessario testare la versione di `PostgreSQL` che utilizza il tablewriter in quanto questa funzionalità interviene solamente nel modulo di caricamento. I test sono stati eseguiti in ambiente locale: non è stato valutato interessante ripetere i test su un database remoto, in quanto già con i risultati del test in locale emergono differenze notevoli.

4.1.1 Il sistema utilizzato per i test

Tutti i test sono stati eseguiti su un notebook Acer Aspire 5750 con le seguenti caratteristiche:

- processore: Intel i3 dual core, 2.1GHz, 3MB cache;
- memoria RAM: 4GB DDR3 a 1333MHz + 2 GB swap disk;
- sistema operativo: Ubuntu 14.04 LTS.

Sono state utilizzate le seguenti versioni delle librerie:

- `PostgreSQL`: 9.3.5;
- `libpqxx`: 3.1;
- `SQLite`: 3.8.2;
- `MySQL`: 5.5.38;
- `mysqlpp`: 1.1.3.

I campioni

I risultati sono stati rilevati su un esempio di database 'medio-piccolo' (della dimensione di circa 10 MB) ed un esempio di database 'grande' (di circa 1.5 GB).
La tabella seguente mostra alcune dimensioni approssimate, per rendere l'idea della complessità dei database di riferimento.

Tabella 4.1: Numero di entrate per le tabelle principali del database, nel database 'medio-piccolo' e nel database 'grande'.

Tabella	Database piccolo	Database grande
`frame`	70	40
`hfile`	200	200
`area`	4300	3500000
`hreport`	2500	990000

4.1.2 Tabelle di confronto

Di seguito vengono presentate le tabelle di confronto dei dati; si procede successivamente a fare qualche considerazione sui risultati presentati.

Tabella 4.2: Tempi di caricamento del database a confronto. S = `SQLite`, P = `PostgreSQL`, P+T = `PostgreSQL` con l'utilizzo del tablewriter, M = `MySQL`.

	Applicazione originale	Applicazione finale			
	S	S	P	P + T	M
Database piccolo	1.5 s	5 s	24 s	14 s	77 m
Database grande	12 m	11 m	36 m	23 m	> 180 m

4.1.3 Confronto tra backend: `load`

Come si può notare, nonostante la cura nell'evitare di appesantire il backend `SQLite` si è verificato un lieve rallentamento della fase di caricamento. Questo rallentamento è visibile solo su database di piccole dimensioni e non sembra influire sul tempo di caricamento di protobuf più pesanti: per questo motivo è ragionevole supporre che la differenza visibile sui database di piccole dimensioni sia pressoché costante e non scali con la mole di dati da caricare.

In alcuni test preliminari, effettuati su una versione iniziale del backend `SQLite`, sono stati misurati i tempi di caricamento del database di piccole dimensioni per una media di circa 9 secondi. Confrontato con quello impiegato dall'applicazione originale, questo tempo medio era allarmante: la velocità di caricamento risulta 5 volte minore, un risultato inaccettabile. Un'analisi approfondita ha rivelato che il problema consisteva in un commit aggiunto in

Tabella 4.3: Opzioni di `summarize` a confronto su database di piccole-medie dimensioni

Opzione	Applicazione originale	Applicazione finale	
	S	S	P
`overall`	0.2 s	0.3 s	2.3 s
`dirs`	0.3 s	0.3 s	2.3 s
`rules`	0.2 s	0.4 s	2.3 s
`files`	0.2 s	0.5 s	2.2 s
`frames`	0.2 s	0.4 s	2.2 s
`rules-with-dirs`	0.2 s	0.5 s	2.3 s
`rules-with-files`	0.2 s	0.5 s	2.3 s
`frames-with-files`	0.3 s	0.5 s	2.3 s

una funzione di inserimento invocata ad ogni frame: nonostante non ci fosse nessuna modifica da salvare sul database, l'esecuzione del commit impiegava comunque un tempo rilevabile per eseguire (circa 0,05 secondi). Considerando che il protobuf di partenza consisteva di circa 70 frames, il carico aggiunto totale era di circa 3,5 secondi, più di un terzo del totale. Questa svista ha evidenziato un limite importante di `SQLite`: i commit, siano essi preceduti da operazioni rilevanti o meno, costano.

Nonostante il limite evidenziato sopra, l'applicazione basata su `SQLite` resta performante: `PostgreSQL` si dimostra da 3 a 5 volte più lento di `SQLite` in fase di caricamento. I tempi di `PostgreSQL` vengono notevolmente ridotti (si notano risparmi intorno al 35% - 40%) utilizzando il tablewriter.

