A Database Experiment With LuajitLaTeX

Roberto Giacomelli

Sommario

Oggigiorno i database sono tra le risorse più importanti in ogni moderna organizzazione. Molti sforzi vengono fatti affinché i dati siano sicuri, affidabili, strutturati tramite relazioni e immediatamente disponibili.

In ogni lavoro di produzione, i documenti contengono ogni sorta di dati che, nonostante tutto, vengono digitati a mano più e più volte, o copiati e incollati, oppure processati da strumenti esterni.

Nel mondo TeX non esistono database con cui lavorare — TeX non è un *reporting tool* — tuttavia le cose sono cambiate grazie all'arrivo della nuova coppia LuaTeX e LuajitTeX.

Questo lavoro è una guida per l'utente a questo nuovo e potenziato mondo TeX spinto dal linguaggio Lua.

Abstract

Nowdays databases are among the most important resources in any modern organization. A lot of effort is done to keep data safe, reliable, structured by means of relationships and instantly available.

Then in every production work, the related documents have plenty of different kind of data, but they are likely to be keyed again and again by hand or copy/pasted or processed by external tools.

In the TeX world there aren't databases to work with—TeX is not a reporting tool—but things have been changed thanks to the new siblings LuaTeX and LuajitTeX.

This paper is a user's guide to that new enhanced TeX world powered by Lua.

1 Motivazione

Al GUIT meeting 2015 di Trento presentai assieme a Gianluca Pignalberi un articolo (Giacomelli e Pignalberi, 2015) che trattava del problema di generare report con il sistema TeX estraendo i dati — per esempio da database — e costruendone un file sorgente pronto da compilare.

Nelle conclusioni di quel lavoro si accennava alla possibile ulteriore esplorazione di soluzioni basate sul nuovo motore di composizione LuaTeX in grado di connettersi *direttamente* a sorgenti dati relazionali attraverso apposite librerie in Lua. Ebbene, con questo lavoro raccolgo quel testimone.

2 Un problema, più soluzioni

Lo scenario che immagino per questa discussione è quello aziendale o professionale nel quale un gran numero di attività coinvolgono la produzione di documentazione contenente *dati*, per la quale l'utente candida il sistema TeX al ruolo di strumento di reporting di elevata qualità in grado di facilitare la creazione di contenuti in un quadro in continua evoluzione.

2.1 Cambiamenti ed evoluzione dati

Nel contesto aziendale o professionale molto spesso i dati vengono memorizzati in *database relazionali*. Questa tecnologia è tra le più affidabili tra quelle oggi disponibili per la gestione dell'informazione digitale. Essa si basa sui vincoli di relazione tra insiemi omogenei detti *tabelle*. Banche, istituti di ricerca, industrie, amministrazioni pubbliche gestiscono ogni giorno grandi quantità di dati in modo affidabile e sicuro evitando gli enormi problemi organizzativi attribuibili allo spezzettamento dei *repository* informativi tra diversi settori dell'organizzazione.

Spesso i database sono centralizzati in server Internet e perciò raggiungibili con i più vari dispositivi e praticamente ovunque. Ciò ha reso addirittura possibile un intero nuovo panorama tecnologico conosciuto sotto il nome di *cloud*.

2.2 TeX e le tecnologie cloud

Il sistema TeX può essere utilmente impiegato per realizzare documenti di elevata qualità tipografica che incorporano dati provenienti da database. Lo si può fare con due modalità automatiche diverse anche se dopo tutto concettualmente simili:

1. estraendo i dati dal database tramite un programma appositamente impostato dall'utente che genera file sorgenti compilabili dal sistema TeX in documenti PDF;
2. compilando un opportuno file sorgente con LuaTeX o LuajitTeX contenente le istruzioni per estrarre i dati dal database e inviarli alla composizione tipografica per la produzione del documento PDF.

Il metodo più avanzato per implementare la soluzione 1 è usare una libreria di *templating* che adopera file di testo con il ruolo di *modelli* di documento. I template contengono parti testuali fisse e campi variabili e l'articolo citato mostra esempi pratici in Lua e Python per *unire* i dati estratti dal database con i template e spiega come sia

possibile strutturare questi modelli tramite l'inclusione di template in altri template o come derivare un template da un altro tramite l'ereditarietà per produrre documenti di complessità arbitraria.

Nella soluzione 2 il template è sostituito dal sorgente TEX. In qualche modo si dovrà astrarre l'informazione dei campi variabili attraverso nomi come nel templating, e si dovrà preparare un insieme di macro utente per l'interrogazione del database e la formattazione dei dati.

Le due implementazioni sono a ben vedere concettualmente simili — e possono essere addirittura mescolate insieme, per esempio un template genera un sorgente che se compilato compone il documento interrogando autonomamente il database — tuttavia nella pratica esse risultano essere assai diverse in particolare per le potenzialità di sviluppo tipografico dei report a opera dell'utente.

L'obiettivo di questo lavoro è esplorare il modo diretto di accesso ai database e valutare le differenze tra le due tecniche di reporting con riferimento ai seguenti aspetti:

- tecnologico,
- qualità e complessità dei report,
- esperienza d'uso dell'utente.

Nell'articolo verranno fornite le istruzioni per configurare il sistema all'esecuzione degli esempi di codice, perché sia possibile toccare con mano le procedure, comprendere meglio il codice e acquisire le basi per implementare i propri progetti.

3 Requisiti di sistema e librerie

Innanzi tutto occorre disporre di una distribuzione TEX recente — diciamo una TeX Live 2016 o successiva — così da poter compilare i sorgenti con i motori di composizione che incorporano Lua, e perciò in grado di eseguire connessioni a database e *query* scritte nel linguaggio SQL.

Per approfondire il linguaggio SQL e la progettazione dei database sono disponibili tantissimi libri. Un testo di argomento generale e completo per gli scopi che ci si prefigge è "Basi di dati" (Atzeni *et al.*, 2014) di carattere universitario. Invece, un elenco di libri di argomento specifico riguardanti il database manager SQLite è riportato nel sito ufficiale del progetto alla pagina `https://www.sqlite.org/books.html`.

Per imparare la programmazione in Lua può essere studiato il libro dell'autore principale del linguaggio stesso, Roberto Ierusalimschy (Ierusalimschy, 2013), tra l'altro esso stesso composto con LATEX.

Nella sezione seguente giustificherò la scelta del database manager SQLite e descriverò in dettaglio come predisporre il proprio sistema desktop installando le librerie condivise e i pacchetti all'interno della distribuzione TeX Live.

4 Lavorare con SQLite3

SQLite è una libreria scritta in C con licenza di pubblico dominio, utilizzata da numerosissime applicazioni, compreso il browser Firefox e le app di Android, che opera su un unico file per ciascun database. SQLite non è quindi disponibile come servizio di rete perché progettato per un uso desktop su singola postazione.

SQLite è la scelta perfetta per applicazioni locali. Non è richiesta alcuna configurazione, l'affidabilità è provata, e sono disponibili *driver* per la connessione praticamente per tutti i linguaggi di programmazione.

La struttura relazionale composta dall'insieme delle tabelle si può eventualmente migrare facilmente verso altri database di rete — per esempio PostgreSQL — una volta verificate le differenze con il dialetto SQL del server di destinazione.

4.1 Necessità e applicazioni

Immaginiamo per esempio le procedure necessarie per l'emissione delle fatture in uno studio professionale. Lo strumento più usato per produrre questo tipo di documenti è senza dubbio il foglio di calcolo, soluzione semplice da implementare ma che non è scalabile, ovvero non riesce a funzionare bene al crescere delle necessità.

Un applicativo basato su database invece si adatta nel tempo alle mutate esigenze basandosi sulla struttura relazionale dei dati del problema in costante evoluzione.

Non è quindi solo una questione quantitativa in relazione alla mole dei dati da gestire, ma soprattutto è importante che le informazioni siano coerenti nei vincoli di relazione e che i dati siano resi disponibili efficacemente.

4.2 SQLite, un database dinamico

La caratteristica principale di SQLite di cui tener conto è questa: in registrazione non esiste il controllo di corrispondenza tra il tipo di dato e quello dichiarato per la tabella, proprio come accade per i linguaggi dinamici[1]. Non si tratta quindi di uno svantaggio o di una limitazione ma di una scelta di progetto coerente con l'ambiente di sviluppo.

Tuttavia è consigliabile, definendo gli attributi, rispettare i tipi dei dati in modo da obbligarci a sviluppare una struttura relazionale coerente all'informazione del mondo reale, garantendoci al tempo stesso una più semplice e sicura possibilità di migrazione su database server a controllo statico del tipo.

1. I linguaggi di scripting come Lua e Python incorporano il concetto di tipo di dato ma solamente in fase di esecuzione, mentre quelli a tipizzazione statica come C o Rust, eseguono il controllo dei tipi in fase di compilazione. Per questi ultimi linguaggi non è possibile assegnare a un oggetto un tipo che sarà noto solo a run time e perciò sono definiti linguaggi statici.

4.3 Installazione di SQLite3

Su sistemi Debian e derivati, compreso Ubuntu, l'installazione corrisponde al comando:

```
$ sudo apt-get install sqlite3
```

sono inoltre necessari i file di libreria a caricamento dinamico — detti *shared libraries* — presenti nel pacchetto di sviluppo, perciò diamo anche il comando:

```
$ sudo apt-get install libsqlite3-dev
```

Per le altre distribuzioni Linux non è difficile scoprire i comandi equivalenti a questi per il package manager relativo.

Su Windows, l'installazione consiste sostanzialmente nella copia di due file da scaricare dalla pagina di download ufficiale del progetto, uno è la libreria a caricamento dinamico in grado di eseguire operazioni sul database, di estensione `dll`, acronimo di dynamic linked library, e l'altro è l'eseguibile `sqlite3`, un command-line tool che interpreta l'SQL in un ambiente interattivo di tipo REPL[2].

Più precisamente, i file si scaricano dai link della sezione *Precompiled Binaries for Windows* della pagina web ufficiale `https://sqlite.org/download.html`. Lì trovate i file compressi degli eseguibili a corredo e la libreria dinamica per i sistemi a 32 o a 64 bit — scaricate quello corrispondente alla vostra versione di Windows.

Dopo aver decompresso i file, sistemateli in una directory — per esempio `C:\sqlite` — e aggiungetela alla variabile di sistema `PATH` in modo che gli eseguibili e la libreria siano *visibili*.

Su sistemi OS X scaricate il file dalla stessa pagina ma dal link della sezione *Precompiled Binaries for Mac OS X (x86)*.

Per controllare che tutto funzioni, è sufficiente lanciare il programma CLI[3] `sqlite3` in una finestra di console. Al prompt proposto dal comando in esecuzione, digitate `.help` per consultare il testo di aiuto ed `.exit` per uscire.

Naturalmente esistono anche dei programmi grafici per interagire con i database SQLite attraverso il mouse. Consiglio l'ottimo SQLite Manager — un add-on per Firefox — ma ce ne sono tanti altri come *DB Browser for SQLite* per OS X.

4.4 I fiumi italiani

Produco qui un esempio semplice ma non banale allo scopo di mostrare al lettore in cosa consiste lo sviluppo di un database personale, ben conscio del fatto che si tratta di un'attività che richiede studio ed esperienza, comunque non impossibili da raggiungere per gli utenti compreso per quelli del sistema TEX.

2. REPL è l'acronimo di read–eval–print loop.
3. CLI è l'acronimo di Command Line Interface.

Consideriamo alcuni dati sui fiumi italiani, iniziando con un database di un'unica tabella contenente i campi `nome` e `lunghezza` (misurata in km).

L'istruzione SQL che la crea è la seguente:

```
CREATE TABLE fiumi (
    nome      VARCHAR(64) NOT NULL,
    lunghezza INTEGER
);
```

Nel codice SQL ho imposto che il campo `nome` non possa essere nullo perché se mancasse il nome del fiume mancherebbe la principale informazione di riconoscimento e i campi restanti non avrebbero senso. In altre parole, ho espresso un vincolo di esistenza imprescindibile per la relazione `fiumi`.

Vedremo poi come inserire ulteriori relazioni più complesse come per esempio per ricavare l'elenco delle regioni italiane attraversate da ciascun fiume.

Tra le molte diverse modalità di esecuzione svolgerò l'esempio pratico dal prompt di `sqlite3` perché preferibile ai fini didattici grazie al modo interattivo di esecuzione delle query.

Mano a mano che diverrete più esperti preferirete forse il modo grafico oppure quello di inserire il codice SQL in un file di testo per farlo poi interpretare dall'utility `sqlite3` stessa, tutto in una volta.

