Università degli Studi di Padova
Facoltà di Ingegneria

Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di laurea magistrale

Sviluppo di un RDBMS in Java

Laureando:
Alvise Vitturi
Matricola 1014316

Relatore:
Prof. Sergio Congiu

Correlatore:
Ing. Luca Cordioli

Anno Accademico 2012–2013

Se chiudete la porta a tutti gli errori anche la verità ne resterà fuori.

-R. Tagore

PREFAZIONE

L'obiettivo di questa tesi è stato lo sviluppo di un *RDBMS* (Relational Database Management System) presso l'azienda *MBM*, per facilitare l'utente nell'accedere alla collezione di dati risultato del *software* di pianificazione, componente del gestionale aziendale. In altre parole il *dbms* è nato con l'intento di sostituire l'interfaccia presente di accesso con quella tipica dei moderni sistemi di gestione di basi di dati, permettendo il reperimento di infomazioni con il linguaggio *SQL*. Il problema principale che si è dovuto affrontare consiste nell'interpretare i dati come tabelle e nascondere che le strutture dati non sono state realizzate tenendo presente l'eventualità di un utilizzo di questo tipo. Si è reso necessario lo sviluppo di un *driver JDBC* per permettere l'accesso con applicativi come *Squirrel* e per l'utilizzo di eventuali *framework* quali *Spring* ed *Hibernate*. È stato scelto di basare lo sviluppo nel linguaggio *Java* partendo dal *software* libero *HSQLDB*, studiando attentamente il codice e adattandolo ove possibile. Va anticipato che non è stata terminata la fase di sviluppo, questa prima versione sarà sviluppata per tutto il tempo necessario affinchè sia al livello dei *dbms* commerciali, prestando questa volta maggior attenzione all'analisi delle prestazioni. La tesi è stata composta in undici capitoli perchè si è reso necessario lo sviluppo di molti componenti, i più importanti sono:

1. **Capitolo 3**. Si tratterà la struttura portante del progetto, assieme ai componenti a cui non è stato dedicato un intero capitolo.

2. **Capitolo 4**. Verranno riportate le strutture dati scelte, senza entrare troppo in dettaglio e i principali algoritmi utilizzati nel motore del *dbms*.

3. **Capitolo 5**. Qui sarà affrontato il componente chiamato analizzatore lessicale, prima fase della compilazione.

4. **Capitolo 6**. Verrà analizzato il cuore del compilatore: l'analizzatore sintattico.

RINGRAZIAMENTI

Desidero innanzitutto ringraziare il docente Sergio Congiu per avermi concesso l'opportunità di sviluppare la tesi nell'ambito dello *stage* presso *MBM*. Inoltre, ringrazio sentitamente l' Ing. Luca Cordioli di *MBM* per il tempo e tutti i consigli dati durante i sei mesi di lavoro. Mi sento infine di ringraziare il *team* di sviluppo di *HSQLDB* sperando di poter dedicare del tempo per aiutarli nel progetto.

INDICE

ELENCO DELLE FIGURE

ELENCO DEI CODICI

1 INTRODUZIONE

> Se i costruttori costruissero come i programmatori programmano, il primo picchio che passa potrebbe distruggere l'intera civiltà.
>
> -Anonimo

1.1 PIANIFICAZIONE

In ambito aziendale è facile scontrarsi con la parola pianificazione. Consiste nell'anticipazione di una collezione di decisioni che devono essere coordinate. In altri termini si intende l'attività il cui scopo è assolvere il compito di valutare la fattibilità delle idee e trasformarle in una serie di decisioni. L'output del processo di pianificazione è chiamato piano, contenente i dati riguardanti la gestione futura. Chiaramente il piano è legato dagli obiettivi prefissati, dalle risorse diponibili e da alcuni vincoli. Gli obiettivi d'altro canto sono la ragione della pianificazione perchè rappresentano il fine da raggiungere.