È opportuno spendere qualche breve considerazione riguardo il backend `MySQL`. Il tempo di caricamento risulta spropositato rispetto agli altri backend, ma questo non è un limite di `MySQL`: come specificato ad inizio capitolo, questo backend è stato sviluppato con l'unico scopo di ottenere il supporto di base dell'applicazione verso un altro database, quindi su di esso non è stato svolto nessuno studio di efficienza. La traduzione dei metodi e della libreria è stata pressoché letterale, mirata ad un prodotto funzionante in breve tempo. Le caratteristiche del database, così come le sue configurazioni e le sue possibilità di ottimizzazione sono state ignorate: il risultato è un backend funzionante, ma inutilizzabile all'atto pratico.

Questo esempio è particolarmente significativo per mostrare quanto sia necessario avere la possibilità di sfruttare al meglio le caratteristiche del database. Una libreria di interfacciamento generica avrebbe limitato fortemente le ottimizzazioni consentite, portando ad un'applicazione con performance

Tabella 4.4: Opzioni di `summarize` a confronto su database di grandi dimensioni

Opzione	Applicazione originale	Applicazione finale	
	S	S	P
`overall`	16 s	23 s	185 s
`dirs`	17 s	18 s	183 s
`rules`	16 s	18 s	133 s
`files`	11 s	11 s	109 s
`frames`	11 s	11 s	83 s
`rules-with-dirs`	27 s	30 s	117 s
`rules-with-files`	29 s	31 s	318 s
`frames-with-files`	11 s	11 s	104 s

pessime su ogni database. Come ulteriore esempio si porta la prima versione funzionante del backend `PostgreSQL`, la quale impiegava circa un'ora per caricare il database di grosse dimensioni.

4.1.4 Confronto tra backend: `summarize`

La seconda parte di test riguarda il comportamento del sistema in sola lettura: il comando di `summarize` raccoglie statistiche sul database e le restituisce all'utente, effettuando principalmente query di selezione e raggruppamento. In questa operazione si nota che l'ordine di grandezza della differenza tra `PostgreSQL` e `SQLite` cresce rispetto al caricamento: in media `PostgreSQL` è 7 - 8 volte più lento.

Il motivo della lentezza risiede probabilmente in una fase 'nascosta' dell'operazione di sommario: il caricamento delle tabelle di supporto. Prima di procedere alle operazioni di raggruppamento e statistiche, infatti, vengono create e riempite alcune tabelle nel database di cache. Come evidenziato nel capitolo 3.6, nel backend `SQLite` questo database viene caricato direttamente nella memoria dell'applicazione e permette uno scambio di informazioni molto più rapido del normale. Non è possibile sfruttare il tablewriter per compensare questo svantaggio: se utilizzato i dati viaggerebbero dal processo del server al processo del client e viceversa, introducendo un overhead inutile e potenzialmente elevato. La scelta migliore consiste nell'effettuare le query senza ottimizzazioni.

4.2 Portabilità

Sono disponibili (e necessari) molti parametri che permettono di quantificare l'impegno che ha richiesto quest'attività. Considerando che un obiettivo secondario consisteva nel rendere più agevole lo sviluppo del supporto di altri backend per l'applicazione, è rilevante analizzare questi parametri per poter quantificare il livello di soddisfazione di tale requisito.

4.2.1 Statistiche sul repository

Il lavoro svolto è durato circa 5 mesi/persona, distribuiti in un periodo di 8 mesi. Durante il lavoro sono stati creati in totale 13 nuovi file sorgenti. Per coordinare il lavoro di refactoring tra il candidato ed il team di sviluppo dell'applicazione è stato utilizzato un repository `git` [17]. Sul branch `master` è manutenuta la versione finale dell'applicazione con i tre backend, mentre sul branch `pgres-tablewriter` è manutenuta l'applicazione con il backend `PostgreSQL` che utilizza il tablewriter. Sul branch master il numero di commit effettuati sul repository (compreso quello iniziale) è di 150, per un totale di circa 15000 righe inserite e circa 14000 righe eliminate. Di queste modifiche, il 93% circa è stato svolto dal candidato (in 120 commit) ed il restante 7% congiuntamente con gli sviluppatori dell'applicazione.

4.2.2 Ripartizione del lavoro svolto

Il primo mese/persona è stato speso sulla familiarizzazione con il funzionamento dell'applicazione e sull'analisi e progettazione delle modifiche da apportare al codice. Durante questo periodo sono emerse le principali problematiche discusse nel capitolo 3 (relative alla sintassi non uniforme tra i due backend) e sono state progettate le soluzioni.