Per creare il nuovo database, all'interno di una finestra di terminale, dopo aver impostato la directory di lavoro in cui verrà salvato il file, si lancia il comando:

```
$ sqlite3 fiumi.db
```

Se il file `fiumi.db` non esiste, esso verrà creato e aperto come database attivo. Se esiste verrà solamente aperto stabilendone la connessione.

All'interno della sessione interattiva a linea di comando, qualsiasi istruzione SQL avrà effetto sul database corrente. Per accertarci che sia effettivamente così, digitiamo il comando puntato `.databases`:

```
sqlite> .databases
seq name  file
--- ----- ----------------------------------
0   main  /home/roberto/testsqlite/fiumi.db
```

Digitiamo ora direttamente il codice SQL per creare la nostra tabella:

```
sqlite> CREATE TABLE fiumi (
   ...> nome VARCHAR(64) NOT NULL,
   ...> lunghezza INTEGER
   ...> );
sqlite>
```

Ricordiamoci che nella sintassi SQL è obbligatorio separare le definizioni dei campi con una virgola e inserire il punto e virgola finale. Come avrete notato l'utility interattiva non torna al prompt finché l'istruzione digitata su più linee non è terminata.

Passiamo ora a popolare la tabella appena creata con i dati dei primi dieci fiumi più importanti. L'istruzione è la seguente, con ciascuna tupla separata da una virgola e il solito punto e virgola finale di chiusura:

```
INSERT INTO fiumi VALUES
('Oglio',  280),
('Adige',  410),
('Tanaro', 276),
('Reno',   210),
('Tevere', 405),
('Po',     652),
('Piave',  220),
('Adda',   313),
('Ticino', 248),
('Arno',   241);
```

Sempre in modalità interattiva, eseguiamo ora la query dei dati appena inseriti ordinando le tuple in senso decrescente secondo la lunghezza dei fiumi, e richiedendo la stampa nel modo a colonne con il comando puntato `.mode` e l'opzione `column`):

```
sqlite> .mode column
sqlite> SELECT * FROM fiumi
   ...> ORDER BY lunghezza DESC;
Po          652
Adige       410
Tevere      405
Adda        313
Oglio       280
Tanaro      276
Ticino      248
Arno        241
Piave       220
Reno        210
```

Come esercizio si provi a inserire un fiume con nome pari a `NULL` per verificare se effettivamente viene sollevato un errore, e a scrivere la query per calcolare la somma delle lunghezze dei fiumi usando la funzione `sum()` sul campo `lunghezza`.

4.5 Quali sono le regioni attraversate?

L'idea è quella di creare una nuova tabella che rappresenti le regioni e una ulteriore tabella che rappresenti le relazioni tra fiumi e regioni. Per rendere le relazioni tra elementi univoche si utilizza generalmente una *chiave primaria*, cioè un campo o un insieme di campi in una tabella, i cui valori non si ripetono e non sono mai nulli.

La tabella `fiumi` si modifica in questo modo:

```
CREATE TABLE fiumi (
    id INTEGER PRIMARY KEY,
    nome VARCHAR(64) NOT NULL,
    lunghezza INTEGER,
    CHECK (lunghezza > 0)
);
```

La nuova tabella `regioni` sarà presumibilmente la seguente:

```
CREATE TABLE regioni (
    id INTEGER PRIMARY KEY,
    nome VARCHAR(64) NOT NULL
);
```

e la tabella di relazione `passa_da` potrebbe essere:

```
CREATE TABLE passa_da (
    id_fiume INTEGER NOT NULL,
    id_regione INTEGER NOT NULL,

    PRIMARY KEY (id_fiume, id_regione),

    FOREIGN KEY (id_fiume)
    REFERENCES fiumi (id),

    FOREIGN KEY (id_regione)
    REFERENCES regioni (id)
);
```

sulla quale è presente il vincolo di chiave primaria a doppio campo che esprime la condizione biunivoca che un fiume non attraversi la stessa regione più di una volta e, viceversa, che una regione non sia attraversara dallo stesso fiume più di una volta.

Completiamo la definizione richiedendo che effettivamente i valori interi contenuti nei record di questa tabella siano effettivamente identificatori presenti nelle tabelle `fiumi` e `regioni`, e questo è espresso nelle ultime 4 righe di codice SQL.

Per ragioni di compatibilità con versioni precedenti, in SQLite occorre attivare esplicitamente il controllo per i vincoli di chiave esterna attraverso l'esecuzione del comando seguente:

```
sqlite> PRAGMA foreign_keys = ON;
```

questa operazione va *sempre* fatta quando si vuole inserire dati nel database.

Inserendo i dati, saremo in grado di costruire la tabella 1 riportata fuori testo nella prossima pagina, eseguendo la query con il doppio join:

```
SELECT
fiumi.id        AS num,
fiumi.nome      AS fiume,
fiumi.lunghezza AS len,
regioni.nome    AS regione
FROM fiumi, regioni, passa_da
WHERE
passa_da.id_fiume = fiumi.id AND
passa_da.id_regione = regioni.id
ORDER BY fiumi.lunghezza DESC;
```

La query si può inserire in modo permanente nel database sotto forma di *vista*. Lo scopo delle viste è astrarre l'applicazione dai dettagli. Le viste tendono infatti a mantenere la stessa struttura verso le applicazioni nonostante le tabelle sottostanti cambino per il mutare dell'informazione reale rappresentata.

Questa query trasformata in vista può essere ulteriormente migliorata scrivendo questa istruzione SQL in cui la funzione scalare predefinita `group_concat()` concatenerà i nomi delle regioni in una lista separata da virgole:

TABELLA 1: Una tabella ottenuta con la compilazione di un sorgente LuajitLaTeX nel quale il codice esegue una query verso un database SQLite3, come spiegato alla sezione 7.1.

N.	Fiume	Lunghezza (km)	Regioni attraversate
1	Po	652	Piemonte, Lombardia, Emilia-Romagna, Veneto
2	Adige	410	Trentino-Alto Adige, Veneto
3	Tevere	405	Emilia-Romagna, Toscana, Umbria, Lazio
4	Adda	313	Lombardia
5	Oglio	280	Lombardia
6	Tanaro	276	Piemonte, Liguria
7	Ticino	248	Svizzera, Piemonte, Lombardia
8	Arno	241	Toscana
9	Piave	220	Veneto
10	Reno	210	Toscana, Emilia-Romagna
11	Sarca-Mincio	203	Trentino-Alto Adige, Veneto, Lombardia
12	Volturno	175	Molise, Campania
13	Brenta	174	Trentino-Alto Adige, Veneto
14	Secchia	172	Emilia-Romagna, Lombardia
15	Ofanto	170	Campania, Basilicata, Puglia
16	Tagliamento	170	Friuli-Venezia Giulia, Veneto
17	Ombrone	161	Toscana
18	Chiese	160	Trentino-Alto Adige, Lombardia
19	Dora Baltea	160	Valle d'Aosta, Piemonte
20	Liri-Garigliano	158	Abruzzo, Lazio, Campania

```
CREATE VIEW vw_passa_da AS
SELECT
fiumi.nome      AS fiume,
fiumi.lunghezza AS len,
group_concat(regioni.nome, ", ") AS regioni
FROM fiumi, regioni, passa_da
WHERE
passa_da.id_fiume = fiumi.id AND
passa_da.id_regione = regioni.id
GROUP BY fiumi.id
ORDER BY fiumi.lunghezza DESC;
```

4.6 LuajitTeX

Entra finalmente in scena il protagonista: LuajitTeX, il nuovo motore di composizione identico a LuaTeX tranne per il fatto che anziché contenere l'interprete tradizionale Lua, contiene LuaJIT.

LuaJIT è un interprete Lua a cui sono state aggiunte alcune funzionalità, in particolare quella per cui il codice viene compilato con tecniche dinamiche dette *just in time*. Queste tecniche di ottimizzazione hanno per effetto il miglioramento dei tempi di esecuzione.

Devo però chiarire il perché, trattandosi di linguaggio interpretato, ho comunque parlato di *compilazione* per Lua, termine riservato ai linguaggi il cui codice prima dell'esecuzione viene tradotto in codice macchina. Ogni volta che Lua esegue un blocco di codice non lo fa direttamente: esegue prima il processo di compilazione che produce una forma virtuale di linguaggio macchina chiamato *bytecode*, poi esegue il bytecode stesso.

Maggiori informazioni sulla tecnologia di compilazione just-in-time possono essere reperite nell'articolo di Luigi Scarso (SCARSO, 2013), dedicato a LuajitTeX.

LuaJIT comprende anche l'estensione `ffi`, *foreign function interface*, che rende molto semplice il binding con librerie esterne rispetto a quanto si può fare con Lua, non richiedendo la scrittura e la compilazione di codice in linguaggio C. Spero di ritornare sull'argomento in un prossimo articolo.

Per generare una connessione a un database SQLite da Lua o da LuaJIT — e quindi anche dai corrispondenti motori di composizione in dotazione al sistema TeX — il modo più efficace è quello di creare un binding, ossia un componente software che rende possibile l'esecuzione di funzionalità contenute nella libreria dinamica di gestione del database scritta in C, all'interno di un componente esterno, nel nostro caso uno script in Lua.

E proprio grazie alle funzionalità `ffi` che in LuaJIT è possibile scrivere il binding semplicemente digitando le *firme* delle funzioni della libreria ospite, anzi, non è nemmeno necessario farlo perché lo ha già fatto Stefano Peluchetti nell'ambito del suo progetto SciLua.org con il modulo che egli ha chiamato `LJSQlite3` e che trovate alla pagina web `http://scilua.org/ljsqlite3.html`.

Anche per Lua esiste il binding grazie al progetto Kepler — indirizzo web `http://keplerproject.github.io/luasql/` — che si concretizza nel modulo LuaSQL capace di connettersi a database Oracle, MySQL, SQLite, Firebird e PostgreSQL e a quelli che implementano i protocolli ODBC e ADO.

LuaSQL dovrebbe essere la scelta da privilegiare perché al progetto Kepler appartengono gli stessi autori di Lua, e non solamente perché è sicuramente più stabile del piccolo modulo `LJSQLite3`.

Tuttavia ho scelto per i miei progetti di gestione dati con SQLite, proprio il pacchetto `LJSQLite3` per una ragione del tutto casuale: quando iniziai nel 2012, sul mio computer non funzionava bene il gestore di pacchetti Luarocks e non riuscii a configurare LuaSQL nemmeno manualmente. Di contro, l'installazione di `LJSQLite3` fu banale: bastò copiare i file in una sottocartella dove si trovava installato LuaJIT.

In disaccordo con la scelta più logica, preferisco qui puntare su moduli software più esotici certo, ma molto veloci nell'esecuzione e d'immediata installazione.

Altro progetto interessante è LuaSQLite3 alla pagina web http://lua.sqlite.org/index.cgi/index che propone una libreria che si basa sul binding diretto tra Lua e SQLite. Ho scoperto questo modulo durante la stesura dell'articolo consultando il sito web che propone una dettagliata pagina dedicata alla documentazione che mostra chiaramente la mappatura delle funzioni dell'`api` di SQLite, ma non l'ho né installato né provato. Esso dovrebbe poter essere utilizzabile anche in LuaTEX e, per come è costruito, dovrebbe offrire una buona velocità di esecuzione assieme alla completezza di funzioni.

4.7 La ricerca dei file

Abbiamo ancora un particolare che riguarda la distribuzione TeX Live e che ci servirà tra poco per capire come rendere raggiungibili i file delle librerie dai programmi di composizione.

Quando un programma della famiglia TEX cerca un file lo fa attraverso l'utility `kpathsea` che a sua volta legge file binari chiamati database `ls-R` che racchiudono tutta la rappresentazione ad albero di una directory nel file system della ditribuzione.

La ricerca del percorso di un file, per esempio un pacchetto LATEX, avviene sfruttando solamente i database `ls-R` delle directory specificate dall'utente attraverso i file di configurazione `texmf.cnf`.

Sapendo che per `YYYY` s'intende l'anno di uscita della distribuzione, il file `texmf.cnf` principale si trova nel percorso:

```
texlive/YYYY/texmf-dist/web2c/texmf.cnf
```

Leggerne i commenti con un po' di pazienza, permette di capire come sono determinate le directory di ricerca: attraverso variabili e l'espansione di espressioni.