1.1.1 Sistema informativo

Oggi la pianificazione necessita dell' uso di un sistema informativo come ad esempio i sistemi Enterprise Resource Planning, abbreviato ERP. Il sistema informativo aziendale è composto da più parti:

1. la componenete fisica, cioè l'hardware;
2. i dati e le informazioni;
3. le persone atte a catalogare e raccogliere i dati;
4. il software di gestione;

Non va assolutamente confuso il sistema informativo con il sistema informatico che è una sua componenete.

1.1.2 Vincoli

Per la pianificazione si parte sempre dall'analisi dei vincoli interni ed esterni all'azienda. Non sarebbe intelligente la creazione di

Interni	condizioni economiche condizione dei fattori produttivi da impiegare condizioni del mercato
Esterni	esistenza delle teconologie facilità del reperimento dei fattori produttivi disponibilità della forza lavoro possibilià di accedere alla liquidità necessaria

Figura 1.: Vincoli nella pianificazione.

un progetto evitando di considerare le variabili ambientali, delle risorse disponibili e quanto altro dell'azienda. Esempi di vincoli vedi 1. Chiaramente i vincoli all'interno di tutto il complesso del sistema informativo sono espressi mediante il formalismo della matematica. Spesso sono rappresentati per mezzo di equazioni, disequazioni, etc. Per un'introduzione sulla ricerca operativa, utile alla pianificazione, si consiglia di leggere un libro come [12].

1.1.3 Software di pianificazione

Entriamo più in dettaglio su questi concetti, dato che quanto detto rappresenta solo la punta dell' *iceberg* di cosa sia, a livello aziendale, la pianificazione. Ad esempio, il *software* di pianficazione integrato nel gestionale *ERP* di un'azienda è composto da più componenti, anzi, sono da considerarsi proprio diversi tipi di pianificazione, sempre più completi. Si possono individuare, prendendo come esempio *GPS* (Global Planning System) di *MBM* i seguenti componenti:

1. **MRP**. Material Requirements Planning;
2. **PRM**. Pegging Resolution Management;
3. **RAW**. Raw Capacity Planning (Macrovincoli);
4. **CRP**. Capacity Requirements Planning;
5. **FCP**. Finite Capacity Planning;
6. **ESS**. Enterprise Scheduling System;

Entriamo più nel dettaglio di *MRP*, *CRP* e dello schedulatore.

MRP

Il **Material Requirements Planning** (pianificazione dei fabbisogni di materiali) è lo strumento che serve, in genere, a rispondere alle seguenti domande:

1. Che cosa produrre e acquistare;

2. Quanto produrre e acquistare;

3. Quando produrre e acquistare;

È costituito in sostanza da un algoritmo che riceve come *input* le previsoni di vendita, la distinta base, le scorte presenti e restituisce come *output* gli ordini di produzione e di acquisto. In altre parole trasforma i fabbisogni dei prodotti finiti nei fabbisogni delle materie prime e dei componenti necessari alla loro realizzazione. I due problemi che affliggono questa prima pianifiazione sono:

1. opera con capacità infinita. Non si pone cioè problemi che riguardano le capacità delle macchine e ne nel numero di ore di lavoro disponibili;

2. considera il *lead time* costante;

CRP

L'algoritmo *MRP* ha il difetto di considerare la capacità infinita, da questo è nato il *CRP* **(Capacity Requirements Planning)**, algoritmo che riceve come *input* l'*output* di *MRP* mettendo in relazione il piano precedente con i tempi di produzione, estratti dai cicli di produzione. Realizzando come *output* (piano del fabbisogno delle risorse) il calcolo del fabbisogno di ore lavoro per ogni reparto, macchina, etc. permettendo così di verificare la disponibilità effettiva del tempo necessario alla produzione.

Scheduler

Il problema che affronta lo *scheduler* è: eseguire una schedulazione dei reparti produttivi in grado di ottimizzare l'impiego delle risorse e l'utilizzo dei materiali. Dentro *GPS* tale questione è risolta da *ESS* parte di una *suite* di applicazioni *software* che fanno uso dei motori di ottimizzazione matematica *IBM ILOG*.