Durante il secondo e terzo mese/persona è stato effettuato il grosso del refactoring: sono state create le nuove classi e sono stati spostati i metodi rilevanti all'interno dell'interfaccia. Durante questo periodo sono emerse le difficoltà sulla manipolazione dei prepared statement e sull'utilizzo del database `:memory:`.

Il quarto mese/persona è stato dedicato al raffinamento della struttura ottenuta. Sono state eliminate le differenze sui timestamp e sull'ordinamento descritte nel capitolo 3.

La prima parte del quinto mese/persona è stata dedicata all'implementazione dell'ottimizzazione basata sul tablewriter. Durante questa operazione sono stati effettuati alcuni test per verificare che le modifiche effettuate migliorassero le prestazioni del backend `PostgreSQL`.

La seconda parte del quinto mese/persona è stato dedicato all'analisi, progettazione e codifica del backend di supporto di `MySQL`.

4.2.3 Efficacia dell'astrazione

Per la codifica della parte `MySQL` sono stati necessari circa 12 giorni, per un totale netto di ore/persona circa pari a 40. Di queste, è possibile stimare che 20 siano state passate a lavorare sulla traduzione delle parti di codice che si interfacciano alla libreria software e le restanti 20 a tradurre le query `SQL` caratterizzate da una sintassi non completamente compatibile con il nuovo sistema.

Al momento di iniziare le operazioni di porting verso il DBMS `MySQL`, questo sistema era totalmente estraneo al candidato, così come la libreria utilizzata per il suo interfacciamento con programmi `C/C++`. Può essere approssimato in circa 15 ore/persona il tempo speso per familiarizzare con i nuovi sistemi e progettare le soluzioni alle problematiche discusse nei capitoli 2 e 3. Considerando il breve tempo necessario a rendere il backend funzionale, si può concludere che la struttura di classi descritta nel capitolo 2 sia efficace e sufficientemente stabile rispetto alla portabilità: in altre parole, rispetta in misura accettabile i principi SOLID.

Conclusioni

Il refactoring dell'applicazione è andato a buon fine: l'applicazione finale presenta un'interfaccia abbastanza pulita ed adattabile ad una buona varietà di sistemi diversi. Gli ostacoli principali che possono emergere durante l'implementazione di un nuovo backend sono indipendenti dalla struttura adottata. Al contrario, le interfacce della struttura finale possono intuitivamente guidare lo sviluppatore durante la codifica, senza bisogno di ulteriori analisi del problema.

I risultati sperimentali confermano l'ipotesi iniziale, ovvero la necessità di interfacciarsi ad ogni singolo DBMS utilizzando la sua specifica libreria, così da poter sfruttare le peculiarità dei vari sistemi. Una libreria di interfacciamento generica avrebbe portato a performance inaccettabili per l'applicazione considerata.

Non è stato ritenuto necessario né opportuno spendere tempo per implementare una versione multithread dell'applicazione, principalmente per motivi di coerenza con l'applicazione originaria. Questo non esclude la possibilità di implementarla, almeno per `PostgreSQL`, con probabili vantaggi dal punto di vista delle performance.

La parallelizzazione dovrebbe essere sfruttata principalmente durante la fase di caricamento. In questo caso torna naturale sfruttare la divisione in frames del protobuf che viene analizzato, facendo analizzare e caricare un numero arbitrario di frames contemporaneamente. Combinare il parallelismo e l'utilizzo dei tablewriter porterebbe ad un ulteriore incremento della velocità, ma occorre implementare ulteriori meccanismi di controllo per evitare la violazione di vincoli. Bisogna inoltre tener conto che per gestire correttamente le *race condition* è necessario imporre meccanismi di serializzazione sulle risorse condivise (ad esempio, le tabelle hash) introducendo limitazioni aggiuntive alla parallelizzazione. Va infine tenuto a mente che qualsiasi modifica effettuata all'applicazione non deve incidere sulle performance del backend `SQLite`.

Glossario

Segue una lista dei principali termini utilizzati durante l'elaborato con una breve descrizione:

- Database: insieme organizzato di dati, contenente metadati e vincoli che ne arricchiscono il significato.
- DBMS: acronimo di DataBase Management System, è il programma che si occupa di gestire il database.
- Query: istruzione in `SQL` che definisce o modifica i dati presenti in un database.
- Prepared Statement: particolare tipo di query che viene analizzata una volta sola e può essere eseguita più volte usando parametri diversi.
- Backend: modulo che realizza il supporto al database.
- Vincolo: regola che dev'essere rispettata dai dati: non tutti i DBMS impongono vincoli con la dovuta severità.
- Protobuf: abbreviativo per Protocol Buffers, è un meccanismo di memorizzazione dati.