Ci interessano particolarmente le linee 448 e 449 — numeri di linea per il file `texmf.cnf` della versione 2016 — che riporto di seguito su tre linee anziché su due, tra i margini di colonna:

```
% Architecture independent executables.
TEXMFSCRIPTS =
    $TEXMF/scripts/{$progname,$engine,}//
```

Si tratta della definizione di dove nel file system possono essere reperiti file Lua da parte dei programmi di TeX Live.

Tabella 2: Directory contenute nella variabile `$TEXMF` per la distribuzione TeX Live dell'autore. I doppi punti esclamativi iniziali del percorso impongono che verranno ricercati i file solamente tra quelli indicizzati nel database `ls-R` e mai sul disco. Nelle prime righe sono riportati i percorsi nell'albero principale, nella riga centrale il percorso dell'albero locale, e nelle ultime righe quelle dell'albero personale.

Percorsi contenuti nella variabile `$TEXMF`
`!!/usr/local/texlive/2016/texmf-config`
`!!/usr/local/texlive/2016/texmf-var`
`!!/usr/local/texlive/2016/texmf-dist`
`!!/usr/local/texlive/texmf-local`
`/home/roberto/.texlive2016/texmf-config`
`/home/roberto/.texlive2016/texmf-var`
`/home/roberto/texmf`

La variabile `$TEXMF` contiene molte directory riportate nella tabella 2, come è possibile verificare dando in una finestra di console il comando:

```
$ kpsewhich --expand-var '$TEXMF'
```

L'albero locale della TeX Live rende disponibili i pacchetti per tutti gli utenti che accedono al computer, perciò è ragionevole installare i file di libreria in questa parte del file system TDS. Per di più quando si aggiorna la distribuzione con quella dell'anno successivo, l'albero locale non viene alterato dal procedimento ed è di nuovo funzionante con la nuova TeX Live.

In definitiva, uno script per LuajitTEX verrebbe ricercato nel percorso (da leggere come se fosse solo su una riga):

```
!!/usr/local/texlive/texmf-local/
    scripts/luajittex
```

I doppi punti esclamativi indicano che la ricerca dei file verrà effettuata solamente tra i percorsi indicizzati nel database `ls-R`, che va quindi rigenerato dopo aver copiato i file nell'albero locale con il comando `mktexlsr`.

Si lascia al lettore il compito di convertire i percorsi in quelli equivalenti per il sistema operativo Windows o per OS X.

4.8 Installare `LJSQLite3` per LuajitTEX

Completando le operazioni descritte in questa sezione si avrà l'accesso a database SQLite direttamente dall'interno di un sorgente TEX. La cosa dovrebbe fare un certo effetto.

Del resto Lua introduce nel mondo tipografico di TEX possibilità davvero innovative che lasciano immaginare sviluppi futuri a dir poco spettacolari.

I passi dell'installazione sono i seguenti:

1. scaricare il pacchetto `LJSQLite3` sotto forma di file compresso dal repository GitHub all'indirizzo https://github.com/stepelu/lua-ljsqlite3, per un peso inferiore a 10KB.

Molto comodo è ricorrere al pulsante "Clone or download" che si trova sulla destra della pagina web;

2. decomprimere e rinominare la cartella di `LJSQLite3` in `ljsqlite3` con sole lettere minuscole; all'interno della cartella rinominare il file `init.lua` in `ljsqlite3.lua`;

3. scaricare il file del modulo `lua-xsys` dalla pagina GitHub https://github.com/stepelu/lua-xsys, una dipendenza del pacchetto precedente che ne incorpora una ulteriore chiamata `templet`;

4. rinominare allo stesso modo la cartella decompressa di `xsys` con lo stesso nome del pacchetto; all'interno rinominiamo il file `init.lua` in `xsys.lua`;

5. rinominiamo anche il file `init.lua` in `templet.lua` all'interno della sub-directory `_dep/templet`;

6. con un editor di testi modificare l'istruzione di caricamento del modulo `templet` all'interno del file `xsys.lua` così rinominato al passo 4: intorno alla riga 18 modificare il codice (riportato su due righe anziché una) da:

```
local templet =
   require "xsys._dep.templet"
```

   in

```
local templet = require "templet"
```

7. copiare nella cartella dell'albero locale `/usr/local/texlive/texmf-local` di TeX Live, o nell'equivalente per il vostro sistema operativo e la vostra configurazione, la cartella `scripts/luajittex` (da creare se non esiste); percorso determinato alla sezione 4.7;

8. eseguire il comando `mktexlsr` per rigenerare i database `ls-R`.

Il cambiamento dei nomi dei file e delle cartelle si rende necessario a causa del differente comportamento delle funzioni di caricamento di LuaJIT e LuajitTEX.

Tutte le operazioni manuali sono semplici anche se sappiamo che la maggior parte di esse non sarebbe stata necessaria per lavorare con LuaJIT mentre lo sono con LuajitTEX.

4.9 LuajitTEX in azione

Per verificare che tutto funzioni è sufficiente compilare con LuajitTEX il sorgente seguente dove l'unica cosa che viene eseguito è il caricamento della libreria `LJSQLite3`:

```
% !TeX program = LuajitTeX

\directlua{
local sql = require "ljsqlite3"
}
\bye
```

Un primo esempio di codice consiste nell'effettuare la connessione al database `fiumi.db` prodotto in precedenza e stampare l'elenco dei fiumi memorizzati nella tabella `fiumi`.

La documentazione del modulo di binding `LJSQLite3` all'indirizzo web http://scilua.org/ljsqlite3.html, consente abbastanza semplicemente di ricavare il codice:

```
% !TeX program = LuajitTeX

\directlua{
local sql = require "ljsqlite3"
local conn = sql.open("fiumi.db", "ro")

local query = [[
    SELECT nome, lunghezza FROM fiumi
    ORDER BY lunghezza DESC;
]]
local res, nr = conn:exec(query)
conn:close()
for i = 1, nr do
    tex.print(i)
    tex.print(res.nome[i])
    tex.print(tonumber(res.lunghezza[i]))
    tex.print("")
end
}
\bye
```

e il risultato è:

```
1 Po 652
2 Adige 410
3 Tevere 405
4 Adda 313
5 Oglio 280
6 Tanaro 276
7 Ticino 248
8 Arno 241
9 Piave 220
10 Reno 210
```

Dopo aver caricato la libreria `LJSQLite3`, si crea la connessione dati con la funzione `open()` che accetta come primo argomento il nome del file del database, che in questo caso si trova nella stessa cartella del sorgente, e come secondo parametro la stringa che determina la modalità di apertura. La chiave `"ro"` sta per read-only.

Sull'oggetto *connessione* viene chiamato poi il metodo `exec()` passando come argomento il testo che rappresenta la query in linguaggio SQL. Il metodo restituisce due riferimenti: il result set con tutti i dati richiesti dalla query e il numero dei record trovati.

Si chiude poi la connessione chiamando il metodo `close()`[4], mentre poi un ciclo iterativo provvede a stampare i dati nel documento con chiamate alla funzione `tex.print()`.

La funzione `tonumber()` converte il tipo originario intero `cdata` nel tipo numerico nativo di Lua. Se non la inserissimo la funzione `tex.print()` segnalerebbe un errore perché non è in grado da sola di effettuare la conversione.

4.9.1 Struttura del result set

Di ritorno dalla query, i dati vengono memorizzati in una normale tabella Lua che ha però alcune peculiarità: i dati dell'*i*-esima tupla risultato sono indicizzati sia con la chiave corrispondente al nome della colonna derivante dalla query, in questo caso dalle chiavi `nome` e `lunghezza`, che con il numero della colonna a partire da 1. In modo equivalente sia il nome della colonna sia l'indice intero indicizzano le tuple ordinate.

In altre parole il result set è una tabella contenente le tabelle/array delle *colonne*. Valgono le uguaglianze:

```
res.nome[1] = res[1][1] = "Po"
res.lunghezza[1] = res[2][1] = 652
```

e in generale:

```
res.<colName1>[<i-tupla>] = res[1][<i-tupla>]
res.<colName2>[<i-tupla>] = res[2][<i-tupla>]
...
res.<colNameN>[<i-tupla>] = res[N][<i-tupla>]
```

Il metodo `exec()` accetta anche il parametro opzionale di tipo stringa che configura la struttura del result set. In particolare, se esso contiene il carattere `"h"` che sta per header, l'elemento di indice zero conterrà i nomi delle colonne.

4.10 Compilare con LuajitLATEX

Su alcuni sistemi compreso Linux, in TeX Live non è disponibile il programma LuajitLATEX, l'equivalente di LuaLATEX. I passi che ho seguito per rimediare in modo semplice e veloce non prevedendo la creazione di un eseguibile, sono questi:

1. abilitare il formato modificando il file `fmtutil.cnf` principale (oppure locale) della distribuzione;
2. creare il file precompilato;
3. abilitare lo shell editor alla compilazione con il formato.

Per il passo 1 ho dato il seguente comando da digitare su una sola linea (da adattare se il sistema non è un derivato Debian):

4. Questo passaggio non è essenziale perché Lua chiude la connessione in automatico al termine del blocco di codice un momento prima di distruggere l'oggetto in memoria, ma è buona norma chiudere la connessione nel momento in cui non è più necessaria.

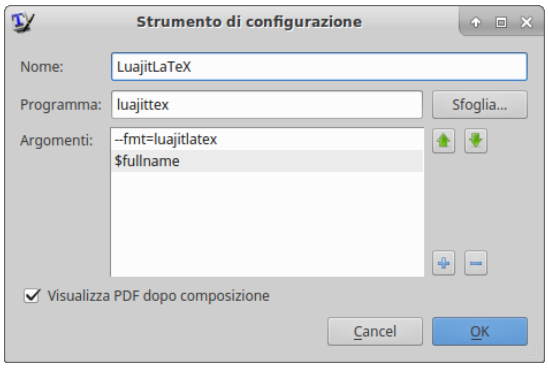

FIGURA 1: Dialogo dello shell editor TeX Works di configurazione del comando per la compilazione di sorgenti nel formato LuajitLATEX sui sitemi che hanno questa configurazione disabilitata. Il comando è poi selezionabile specificandone il nome all'interno delle righe magiche, come mostrato nei sorgenti completi riportati nel testo.

```
$ sudo gedit /usr/local/texlive/
    2016/texmf-dist/web2c/fmtutil.cnf
```

Basta decommentare la riga seguente (riportata qui su due linee) togliendo i caratteri iniziali `#!` che stanno a indicare una configurazione disabilitata:

```
luajitlatex luajittex language.dat,
    language.dat.lua lualatex.ini
```

Per il passo 2 il comando è:

```
$ sudo fmtutil-sys --byfmt luajitlatex
```

Adesso dovrebbe essere possibile compilare sorgenti nel formato LuajitLATEX con il comando:

```
$ luajittex --fmt=luajitlatex <nome-file>
```

Per comodità, questo comando può essere lanciato automaticamente dall'interno di uno shell editor premendo il bottone o la combinazione di tasti prevista. Con TeX Works si configura nel dialogo delle Preferenze, scheda Composizione, un nuovo comando come mostrato in figura 1, il cui nome può essere inserito nelle righe magiche del sorgente. Le istruzioni di lancio del comando comprende l'opzione `-fmt=luajitlatex`.

La verifica può essere fatta sul seguente sorgente:

```
% !TeX program = LuajitLaTeX

\documentclass{article}
\usepackage{booktabs}
\begin{document}

\begin{tabular}{lc}
\toprule
Nome & Lunghezza (km)\\
\midrule
\directlua{
local sql = require "ljsqlite3"
local conn = sql.open("fiumi.db", "ro")

local query = [[
```

```
    SELECT nome, lunghezza FROM fiumi
    ORDER BY lunghezza DESC;
]]
local res, nr = conn:exec(query)
conn:close()
for i = 1, nr do
    tex.print(res.nome[i])
    tex.print("&")
    tex.print(tonumber(res.lunghezza[i]))
    tex.print("\string\\\string\\")
end
}
\bottomrule
\end{tabular}
\end{document}
```

5 La regola aurea

Termino questa sezione introduttiva enunciandovi una regola che osservo senza alcuna eccezione. Non farlo genera inconvenienti.

La regola aurea che l'utente TEX deve rispettare quando si connette a un database è questa:

Connessioni in sola lettura!

In altre parole la regola equivale a usare i motori di composizione LuaTEX o LuajitTEX esclusivamente come programmi di reporting. Le operazioni di popolamento, cancellazione e modifica dovranno essere implementate con altri strumenti.

I dati sono quindi la risorsa principale al centro del sistema, in cui componenti software inviano informazioni verso il database, mentre altri del tutto indipendenti dai precedenti leggono i dati con cui realizzare report.

6 Dati e comandi

Dedicherò questa parte dell'articolo alla ricerca di soluzioni per creare un ponte tra i dati disponibili in Lua e la loro rappresentazione come oggetti tipografici composti, attraverso la definizione di comandi TEX.

Una volta eseguite le query verso il database, i dati risiederanno in memoria come strutture disponibili all'interno di Lua. Essi potranno fluire verso la rappresentazione a stampa nel documento finito per elaborazione TEX, in due modi:

- inserimento di caratteri nello stream di *token*,
- inserimento di oggetti tipografici completi nello stream delle liste principali del *processo visuale* di TEX.

Il primo modo si concretizza nelle varie funzioni `print` del modulo interno `tex` di LuaTEX o LuajitTEX, per esempio come nel seguente sorgente:

```
% !TeX program = LuaTeX

\directlua{
tex.print(math.pi)
}
\bye
```

L'espansione della primitiva `\directlua` è vuota ma l'esecuzione del codice Lua ha in questo caso, l'effetto di inserire le cifre di π — fino a una certa precisione — come se fossero già state presenti nel file sorgente. In altre parole, quella funzione immette caratteri alimentando lo stadio iniziale dell'elaborazione del motore tipografico.

La seconda modalità è molto più sofisticata. Consiste nel creare oggetti Lua chiamati *nodi* equivalenti a quelli presenti nelle liste di composizione dell'ultima fase — detta visuale — quando gli elementi tipografici ormai completi vengono composti sulla pagina.

I nodi devono formare liste concatenate. Ve ne sono molti diversi tipi come glifi, grandezze elastiche, e scatole orizzontali e verticali, che formano quindi catene di oggetti elementari ricchi di dettagli tipografici e costruiti con la precisione del punto scalato, l'unità di misura interna di TEX che corrisponde a una frazione davvero minuscola del millimetro.

Nel seguito utilizzerò il primo modo descritto con la stampa di token, meno sofisticato ma più semplice che non la creazione di oggetti tipografici in Lua, al fine di concentrare l'attenzione sull'elaborazione dei dati e sulla scrittura di macro per l'utente finale. Il lettore può riferirsi al manuale di LuaTEX (The LuaTEX development team, 2016) o ai miei due recenti articoli apparsi nel 2016 su *Ars*, (Giacomelli, 2016b), (Giacomelli, 2016a) per approfondire la tecnologia dei nodi.

6.1 Espressioni

L'unico oggetto strutturato di Lua — non considerando possibili estensioni scritte in C o anche in altri linguaggi — è la *tabella*, allo stesso tempo un dizionario chiave/valore e/o un array a una dimensione.

La particolare sintassi Lua detta *dot notation* restituisce il valore associato alla chiave stringa con qualifica di identificatore:

```
math.pi == math["pi"]
```

ed è un'espressione. Potremo quindi scrivere la seguente macro `\print` che accetta un testo che corrisponde a una espressione Lua:

```
% !TeX program = LuaTeX

\def\print#1{%
\directlua{tex.print(#1)}}

\print{math.pi}
\bye
```

Con questa semplicissima macro è possibile inserire nel documento qualsiasi espressione Lua e

in particolare, non solo valori associati a chiavi di tabelle, ma anche chiamate di funzione:

```
% !TeX program = LuaTeX

\def\print#1{%
\directlua{
tex.print(#1)
}}

\begingroup
  \catcode`\%=12\relax
  \gdef\prc{%}
\endgroup

% definizione di una funzione
\directlua{
cerchio = {}
cerchio.area = function(r, prec)
    prec = prec or 2
    local fmt = "\prc0."..prec.."f"
    return string.format(fmt, math.pi*r^2)
end
}

\print{cerchio.area(5)}
\print{cerchio.area(15, 6)}
\print{cerchio.area(2) - cerchio.area(1)}
\bye
```

6.2 Stampa condizionale

Per stampa condizionale intendo una macro TEX che stampa il testo che corrisponde al valore di una condizione vera o falsa, la cui definizione di base potrebbe essere la seguente:

```
% !TeX program = LuaTeX

\def\ifprint#1#2#3{%
\directlua{
if #1 then
    tex.print([==[#2]==])
else
    tex.print([==[#3]==])
end
}}

\directlua{x = 5}

\ifprint{x > 10}{%
$x$ maggiore di 10}{%
$x$ non maggiore di 10}
\bye
```

Il primo argomento della macro `\ifprint` è l'espressione Lua che è valutata nel valore booleano vero o falso, il secondo argomento è il testo da stampare se la condizione è vera, e il terzo argomento è il testo da stampare se la condizione è falsa.

Non è necessario racchiudere gli argomenti testuali in apici o doppi apici poiché nel normale meccanismo di sostituzione degli argomenti, TEX ne espanderà il valore all'interno di delimitatori estesi del tipo stringa di Lua — la coppia simmetrica `[==[` e `]==]`.

Interessante notare che l'espansione degli argomenti fa si che questo comando funzioni senza alcun problema:

```
\ifprint{x < 10}{%
$x = \print{x}$ maggiore di 10!}{%
$x = \print{x}$ non maggiore di 10}
```

Infatti la macro condizionale `\ifprint` riceverà gli argomenti già espansi, perciò con il valore della variabile `x` al posto dei token `\print{x}`.

6.3 Stampa se l'espressione è vera

Potremo trovarci a scrivere macro condizionali `\ifprint` per stampare un testo solamente nel caso in cui un oggetto esiste oppure no. Uno dei due argomenti testuali dovrebbe essere un argomento tra graffe vuoto. Tuttavia un comando specifico avrebbe una semantica più significativa.

La definizione seguente è altrettanto semplice di quelle precedenti:

```
% !TeX program = LuaTeX

\def\iftrueprint#1#2{%
\directlua{
if #1 then
    tex.print([==[#2]==])
end
}}

\directlua{x = 5}
\iftrueprint{x > 10}{%
$x$ maggiore di 10}
\bye
```

6.4 Stampa con iteratori

Un iteratore è un modo semplice di elaborare uno alla volta gli elementi di una collezione di dati. In Lua si utilizza il costrutto chiamato *generic for* specificando i nomi delle variabili, che saranno disponibili all'interno del ciclo, e la funzione che restituisce l'iteratore. Traslando il generic for in una macro potremo scrivere:

```
\def\iterprint#1in#2do#3{%
\directlua{
for #1 in #2 do
    tex.print(#3)
end
}}
```

Sfruttando la tecnica ad argomenti delimitati, la macro `\iterprint` accetta come argomento 3 un'espressione che potrà elaborare le variabili di ciclo denominate all'argomento 1, e il cui risultato sarà valutato a ogni iterazione del ciclo definito dall'iteratore dato come argomento 2, per poi essere immesso nel testo del documento.

Se volessimo stampare tutte le coppie chiavi/valore contenute in una tabella potremo usare l'iteratore predefinito `pairs()` che restituisce a

ogni ciclo in prima posizione la chiave e in seconda il corrispondente valore. Il seguente codice stamperà tutti gli oggetti contenuti nella tabella `math` che è il modulo matematico di Lua:

```
% !TeX program = LuaTeX

\def\iterprint#1in#2do#3{%
\directlua{
for #1 in #2 do
    tex.print(#3)
end
}}

\iterprint k, v in pairs(math) do {
    k.." = "..type(v).."\string\\par"
}
\bye
```

Invece di un'espressione è possibile stampare il risultato di una funzione. In questo modo gli elementi della collezione vengono mappati e/o filtrati in una seconda collezione di oggetti. Questo esempio spiega meglio l'operazione:

```
% !TeX program = LuaTeX

\def\itermapprint#1in#2do#3{%
\directlua{
local function tmp (#1)
    #3
end
for #1 in #2 do
    local s = tmp(#1)
    if s then
        tex.print(s)
    end
end
}}

\itermapprint k, v in pairs(math) do {
    if type(v) == "number" then
        return k.." = "..v.."\string\\par"
    end
}
\bye
```

La `\itermapprint` rispetto alla macro precedente esegue il proprio argomento 3 come una funzione con gli argomenti di ciclo. Se l'esecuzione restituisce un valore esso verrà stampato altrimenti si salta alla successiva iterazione.

Questo è proprio quello che succede nel codice di esempio dove solamente gli elementi di tipo numerico della tabella `math` vengono mappati nella stringa field name = numeric value.

La mappatura o il filtraggio possono essere anche espressi come funzioni che accettano un iteratore e ne restituiscono un secondo. Tuttavia ne risulterebbe una sintassi poco chiara perché, per come sono costruiti gli iteratori in Lua, non è possibile concatenarli con la sintassi in dot notation, ma solamente come funzione dentro funzione.

6.5 Includere query

Queste macro possono essere adattate per eseguire query e lasciare che l'utente gestisca con un'espressione i risultati. I comandi ad argomenti delimitati rendono semplice la definizione, e una libreria in Lua può nascondere i dettagli del codice per le interrogazioni pur mantenendo la generalità.

Scriveremo queste funzionalità nella prossima sezione, sviluppando l'esempio della creazione di report rispetto al semplice database sui fiumi.

7 Fiumi

7.1 Da tabella a tabular

Nella sezione 4.4 ho definito un semplice database che contiene dati sui principali fiumi italiani. In particolare esso contiene le lunghezze in km e le regioni attraversate per ciascun fiume. In questa sezione spiegherò come, con le conoscenze acquisite nelle sezioni precedenti, sia possibile ottenere la tabella 1 riportata a pagina 5.

Prepariamo allo scopo alcune funzioni Lua che ci serviranno per connetterci al database e iterare le tuple risultato della query per creare le righe della tabella. Requisito di progetto di queste funzioni è la generalità, cioè dovremo fare in modo che valgano per qualsiasi query su un qualsiasi database SQLite.

Salviamo nella stessa directory del database un file di testo che chiameremo `dblib.lua`, con il codice diviso in quattro parti:

1. caricamento della libreria LuaJIT in una variabile locale;
2. creazione di una tabella di modulo;
3. definizione delle funzioni e dei parametri come oggetti da assegnare a chiavi nella tabella di modulo;
4. istruzione di ritorno della tabella di modulo.

Quando questo file verrà caricato con l'istruzione `require()` all'interno del sorgente LuajitLaTeX, potremmo disporre delle funzionalità definite nella tabella di modulo mentre la libreria LJSQLite3 sarà un oggetto nascosto all'interno della closure d'ambiente.

Per provare che funzioni questa semplice ma efficace soluzione, scriviamo dapprima solamente le funzioni per l'apertura in sola lettura e la chiusura di una connessione a un database, per la nostra libreria:

```
local sql = require "ljsqlite3" -- parte 1
local db = {} -- parte 2

function db:connect(dbpath) -- parte 3
    self.conn = sql.open(dbpath, "ro")
end
```

```
function db:close()
    self.conn:close()
end

return db -- parte 4
```

La notazione detta *method call* è stata introdotta in Lua assieme ai metametodi e alle metatabelle per la programmazione a oggetti. Questa particolare sintassi in realtà è molto semplice perché aggiunge un primo argomento con il riferimento alla tabella stessa con il nome di `self` alla funzione. Si tratta solamente di *zucchero sintattico*, cioè una funzionalità dell'interprete, e non del linguaggio, che semplifica il codice. Le due espressioni seguenti sono del tutto equivalenti:

```
-- method call
db:connect("dbname")

-- equivale a chiamare:
db.connect(db, "dbpath")
```

Con il method call, la funzione `connect()` memorizza all'interno della tabella di modulo stessa il riferimento alla connessione del database, con uno stile non proprio pulito. Dovremo infatti utilizzare pienamente il paradigma a oggetti ma qui ci interessa un codice più semplice possibile per concentrarci sull'obiettivo. Il sorgente LuajitTEX che mostra come usare la libreria appena scritta è il seguente, ipotizzando che il database sia un file chiamato `fiumi.sqlite`:

```
% !TeX program = LuaJitTeX

\directlua{
local dblib = require "dblib"
dblib:connect("fiumi.sqlite")
dblib:close()
}
\bye
```

ed essendo un documento vuoto, nessun PDF sarà composto. Lo scopo è quello di controllare se tutto sia in ordine e funzionante. Passiamo ora a scrivere la funzione centrale: essa dovrà eseguire la query e, compito più complesso, restituire un iteratore sull'insieme delle tuple della tabella risultato.

L'iteratore è un buon modo per rendere semplice l'elaborazione delle tuple. Tecnicamente però, i dati si trovano già tutti caricati in memoria al momento dell'iterazione e perciò non si tratta di un vero *cursore* che invece è una funzionalità del database.