Ringraziamenti

Il primo e più grande ringraziamento va ad Enea Zaffanella, relatore e maestro dentro e fuori dall'aula. La sua competenza ed infinita pazienza mi hanno permesso di crescere molto più di quanto sia possibile sui libri.
Il secondo, ma non meno importante, va agli altri professori che mi hanno formato in questi anni, con professionalità ed energia: Federico Bergenti, Gianfranco Rossi, Alessandro Dal Palù, Grazia Lotti, Roberto Bagnara, Roberto Alfieri, Alessandra Aimi, Giorgio Picchi e Francesco Di Renzo. Un bel corso è tenuto da un bravo professore ed in questi anni ho avuto la fortuna di incontrare quasi sempre corsi interessanti. In questo ringraziamento vorrei includere anche Vanni Gorni, mio professore di informatica del liceo, la cui impostazione è stata fondamentale durante questi anni.
Ringrazio i miei genitori Angela e Gianluca e la mia fidanzata, Marina, per avermi sopportato e supportato: poter condividere i momenti bui con qualcuno è stato fondamentale per resistere fino alla fine.
Un forte abbraccio ai miei colleghi di studio, per aver condiviso il sudore del percorso. Tra questi un saluto speciale a Maxim Gaina, Sebastian Davrieux, Alessio Bortolotti, Paolo Grossi e Francesco Trombi: senza di voi ogni passo sarebbe stato più pesante.
Infine, un ringraziamento ad ER-GO ed all'Università degli Studi di Parma, con tanto di segreterie e strutture: senza l'Università non avrei iniziato questo percorso e senza borsa di studio non sarei riuscito a completarlo.

Siamo come nani sulle spalle di giganti,
così che possiamo vedere più cose di loro e più lontane,
non certo per l'acume della vista o l'altezza del nostro corpo,
ma perché siamo sollevati e portati in alto dalla statura dei giganti.

- Bernard de Chartres

Bibliografia

[1] Wikipedia Community. *Diamond Problem*. URL: http://en.wikipedia.org/wiki/Multiple_inheritance.

[2] Wikipedia Community. *RAII/RRID idiom*. URL: http://en.wikipedia.org/wiki/Resource_Acquisition_Is_Initialization.

[3] Wikipedia Community. *SOLID principles*. URL: http://en.wikipedia.org/wiki/SOLID_(object-oriented_design).

[4] Oracle Corporation. *MySQL*. URL: http://www.mysql.it/products/enterprise/.

[5] Oracle Corporation. *MySQL C++ Connector*. URL: http://tangentsoft.net/mysql++/doc/html/refman/index.html.

[6] Boost Developers. *Boost Library*. URL: http://www.boost.org/.

[7] SOCI Developers. *SOCI library*. URL: http://soci.sourceforge.net/.

[8] Erich Gamma et al. *Design Patterns: Elementi per il Riuso di Software ad Oggetti*. Pearson Education Italia, 2002.

[9] PostgreSQL Global Development Group. *PostgreSQL*. URL: http://www.postgresql.org/about/.

[10] PostgreSQL Global Development Group. *PostgreSQL C Connector*. URL: http://www.postgresql.org/docs/9.3/interactive/libpq.html.

[11] SQLite Development Group. *SQLite*. URL: http://www.sqlite.org/about.html.

[12] Google Inc. *Google Protocol Buffers*. URL: https://developers.google.com/protocol-buffers/docs/overview?hl=it.

[13] Lockeed Martin. *JSF Coding Standard*. URL: http://www.stroustrup.com/JSF-AV-rules.pdf.

[14] PUC-Rio. *Lua*. URL: http://www.lua.org/about.html.

[15] BUGSENG s.r.l. *ECLAIR*. URL: http://bugseng.com/it/prodotti/eclair.

[16] MISRA C Team. *MISRA C*. URL: http://www.misra-c.com/Activities/MISRAC/tabid/160/Default.aspx.

[17] Linus Torvalds. *Git*. URL: http://git-scm.com/.

[18] Jeroen T. Vermeulen. *PostgreSQL C++ Connector*. URL: http://pqxx.org/development/libpqxx/.