L'implementazione dell'iteratore che useremo tra poco è questa:

```
function next(inv_state, ctrl_var)
    local res = inv_state.rs
    local nr = inv_state.nr
    ctrl_var = ctrl_var + 1
    if ctrl_var <= nr then
        local cols = #res
        local t = {}
        for i = 1, cols do
            local data = res[i][ctrl_var]
            if type(data)  == "cdata" then
                data = tonumber(data)
            end
            t[#t+1] = data
        end
        return ctrl_var, t
    end
end

function db:iter_row(query, nrecords)
    local conn = self.conn
    -- prepared statement:
    local stmt = conn:prepare(query)
    local rs, nr =
        stmt:resultset(nil, nrecords)
    stmt:close()
    return next, {nr = nr, rs = rs}, 0
end
```

La funzione `iter_row()` ha due argomenti, il primo è la stringa contenente il codice SQL che rappresenta la query, il secondo è il numero massimo dei record da restituire in base all'ordinamento stabilito dalla query. Se questo parametro non è fornito vengono restituiti tutti i record.

Dovremo quindi scrivere la query come codice SQL, ma era implicito nelle specifiche di progetto, quando si richiedeva la generalità sull'interrogazione. Sarebbe possibile impiegare un modulo software chiamato ORM acronimo di Object Relational Mapping, per esempio http://fperrad.github.io/lua-CoatPersistent/. Un ORM permetterebbe anche di astrarre il codice anche rispetto al particolare database management system.

Nel dettaglio, all'inizio dell'iterazione la funzione `iter_row()` viene eseguita e i tre valori di ritorno assegnati ai parametri interni del ciclo `for`. Il primo è la funzione da chiamare per ottenere gli elementi, il secondo è lo stato invariante e il terzo un indice di controllo.

Si tratta quindi di uno *stateless iterator* perché la funzione `next()` non fa uso di variabili interne alla closure corrispondente, ma solamente dei tre parametri interni. Se vogliamo, anche il costrutto `for` può essere considerato zucchero sintattico nel senso che esso è equivalente a un opportuno ciclo `while`.

Osserviamo infine che la funzione `next()` prende come argomenti lo stato invariante e l'indice di controllo e restituisce il valore da assegnare all'indice di controllo per il ciclo successivo e le variabili d'iterazione, che in questo caso è la sola tabella array del record corrente.

L'iteratore così definito, rispettando le specifiche previste dal generic for di Lua, può essere impiegato in una delle due macro introdotte nella sezione 6.4. Il sorgente è il seguente:

```
% !TeX program = LuaJitLaTeX
```

```
\directlua{
dblib = require "dblib"
dblib:connect("fiumi.sqlite")
}

\def\iterprint#1in#2do#3{%
\directlua{
for #1 in #2 do
    tex.print(#3)
end
}}

\def\endrow{\string\\\string\\}

\documentclass[margin=2pt]{standalone}
\usepackage{luatex85}
\usepackage{fontspec}
\usepackage{booktabs}
\begin{document}
\small
\begin{tabular}{llcl}
\toprule
N. & Fiume & Lunghezza (km) & Regioni
attraversate\\
\midrule
\iterprint i, row in
    dblib:iter_row(
      [[SELECT * FROM vw_passa_da;]],
      15
    ) do {
    i.."&"..table.concat(row, "&")..
    "\endrow"
    }
\bottomrule
\end{tabular}
\directlua{dblib:close()}
\end{document}
```

Interessante notare che in questo esempio il codice Lua occupa solamente un nome globale, nome che è possibile cambiare e che è stato definito come `dblib`. Ciò praticamente annulla i problemi di collisioni di nomi di variabili globali in documenti complessi in cui si caricano molti pacchetti Lua diversi.

Il pacchetto luatex85 definisce alcune macro per ripristinare i vecchi nomi di primitive che furono cambiati nella versione 0.85 di LuaTEX per rendere più chiaro la distinzione tra motore di composizione e formato di uscita. Lo faccio notare perché una volta che gli Autori dei pacchetti avranno aggiornato il proprio codice, questo pacchetto non sarà più necessario.

Per esempio, se nel nostro sorgente non caricassimo il pacchetto luatex85 riceveremmo un errore di Undefined control sequence per `\pdfpagewidth` utilizzata proprio dalla classe standalone. L'alternativa al pacchetto è copiare le definizioni riportate nel manuale di LuaTEX unicamente per le primitive interessate dalla compilazione del documento.

7.2 TEX si connette a un database

Invece di creare e chiudere la connessione al database attraverso codice Lua diretto come abbiamo fatto nel listato precedente, scriviamo due macro generiche. Stiamo quindi creando diversi strati di funzionalità: una sorta di pacchetto utente per fornire le macro di alto livello e un modulo Lua sottostante scritto in un file dedicato.

Le macro potrebbero essere le seguenti, anche se non contengono alcun controllo sugli errori che qui appensentirebbe la comprensione del codice ma che in realtà è irrinunciabile:

```
\def\dbconnect#1{
\directlua{
local dblib = require "dblib"
dblib:connect([==[#1]==])
}}

\def\dbclose{
\directlua{
local dblib = require "dblib"
dblib:close()
}}
```

La funzione `require()` primitiva di Lua, è utilizzata ogni volta per ottenere il riferimento alla tabella che contiene le nostre funzioni del livello base. Se la libreria è già stata caricata essa risulta memorizzata come riferimento nella tabella `package.loaded` e la funzione `require()` restituirà semplicemente quel riferimento. Ad essere unico non dovrà più essere il nome della variabile globale che contiene il modulo ma solamente il nome del modulo stesso.

7.3 Un miglioramento: funzioni formato

All'interno della macro `\iterprint` nel listato del sorgente precedente che produce la tabella sui primi 15 fiumi italiani, dobbiamo ricreare la riga di testo con tutti i campi separati dal carattere `&` e con il doppio backslash finale. Ciò non è proprio elegante.

In dettaglio, la concatenazione di questi campi avviene nell'argomento 3 della macro `\iterprint`, dove dobbiamo inserire un'espressione Lua. Nulla vieta di sostituire l'espressione con la chiamata a una funzione che fa il lavoro di concatenazione per noi.

La nuova versione del sorgente è questa:

```
% !TeX program = LuaJitLaTeX

\def\dbconnect#1{...} % as before
\def\dbclose{...}

\def\iterprint#1in#2do#3{%
\directlua{
for #1 in #2 do
    tex.print(#3)
end
}}
```

```
\dbconnect{fiumi.sqlite}
\directlua{dblib = require "dblib"}

\documentclass[margin=2pt]{standalone}
\usepackage{luatex85}
\usepackage{fontspec}
\usepackage{booktabs}
\begin{document}
\small
\begin{tabular}{llcl}
\toprule
N. & Fiume & Lunghezza (km) & Regioni
attraversate\\
\midrule
\iterprint i, row in
  dblib:iter_row(
     [[SELECT * FROM vw_passa_da;]],
     15
  ) do {
     dblib.concat("&", dblib.endrow, i, row)
  }
\bottomrule
\end{tabular}
\dbclose
\end{document}
```

La funzione `dblib.concat()` si trova quindi nel file di libreria, dobbiamo solo aggiungerla alla sezione 3 del sorgente `dblib.lua`. La funzione è del tutto generale, essa concatena qualsiasi argomento, una volta specificata la stringa di separazione tra i valori e la stringa finale. Oltre ai primi due argomenti obbligatori riceve un qualsiasi numero di variabili, siano esse valori semplici o tabelle, gestendole ricorsivamente.

Si tratta quindi di una funzione piuttosto sofisticata perchè utilizza due peculiarità di Lua: la *tail call*[5] che offre la sicurezza di chiamate ricorsive che *mai* saturano lo stack e il *three dots* che rappresenta un numero variabile di argomenti.

Grazie all'uso di queste e altre caratteristiche avanzate di Lua, l'utente può disporre di funzioni molto flessibili che in questo caso corrispondono a *funzioni formato*, nel contesto dell'elaborazione dei dati per la loro preparazione alla stampa.

Ecco le aggiunte alla libreria:

```
local function conc_rec(buf, sep, t, index)
    if index > #t then return buf end
    local elem = t[index]
    if type(elem) == "table" then
        buf = conc_rec(buf, sep, elem, 1)
    else
        buf[#buf+1] = elem
        buf[#buf+1] = sep
    end
    -- tail call
    return conc_rec(buf, sep, t, index+1)
end

-- wrapper function:
function db.concat(sep, endstr, ...)
    local vararg = {...}
    local buf = conc_rec({}, sep, vararg, 1)
    buf[#buf] = endstr
    return table.concat(buf)
end

db.endrow = [[\\]]
```

Da notare che se ci fossero dei valori `nil` tra gli argomenti variabili, ciò arresterebbe la scansione al primo di essi e i successivi verrebbero ignorati. Prima di apportare modifiche alle funzioni è necessario decidere se trattare il `nil` con il significato di cella vuota oppure no, ignorandolo e saltando all'elemento successivo.

Il controllo più fine sulla lista degli argomenti variabili può esser fatto per mezzo della funzione Lua `select()`.

7.4 Macro e query

Invece di usare la macro `\iterprint` potremo semplificare il codice accettando come argomento non l'iteratore ma direttamente il testo della query in SQL. Chiamiamo quindi questa nuova macro utente come `\selectprint` a indicare con la parte `select` che dovrà essere fornita una query e con la parte `print` che dovrà essere fornita un'espressione che verrà poi valutata e stampata come token nel documento:

```
\def\selectprint#1in#2;#3{%
\directlua{
local db = require "dblib"
local concat = db.concat
local endrow = db.endrow

for #1 in db:iter_row([[#2]]) do
    tex.print(#3)
end
}}
```

che può essere utilizzata all'interno di un ambiente tabular come:

```
\selectprint i, row in
  SELECT * FROM vw_passa_da LIMIT 15;
  {concat("&", endrow, i, row)}
```

7.5 Da tabella a grafico

Sviluppiamo ora un istogramma per rappresentare le lunghezze dei fiumi memorizzati nel database. Questo nuovo lavoro è una concreta applicazione del concetto di modello/vista, sott'inteso a quello di report definito come un documento che contiene dati provenienti da una fonte esterna.

Useremo il pacchetto pgfplots per comporre il grafico costruendo un sorgente statico per la messa a punto del codice che servirà da modello per la versione che caricherà i dati dal database. Il sorgente modello è il seguente dal risultato riportato in figura 2:

5. Maggiori informazioni su questo tipo di chiamata sono reperibili nel manuale di Lua (Ierusalimschy, 2013), e nei testi che trattano gli argomenti della programmazione funzionale.

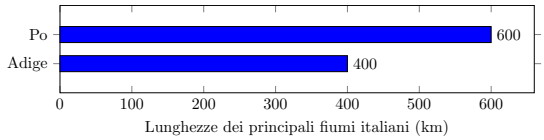

FIGURA 2: Istogramma generato con il pacchetto pgfplots con il codice riportato nel testo per essere utilizzato come base per costruire l'istogramma analogo ma con i dati ricavati da un database.

```
% !TeX program = pdflatex
\documentclass{standalone}
\usepackage{pgfplots}
\pgfplotsset{compat=1.14}

\begin{document}
\begin{tikzpicture}
\begin{axis}[
xbar,
xmin=0,
width=12cm, height=3.5cm,
enlarge y limits=1.0,
xlabel={Lunghezze dei principali fiumi
    italiani (km)},
symbolic y coords={Adige,Po},
ytick=data,
nodes near coords,
nodes near coords align={horizontal}
]
\addplot[draw=black,fill=blue]
coordinates {
(600,Po)
(400,Adige)
};
\end{axis}
\end{tikzpicture}
\end{document}
```

I dati del grafico che devono essere inseriti nel sorgente sono suddivisi in due gruppi: la lista separata da virgole dei nomi dei fiumi da dare all'opzione `symbolic y coord` ordinata dal fiume meno lungo al più lungo, e le coppie di coordinate all'interno del comando `\addplot`, formate dalle coppie ordinate lunghezza e nome fiume. Ricaveremo queste informazioni dal database attraverso query.

Per prima cosa ho messo a punto la query che genera la lista dei nomi dei fiumi a parte, lavorando con il plug-in per Firefox SQLite Manager[6] usando la funzione `group_concat()` su una sub-query[7] che elenca i nomi dei primi 24 fiumi più lunghi ma in ordine inverso, e che richiede a sua volta una query per determinare il numero dei fiumi rimanenti, quelli dalla posizione 24 in poi, da passare al parametro `OFFSET`:

```
SELECT group_concat(nome, ",") FROM (
    SELECT nome FROM fiumi
    ORDER BY lunghezza ASC
    LIMIT 24 OFFSET (
        SELECT count(id) FROM fiumi) - 24
);
```

Poi sono passato alla query che produce la lista delle coordinate, anche questa in sub-query per sfruttare la funzione `group_concat()`:

```
SELECT group_concat(coord , " ") FROM (
SELECT '(' || lunghezza || ',' || nome || ')'
AS coord FROM fiumi
ORDER BY lunghezza DESC
LIMIT 24
);
```

Nel sorgente LuajitLaTeX ho poi inserito il codice SQL all'interno delle macro `\selectprint` a sua volta inseriti in macro utente definite nel preambolo perché il codice sia più ordinato. Il listato completo che genera la figura 3 è risultato così il seguente:

```
% !TeX program = LuaJitLaTeX

% define macro as before

\documentclass[margin=2pt]{standalone}
\usepackage{luatex85}
\usepackage{pgfplots}
\pgfplotsset{compat=1.14}

\newcommand\symbcoord{
\selectprint i, list in
SELECT group_concat(nome, ",") FROM (
    SELECT nome FROM fiumi
    ORDER BY lunghezza ASC
    LIMIT 17 OFFSET (
        SELECT count(id) FROM fiumi
    ) - 17
);{list}}

\newcommand\coords{
\selectprint i, list in
SELECT group_concat(coord, " ") FROM (
SELECT '(' || lunghezza || ',' || nome || ')'
AS coord FROM fiumi
ORDER BY lunghezza DESC
LIMIT 17
);{list}}

\begin{document}
\begin{tikzpicture}
\begin{axis}[
xbar,
bar width=9pt,
width=16cm,
ymin=Ombrone,
ymax=Po,
enlarge y limits=0.035,
xmajorgrids,
```

6. Per editare il codice SQL di una query è molto più comoda la finestra di testo di SQLite Manager che il programma `sqlite3` a riga di comando con cui si lavora solamente una riga alla volta.

7. La funzione `group_concat()` di SQLite non opera quando nella query inseriamo una condizione `LIMIT`, forse perché l'ordine di concatenazione è arbitrario, come si legge nella documentazione di riferimento ufficiale.

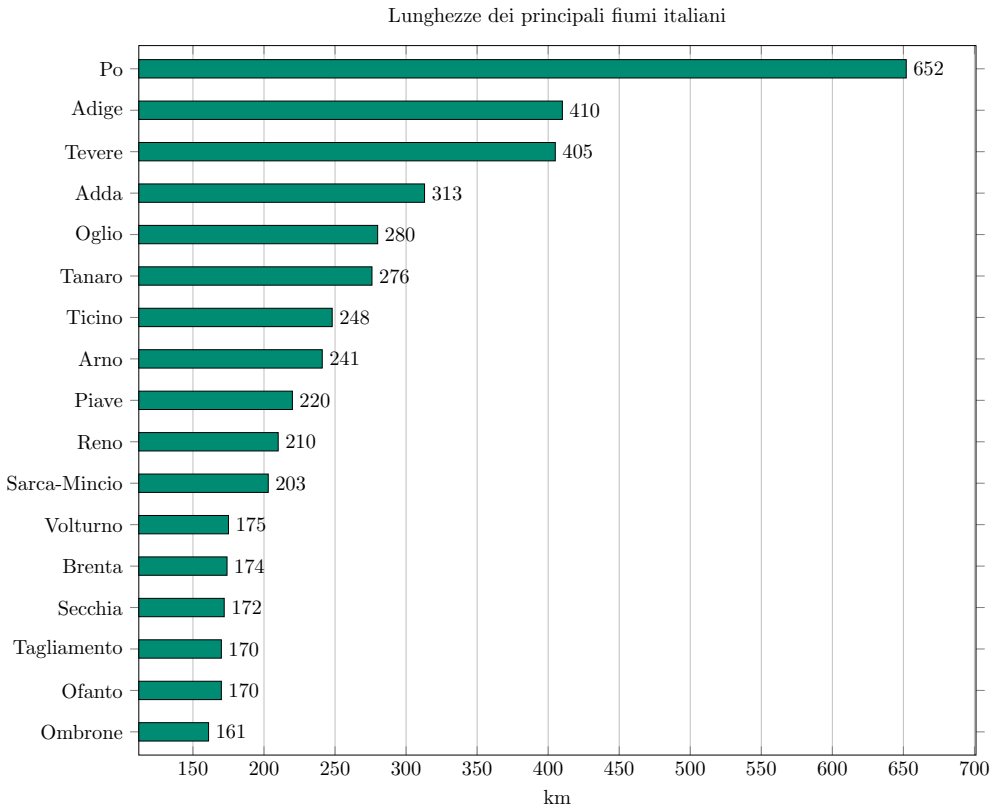

FIGURA 3: Istogramma generato con il pacchetto pgfplots con il codice riportato nel testo che ottiene i dati memorizzati in un database SQLite.

```
title={Lunghezze dei principali fiumi italiani},
xlabel={km},
symbolic y coords/.expanded={\symbcoord},
ytick=data,
nodes near coords
]
\addplot[draw=black, fill=blue!45!green]
coordinates {\coords};
\end{axis}
\end{tikzpicture}
\end{document}
```

Notate l'accortezza di marcare l'opzione `symbolic y coords` con `.expanded` per gestire l'espansione della macro che effettua la query e stampa i token della lista corretta.

Se provaste a modificare il numero n dei fiumi da vuole rappresentare nel grafico, potreste incorrere in un errore di compilazione!

Per esempio, poichè l'ordinamento è richiesto solo per il campo `lunghezza`, con i nostri dati e $n = 15$, l'ultimo fiume del gruppo potrebbe essere per la prima query, che ricava i simboli, il Tagliamento mentre per l'altra query, che restituisce le coordinate, l'Ofanto, perché entrambi di lunghezza di 170 km.

Nella lista delle coordinate avremo quindi un simbolo non dichiarato costringendo pgfplots a interrompere la compilazione.

Questo ci insegna che le query sono strumenti molto potenti ma sono, per così dire, *molto sensibili*.

Per risolvere dovremo includere anche l'ordinamento secondario per nomi. Le nuove query che risolvono il bug si differenziano solo per quest'aggiunta, la prima con ordinamento ascendente e la seconda con quello discendente perché devono produrre ordinamenti opposti, affinché in pgfplots le barre descrescano dall'alto verso il basso:

```
\newcommand\symbcoord{%
\selectprint i, list in
SELECT group_concat(nome, ",") FROM (
    SELECT nome FROM fiumi
    ORDER BY lunghezza ASC, nome ASC
    LIMIT 15 OFFSET (
    SELECT count(id) FROM fiumi) - 15);
{list}}

\def\coords{
\selectprint i, list in
SELECT group_concat(coord , " ") FROM (
SELECT '('|| lunghezza || ',' || nome ||')'
AS coord FROM fiumi
ORDER BY lunghezza DESC, nome DESC
LIMIT 15
);{list}}
```

Questo sorgente inoltre, non è del tutto generale perché da un lato richiede giustamente soltanto

di definire il numero dei fiumi da includere nel grafico, ma dall'altro costringe a passare il primo e l'ultimo fiume dell'intervallo ai parametri `ymin` e `ymax`, parametri utili a evitare che pgfplots inserisca sulla tela del grafico uno spazio bianco troppo grande sopra e sotto le barre dell'istogramma.

7.6 Miglioramenti

Non abbiamo ancora una macro adatta perché `\selectprint`, progettata per ricavare la tabella dei fiumi, presuppone una collezione di dati da stampare. Introduciamo quindi la macro `\selectrowprint` per esprimere il caso frequente di una query che restituisce una sola tupla, senza necessità alcuna d'iterazione.

La macro potrebbe essere la seguente, assegnando direttamente a variabili i campi della tupla risultato:

```
\def\selectrowprint#1<=#2;#3{%
\directlua{
local dblib = require "dblib"
#1 = dblib:rowexec([[#2]])
tex.print(#3)
}}
```

scompare il delimitatore `in` legato al ciclo generic for di Lua, sostituito da una freccia `<=` che può essere letta come *assegna da*.

La funzione `dblib:rowexec()` da aggiungere alla parte 3 della libreria di base `dblib.lua`, è molto semplice ed è la seguente:

```
function db:rowexec(query)
    local conn = self.conn
    return conn:rowexec(query)
end
```

Il grafico può quindi essere composto con questo sorgente in cui è sufficiente modificare il valore della macro `\nfiumi` per variare il numero dei fiumi da rappresentare nell'istogramma, e nell'ordine dal più lungo al più corto:

```
% !TeX program = LuaJitLaTeX

\def\dbconnect#1{...}% as before
\def\dbclose{...}

\def\selectrowprint#1<=#2;#3{%
\directlua{
local dblib = require "dblib"
#1 = dblib:rowexec([[#2]])
tex.print(#3)
}}

\documentclass[margin=2pt]{standalone}
\usepackage{luatex85}
\usepackage{pgfplots}
\pgfplotsset{compat=1.14}

\newcommand\nfiumi{20}

\newcommand\symbcoord[1]{%
\selectrowprint list <=
SELECT group_concat(nome, ",") FROM (
    SELECT nome FROM fiumi
    ORDER BY lunghezza ASC, nome ASC
    LIMIT #1 OFFSET (
    SELECT count(id) FROM fiumi) - #1);
{list}}

\newcommand\coords[1]{
\selectrowprint list <=
SELECT group_concat(coord , " ") FROM (
SELECT '('|| lunghezza || ',' || nome ||')'
AS coord FROM fiumi
ORDER BY lunghezza DESC, nome DESC
LIMIT #1
);{list}}

\dbconnect{fiumi.sqlite}

\begin{document}
\begin{tikzpicture}
\begin{axis}[xbar,
ymax/.expanded={
\selectrowprint last <=
    SELECT nome FROM fiumi
    ORDER BY lunghezza DESC
    LIMIT 1; {last}
},
ymin/.expanded={
\selectrowprint first <=
    SELECT nome FROM fiumi
    ORDER BY lunghezza ASC
    LIMIT 1 OFFSET
    SELECT count(id) FROM fiumi) - \nfiumi
;{first}},
bar width=9pt,
width=16cm,
height=11cm,
enlarge y limits=0.035,
xmajorgrids,
title={Lunghezze dei principali fiumi
    italiani},
xlabel={km},
symbolic y coords/.expanded={\symbcoord
    \nfiumi},
ytick=data,
nodes near coords,]
\addplot[draw=black,fill=blue!45!green]
   coordinates {\coords\nfiumi};
\end{axis}
\end{tikzpicture}
\dbclose
\end{document}
```

8 Chinook

In questa sezione applicativa dell'articolo metteremo in luce con un ulteriore esempio concreto la tecnologia per la connessione ai database SQLite e l'elaborazione dei dati con LuajitTEX e la libreria LJSQLite3.

Sfrutteremo le macro utente definite alla sezione 6 e in quella successiva per eseguire query e formattarne i dati risultanti, adattando il codice

alla particolare struttura del problema.

Si tratta di produrre documentazione commerciale in cui troveremo insiemi di dati ma anche campi semplici. Per questi ultimi non è conveniente eseguire una query ogni volta che è richiesto un campo. La soluzione che adotteremo sarà quella di eseguire la query una sola volta e rendere disponibili i dati attraverso oggetti Lua in memoria.

Ricordo che in osservanza della regola aurea che trovate enunciata alla sezione 5, la costruzione del database avviene con strumenti e modalità esterne al sistema TEX.

Per questa esercitazione utilizzeremo un database con dati di esempio scaricabile dalla rete Internet, il Chinook Database `https://chinookdatabase.codeplex.com/` versione 1.4 rilasciato con licenza MIT. L'archivio contiene dati di clienti, prodotti e fatturazioni, di una plausibile attività commerciale basata sulla distribuzione di musica online a pagamento.

8.1 Requisiti di progetto

Ci proponiamo di fissare all'inizio dello sviluppo dell'applicazione documentale le caratteristiche di progetto generali qui elencate:

- tutte le funzionalità Lua dovranno essere memorizzate in un'unica variabile globale di nome stabilito dall'utente per minimizzare i rischi di collisione con altri pacchetti;
- i dati saranno disponibili tramite espressioni in Lua all'interno di macro utente che non comportino la scrittura di codice SQL, indicizzando campi di tabella in dot notation, o funzioni in colon notation;
- aggiungere nuove funzioni Lua di libreria dovrà essere semplice;
- l'utente dovrà poter controllare esplicitamente la connessione al database e il caricamento dei dati;
- efficienza di esecuzione.

Alcuni di questi requisiti sono già stati soddisfatti nell'esercitazione precedente per i report sui principali fiumi italiani, altri invece no. Per essi si dovrà realizzare una nuova struttura con un pacchetto Lua di livello base e un pacchetto LATEX di livello utente.

8.2 Primo report: scheda cliente

Un report come una scheda cliente è adatto per iniziare a produrre qualche idea per il progetto di questa applicazione documentale. Organizziamo i file in modo che ogni report sia salvato in una sub-directory dedicata, all'interno di una directory di progetto dove si troverà il file del database.

Scarichiamo il file binario del database SQLite dall'indirizzo `https://chinookdatabase.codeplex.com/downloads/get/557773`[8] e, nella sottocartella nominata `schede-cliente`, prepariamo un file per LuajitLATEX.

Siamo pronti per scrivere il sorgente del report minimo che immaginiamo riguardi il cliente numero 28:

```
% !TeX program = LuajitLaTeX
\documentclass{article}
\usepackage{polyglossia}
\setmainlanguage{italian}

\usepackage[dataset=ck]{chinook}
\directlua{ck:connect()}
\directlua{ck:loadCustomer(28)}
\directlua{ck:close()}

\begin{document}
Il nome del cliente è:
\textbf{%
\print{ck.Customer.FirstName}
\print{ck.Customer.LastName}}.
\end{document}
```

Se tutto sarà corretto nel documento compilato vedremo aprendolo questo testo:

Il nome del cliente è: **Julia Barnett**.

Il pacchetto chinook caricherà la libreria base in Lua nella tabella con il nome indicato dall'utente con l'opzione `dataset`, e definirà le macro utente. La funzione Lua `loadCustomer()` eseguirà la query opportuna verso il database creando l'oggetto `Customer` all'interno sempre dell'unica tabella del dataset.

Nel corpo del documento utilizziamo la macro `\print`, già definita alla sezione 6.1, per recuperare le informazioni e stamparle nel documento. L'argomento richiesto da `\print` è un'espressione Lua valida. Nell'esempio minimo questo argomento è semplicemente il nome della variabile Lua che corrisponderà al nome e al cognome del cliente 28.

Stesso risultato per la stampa del nome lo si può ottenere anche scrivendo un unico comando `\print` tramite la concatenazione di stringhe:

```
Il nome del cliente è: \textbf{\print{
ck.Customer.FirstName .. " " ..
ck.Customer.LastName
}}.
```

Cominciamo ora a occuparci del pacchetto chinook scrivendo quindi il codice nella modalità *top-down*: prima le funzioni di alto livello e poi quelle di base che hanno a che fare con l'SQL.

Creiamo un file di nome `chinook.sty` nella subdirectory del report con il codice:

```
\ProvidesPackage{chinook}
  [2016/12/29 v0.1 User macro for dbnook]
```

8. Se accedete con un browser fate attenzione a scaricare il file compresso per SQLite e non quello preparato per uno degli altri DBMS.

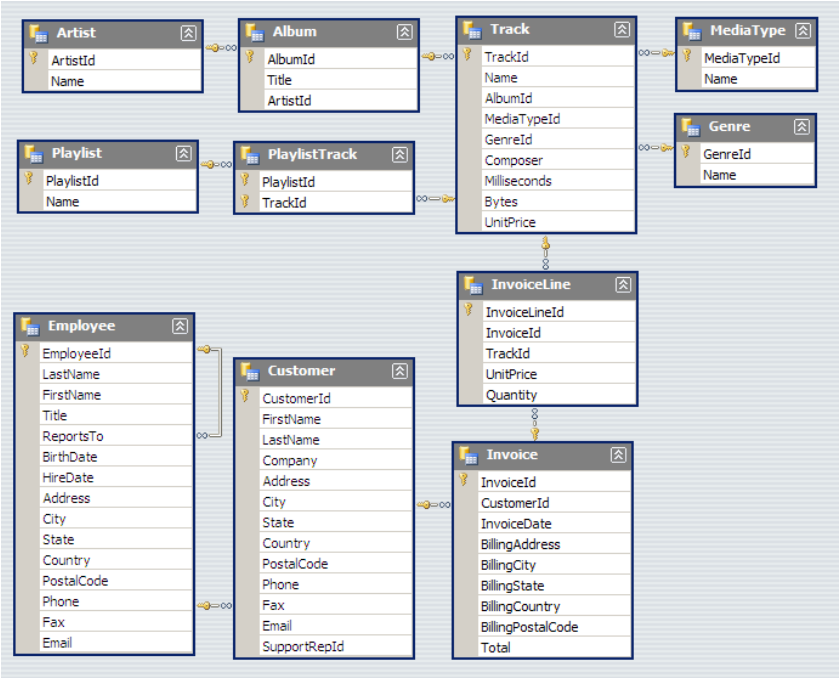

FIGURA 4: Rappresentazione grafica dello schema del database Chinook contenente dati di esempio di un'attività commerciale dimostrativa, e utilizzato nel testo come sorgente dati. L'immagine proviene dalle pagine di documentazione del progetto all'indirizzo `https://chinookdatabase.codeplex.com/wikipage?title=Chinook_Schema&referringTitle=Documentation`. L'uso dell'immagine e del database stesso sono concessi con licenza MIT.

```
\RequirePackage{kvoptions}
\SetupKeyvalOptions{
    family=dbnook,
    prefix=dbnook@
}
\DeclareStringOption{dataset}
\ProcessKeyvalOptions*
\directlua{
\dbnook@dataset = require "dbnook"
}
%
\newcommand\print[1]{%
\directlua{
local expr = #1
if not expr then
    error("Espressione non valida")
else
    tex.print(expr)
end
}}
\endinput
```

Sfruttiamo il pacchetto kvoptions per leggere nel formato chiave/valore l'opzione `dataset`. Carichiamo dunque la libreria che ho fantasiosamente chiamato `dbnook` assegnandone il riferimento al nome di variabile indicato dall'utente per espansione della macro `\dbnook@dataset` creata da kvoptions.

La macro utente `\print` è leggermente più complicata di quella incontrata nella sezione introduttiva 6.1. Infatti prima di stampare l'espressione solleva un errore se eventualmente essa valesse `nil`.

La funzione di libreria `loadCustomer()` dovrà quindi provvedere a creare la tabella `Customer` e a caricarvi l'anagrafica. Infatti nel file `dbnook.lua` troveremo il codice:

```
local sql = require "ljsqlite3"
local dbpath = "../Chinook_Sqlite.sqlite"
local tlib = {}

function tlib:connect()
    self.conn = sql.open(dbpath, "ro")
end

function tlib:close()
    self.conn:close()
    self.conn = nil
end

function tlib:loadCustomer(id)
    id = tonumber(id)
    if not id then
        error("The id ".. id ..
        " is not a valid number")
    end
    local query = string.format([[
    SELECT * FROM Customer
    WHERE CustomerId = %d;
    ]], id)
    local conn = self.conn
    local rs, nr = conn:exec(query)
    if nr ~= 1 then
        error("Customer id ".. id ..
              " is not valid")
    end
```

```
    self.Customer = {
        Id        = tonumber(rs[1][1]),
        FirstName = rs[2][1],
        LastName  = rs[3][1],
        Company   = rs[4][1],
        Address   = rs[5][1],
        City      = rs[6][1],
        State     = rs[7][1],
        Country   = rs[8][1],
    }
end
return tlib
```

La query per reperire i dati di un cliente, osservando lo schema del database riportato nella figura 4, dovrà interrogare la tabella `Customer`. In essa oltre ai campi anagrafici, notiamo la presenza della chiave esterna `SupportRepId` relativa all'impiegato responsabile del cliente, verso la tabella `Employee`.

Una query di join come questa dovrebbe essere salvata nel database come vista:

```
CREATE VIEW vw_customer AS
SELECT * FROM Customer, Employee WHERE
Customer.SupportRepId=Employee.EmployeeId
```

Qualche problema di sicurezza ci potrebbe essere nell'esecuzione della query per via dell'*SQL injection*, una tecnica usata per attaccare applicazioni di gestione dati, con la quale vengono inserite delle stringhe malevole di codice SQL all'interno di campi di input in modo che vengano eseguite. Questo codice malevolo dovrebbe essere scritto nell'opzione `Customer` del piccolo pacchetto `chinook` ma nel nostro caso il rischio è praticamente nullo perché siamo noi stessi che digitiamo da tastiera il numero del cliente.

Inoltre dobbiamo considerare che il database è in sola lettura — ricordate la regola aurea? ebbene è utile anche a questo — e che la funzione `loadCustomer()` esegue la trasformazione da stringa a numero e perciò produrrebbe un errore in caso non lo fosse.

Non aiuta invece il codice che ho scritto per la funzione `loadCustomer()` perché cade nella trappola — che abbiamo visto essere abbastanza remota — sostituendo nel codice della query l'argomento direttamente con la funzione Lua `string.format()`. Avrei dovuto invece utilizzare un *prepared statement*, una funzionalità che crea un oggetto controllato prevedendo l'operazione di binding dei parametri.

8.3 Struttura dati

La tabella Lua creata dal pacchetto chinook rappresenta il database relativo. In essa non compaiono campi e funzioni ma bensì ulteriori tabelle che raccolgono campi e funzioni. Queste tabelle rappresentano oggetti interni al database.

Nell'esempio è stato implementato l'unico oggetto `Customer` che contiene solo campi e non funzioni. Nella prossima sezione presenterò un modo per implementare funzioni formato in modo semplice come prescrive la strategia di progetto.

8.4 Funzioni formato

Per rendere semplice l'aggiunta di funzioni si può creare una tabella all'interno della libreria in Lua, che le memorizzi in relazione agli oggetti. Osserviamo il seguente codice:

```
local fnrepository = {
    Customer = {
        FullName = function (self)
            return self.FirstName ..
            " " .. self.LastName:upper()
        end,
        FullAddress = function (self, sep)
            sep = sep or "p"
            local s1, s2
            if sep == "p" then
                s1, s2 = "(", ")"
            elseif sep == "q" then
                s1, s2 = "[", "]"
            else
                error("Option not valid")
            end
            return self.Address .. ", " ..
                   self.City .. ", " ..
                   s1..self.Country..s2
        end,
    },
}
```

si tratta di una tabella contenente la chiave `Customer`, lo stesso nome dell'unico oggetto creato fino a ora, una tabella che a certe chiavi associa un paio di funzioni.

Il linguaggio è normalissimo codice Lua: impiega il costruttore di tabelle con la sintassi di definizione anonima delle funzioni. Se si desidera aggiungerne di nuove, è sufficiente inserirne la definizione per l'oggetto corrispondente considerando che `self` sarà la tabella che contiene l'oggetto stesso.

Al momento della creazione dell'oggetto, queste funzioni verranno aggiunte controllando che non si verifichino conflitti di nome tra chiavi associate a campi dati e a funzioni.

La funzione `loadCustomer()` dovrà essere modificata nella parte finale in questo modo:

```
function tlib:loadCustomer(id)
-- as before
local res = {
    Id        = tonumber(rs[1][1]),
    FirstName = rs[2][1],
    LastName  = rs[3][1],
    Company   = rs[4][1],
    Address   = rs[5][1],
    City      = rs[6][1],
    State     = rs[7][1],
    Country   = rs[8][1],
}
if fnrepository.Customer then
    for fnname, fnref in
        pairs(fnrepository.Customer) do
```

```
        if res[fnname] then
            error(
                "Internal error: "..
                "conflicted key ["..
                fnname ..
                "] for Customer"
            )
        else
            res[fnname] = fnref
        end
    end
end
self.Customer = res
end
```

Qualsiasi funzione aggiunta al repository per l'oggetto verrà automaticamente inserita nell'oggetto stesso. Perché funzioni, l'utente dovrà usare la dot notation per i campi e la colon notation per le funzioni:

```
-- field access
\print{<dataset>.<Object>.<field>}

-- function access
\print{<dataset>.<Object>:<fn>(<ars>)}
```

Il report minimo della scheda anagrafica, fatte le aggiunte di codice riportate in questa sezione nel file `dbnook.lua` potrà essere:

```
% !TeX program = LuajitLaTeX
\documentclass{article}
\usepackage{polyglossia}
\setmainlanguage{italian}

\usepackage[dataset=ck]{chinook}
\directlua{ck:connect()}
\directlua{ck:loadCustomer(28)}
\directlua{ck:close()}

\begin{document}
Il nome del cliente è:
\textbf{\print{ck.Customer:FullName()}}.

Indirizzo:
\print{ck.Customer:FullAddress("q")}.
\end{document}
```

e nel documento PDF potremo leggere:

> Il nome del cliente è: **Julia BARNETT**.
> Indirizzo: 302 S 700 E, Salt Lake City, [USA].

8.5 Documentare le funzionalità

Osservando i tre componenti per creare il report minimo, il file sorgente del report, il codice del pacchetto chinook e quello della libreria `dbnook.lua`, possiamo vedere che l'utente deve conoscere sia funzionalità del pacchetto sia quelle della libreria.

Da un lato è necessario conoscere le macro di aiuto come per esempio `\print` e dall'altro è necessario conoscere le funzioni Lua per richiamarle direttamente e i nomi dei campi.

Preparare un file di documentazione non è un lavoro accessorio, e non solo per chiarire come sono distribuite le funzionalità tra TEX e Lua. Sintetici e aggiornati fogli di riferimento consentono agli utenti di risparmiare tempo e fatica.

8.6 Esercitazioni

Un report come una fattura è piuttosto complesso. Presenta un'intestazione con il logo e i recapiti dell'azienda che la emette, una numerazione di identificazione, i dati di anagrafica del cliente e il quadro dei prodotti acquistati che riporta in basso i totali.

Lascio al lettore di realizzare il report in grado di stampare una fattura.

9 Progettazione e implementazione

In questa sezione vorrei riassumere l'esperienza che ho maturato nel creare un paio di applicazioni basate sulla centralizzazione dei dati e la creazione di report con il sistema TEX perché possano essere un utile riferimento per gli altri utenti nell'ambito di attività aziendali o professionali.

Con il termine *applicazione documentale* intenderò l'insieme dei software utilizzati per produrre documenti utili a partire da informazioni disponibili in una sorgente dati. Esempi di possibili applicazioni documentali sono la fatturazione o l'emissione di ricevute, la produzione di documentazione tecnica, la stampa di cataloghi, la stampa di moduli precompilati, la presentazione dei dati in report statistici per l'analisi, le rendicontazioni, le comunicazioni alla clientela, eccetera, nel contesto di piccole imprese di servizi o di singoli settori interni di una realtà aziendale più grande.

Quando la documentazione ha a che fare con un insieme di dati, è solitamente conveniente ricorrere a un database. Potrebbe essere già esistente all'interno dell'organizzazione oppure potrebbe essere creato a partire da database esistenti per l'utilizzo locale e legato a un'unica produzione. Oppure potrebbe essere creato da zero.

L'architettura dell'applicazione è la seguente:

1. un database management system DBMS che offre l'interfaccia SQL verso dati strutturati in uno schema di tabelle e relazioni, viste, vincoli, indici, eccetera;

2. uno o più moduli software con il quale vengono inseriti i dati nel database;

3. uno o più software di reporting compreso dei sorgenti LuaTEX o LuajitTEX.

L'utente è la persona o il gruppo di persone che usano o che desiderano cominciare a usare il sistema TEX come componente del processo documentale. L'obiettivo è quello di implementare una soluzione via via sempre più completa e user friendly acquisendo passo passo le conoscenze per la scrittura e la verifica del codice.

Lo scenario in-house non è interessante perché rende possibile progettare e implementare una soluzione tagliata su misura sui processi, ma lo è perché nel tempo è in grado di mantenere questa caratteristica più facilmente di un modulo software prodotto all'esterno.

Mentre lo svantaggio non è tanto quello di dover acquisire conoscenze informatiche di programmazione e di analisi e modellazione dati, quanto la possibilità che l'utente passi ad altro incarico cessando l'attività di sviluppo e manutenzione dell'applicazione documentale. Ciò non influisce negativamente sui progetti unici circoscritti nel tempo, o nel caso che nell'organizzazione si collabori attivamente condividendo conoscenze anche per mezzo di guide, tutorial interni o anche semplici file `readme.txt` esaurienti e ben scritti.

9.1 (Non) scelta del DBMS

SQLite ovvio! Stiamo pur sempre cominciando a riflettere sulle informazioni mettendole in ordine con qualche linea di codice e non vogliamo dedicarci a configurazioni o a sfogliare pagine di documentazione.

Avremo modo di farlo magari con un database management system come PostgreSQL magari in cloud. In altre parole conta di più raggiungere un qualche risultato in breve tempo sfornando codice SQL eseguito senza errori che realizza un primo database con una manciata di tabelle.

Un prototipo minimo ma funzionante di un report da LuajitTEX vi farà rendere conto del grado di difficoltà del codice Lua e vi darà una sguardo generale sul progetto in sviluppo.

9.2 Costruzione del database

Studiando il processo per costruirne un modello relazionale si ottiene una maggiore consapevolezza sul processo stesso. Si tratta di un lavoro che produce un quadro chiaro e univoco sugli elementi informativi dell'attività che molto probabilmente migliorerà il processo stesso.

Un secondo vantaggio deriva dalla flessibilità con cui il modello si evolve nel senso che è relativamente semplice modificarlo per tener conto di ulteriori informazioni o nuove caratteristiche del processo rappresentato.

Per la mia esperienza la costruzione del modello è in sostanza un'operazione di *scomposizione* di elementi in sottostrutture relazionate tra loro.

Ipotizziamo per esempio di elaborare un modello relazionale per questa stessa rivista con l'obiettivo di gestire il processo di ricezione, accettazione, revisione e pubblicazione degli articoli. Molto semplicemente, si inizierebbe con il rispondere ad alcune domande come: cos'è una rivista? da chi è composta e chi vi contribuisce? come si svolge il processo di produzione?

Individueremo subito la Redazione — che si compone del Direttore, del Comitato Scientifico, e dei Revisori — e gli Autori, e poi il prodotto che è la sequenza di numeri pubblicati periodicamente, ciascuno suddiviso in quattro parti: le pagine di copertina, l'editoriale del Direttore, gli articoli e gli annunci degli eventi d'interesse.

A sua volta, ciascun articolo scritto da uno o più Autori, si compone di diverse sezioni, in genere termina con una bibliografia e appare in diverse versioni, da quella inizialmente inviata alla redazione, a quella pubblicata.

9.3 Costruzione del report

La costruzione di un report comincia con lo scrivere a matita le parti che esso dovrà contenere e prosegue con lo sviluppo del sorgente per gradi, formando il codice TEX delle singole parti dapprima in modo statico e poi aggiungendo le macro che inseriranno nel documento i dati prelevati dal database.

Sarà anche naturale inserire il codice Lua necessario, in file di libreria e non all'interno dei sorgenti TEX del report. Da un lato eviteremo tutti i problemi dovuti all'espansione del testo argomento della macro `\directlua` e causati da simboli che in Lua hanno diverso significato rispetto a TEX, come il carattere percento o il backslash, e dall'altro otterremo una più importante pulizia dei listati.

Sia per il formato LATEX che per ConTEXt dovremo tener naturalmente in conto che il compositore dovrà essere LuaTEX o LuajitTEX perché solo questi programmi sono in grado di caricare per mezzo di Lua librerie esterne per la connessione dati. Per LATEX in particolare non si dovrà caricare il pacchetto inputenc perché la codifica del sorgente è obbligatoriamente Unicode utf8, mentre dovremo sostituire i pacchetti fontenc con fontspec e babel con polyglossia. Per ConTEXt dovremo usare la versione Mark IV.

9.4 Primi passi di sviluppo

Riassumendo, per creare un prototipo minimo per un'applicazione documentale in SQLite e TEX, le cose da fare sono:

1. inserire in un file di testo — chiamato per esempio `build.SQL` — il codice SQL con le definizioni delle tabelle corrispondenti agli oggetti che rappresentano l'informazione reale del processo, ed eseguire il file con il comando da console:

   ```
   $ sqlite3 nomedb.sqlite ".read build.SQL"
   ```

2. popolare il database con dati di esempio tramite un secondo file di testo contenente il codice SQL ed eseguirlo allo stesso modo del passo precedente;

3. creare un report con un sorgente per LuajitTEX che esegua una query e stampi i dati nel documento.

Successivamente si potranno scrivere programmi appositi per reperire i dati e inserirli nel database. Questi moduli sono del tutto indipendenti da quelli che realizzano i report. Per esempio, io ho fino a ora sempre utilizzato il formato *data description* di Lua per inserire i dati strutturati in semplici file di testo, e script dedicati per leggerli e salvarli nel database.

Questo formato non è altro che l'uso della sintassi del costruttore di tabelle di Lua unito alla proprietà che è possibile omettere le parentesi tonde della chiamata di funzione se l'argomento è una stringa o una tabella.

I file dati appaiono come una sorta di formato JSON quando in realtà sono una serie di chiamate a funzioni Lua con un argomento tabella. In uno script basta scrivere la definizione della funzione e caricare il file dati con la funzione `dofile()`. Informazioni dettagliate sul formato che ho chiamato data description sono reperibili nell'articolo (Giacomelli e Pignalberi, 2015) alla sezione 9.

L'interazione con i dati è solamente asincrona all'interno dell'editor di testo, ma è difficile immaginare qualcosa di più semplice e rapido da implementare.

10 Conclusioni

Come mostrano gli esempi concreti e compilabili illustrati in questo lavoro, è possibile comporre un documento TEX con dati prodotti da query verso database relazionali.

I due nuovi motori di composizione LuaTEX e LuajitTEX, essendo Lua-powered, sono in grado di eseguire il caricamento di librerie esterne in grado di istanziare interrogazioni verso i più svariati DBMS tra i quali SQLite, particolarmente adatto per sviluppare applicazioni documentali da utenti e non da sviluppatori professionisti.

Questa tecnologia da un lato produce report di elevata qualità tipografica con viste sui dati delle più svariate tipologie, grafici tabelle, figure e testo naturalmente, e dall'altro costituisce un *sistema* con cui l'utente elabora soluzioni via via più complete e sofisticate, al passo con l'evoluzione delle necessità.

La modalità di creazione del report non è basata su un template, il modello di documento, ma sulla redazione di un normale sorgente TEX in cui si ritrovano macro utente progettate per interagire *direttamente* con il database e formattarne i risultati.

Il punto di vista rispetto alla costruzione di un template cambia perché non si lavora più dall'esterno su un modello ma dall'interno sul documento stesso, unico e particolare. Vale perciò l'idea *un sorgente per ogni report* piuttosto che *un modello per più report*.

Lua si dimostra un ottimo strumento anche per programmare nuovi pacchetti specializzati nel reperire dati relazionali e per integrare moduli software scritti in altri linguaggi.

LuaTEX e LuajitTEX offrono tutta la qualità tipografica dei motori tradizionali del sistema TEX e ancor di più, con la loro attitudine verso i font Open Type, e la possibilità di programmare tutta una serie di sofisticati nuovi strumenti sui dati, particolararmente utili per le applicazioni documentali in azienda o negli studi professionali.

11 Ringraziamenti

Ringrazio Luigi Scarso per il suo lavoro su LuajitTEX e per i suoi suggerimenti sugli argomenti trattati in questo lavoro e Stefano Peluchetti per il suo modulo LJSQLite3.

Come sempre, ringrazio la redazione di ArsTEXnica per tutto il lavoro di revisione fatto sui contenuti che trovate in queste pagine.

Ultimo ma non ultimo, un ringraziamento anche al lettore.

Riferimenti bibliografici

Atzeni, P., Ceri, S., Fraternali, P., Paraboschi, S. e Torlone, R. (2014). *Basi di dati*. MC Graw Hill, 4ª edizione.

Giacomelli, R. (2016a). «LuaTEX + `pdfliteral`». *ArsTEXnica*, (22). URL `http://www.guitex.org/home/numero-22`.

— (2016b). «Primi esperimenti con node di LuaTEX». *ArsTEXnica*, (21). URL `http://www.guitex.org/home/numero-21`.

Giacomelli, R. e Pignalberi, G. (2015). «Generare documenti LATEX con diversi linguaggi di programmazione». *ArsTEXnica*, (20), pp. 40–73. URL `http://www.guitex.org/home/numero-20`.

Ierusalimschy, R. (2013). *Programming in Lua*. Lua.org, 3ª edizione.

Knuth, D. E. (1984). *The TeXbook*. Addison-Wesley Professional, 1ª edizione.

Scarso, L. (2013). «LuajitTEX». *ArsTEXnica*, (15), pp. 59–69. URL `http://www.guitex.org/home/numero-15`.

The LuaTEX development team (2016). *LuaTEX Reference Manual*, 0.95 edizione.

▷ Roberto Giacomelli
Carrara
`giaconet dot mailbox at gmail dot com`