1.2 COS'È UN *dbms*?

Un sistema di gestione di basi di dati (*DBMS*, *database management system*) è un insieme di programmi che permettono agli utenti di creare e mantenere una base di dati. Il *DBMS* ha pertanto scopi generali e facilita il processo di definizione, costruzione e, condivisione delle basi di dati per una moltitudine di applicazioni. Un programma applicativo (Squirrel) accede alla base di dati inviando delle interrogazioni, in inglese *query*, o delle richieste dati al *DBMS*.

1.2.1 Natura autodescrittiva

Il *dbms* non contiene solo la base di dati, ma anche una descrizione completa della sua struttura e dei vincoli sui dati presenti. Queste informazioni sono memorizzate nel catalogo del sistema e vengono riferite con il nome di metadati. I metadati sono usati dal *dbms* e dagli utenti della base di dati che necessitano di informazioni sulla struttura della stessa.

1.2.2 I linguaggi dei *dbms*

Concluso il progetto di una base di dati, scelto il *dbms* per implementarla, il principale obbiettivo diventa quello di specificare gli schemi concettuale e interno. Per essere precisi il *dbms* dovrebbe fornire tre linguaggi differenti:

1. ***DDL***, o *data definition language*. La sua funzione è quella di agire sullo schema del database;
2. ***VDL***, o *view definition language*. Serve per specificare le viste dell'utente e il legame con lo schema logico.
3. ***DML***, o *data manipulation language*. Usato per l'inserimento, reperimento, modifica e cancellazione dei dati.

Invece i *dbms* tradizionali usano un unico linguaggio *SQL* (vedi C) che li racchiude tutti.

1.2.3 Il modello relazionale

Il modello relazionale è stato creato nel 1970 da *Ted Codd* un ricercatore dell'*IBM*. Tale modello usa il concetto di relazione matematica (semplificando: una tabella di valori) come suo componente elementare e ha il suo fondamento nella teoria degli insiemi e nella logica dei predicati del primo ordine. Le basi di dati relazionali sono state proposte per separare la memorizzazione fisica dei dati dalla loro rappresentazione concettuale e per poggiare su basi matematiche più solide. Tale modello ha permesso l'introduzione di alcuni linguaggi di interrogazione di alto livello, ormai assodata alternativa ai linguaggi di programmazione essendo più facile la stesura di nuove interrogazioni. Viene spiegato più dettagliatamente in B.

1.2.4 Il *DBMS* relazionale l'*RDBMS*

Non vi è una definizione precisa di quali caratteristiche minime debbano essere presenti in un *Relational Database Management System*, credo sia corretto dire: ogni *DBMS* dove i dati vengono codificati per mezzo di tabelle (righe e colonne) si definisce relazionale.

1.2.5 Architetture *client/server* a due livelli

Nei *dbms* relazionali si è venuto a creare un punto di separazione fondamentale tra *client* e *server*. Soltanto le funzioni transazionali e di interrogazione sono al giorno d'oggi presenti sul *server*. In tale architettura le interfacce utente e le applicazioni possono essere spostate nella loro interezza lato client. Per l'accesso al *dbms* è richiesta una connessione usata per l'invio delle richieste e per i risultati delle interrogazioni. Lo standard, più generale possibile è l'***ODBC***, fornisce un ***API*** (application programming interface) che consente agli applicativi di chiamare il *dbms*. È stato definito più recentemente uno standard per il linguaggio di programmazione *Java*, chiamato *JDBC* (vedi 8). Questa architettura viene chiamata a due livelli, perchè le componenti sono distribuite su due sistemi: *client* e *server*.

1.2.6 Il modello ad ogetti e gli OODBS

Il modello *ER* mostra dei limiti nei confronti di alcuni aspetti molto importanti della realtà che si intende modellare. In particolare esso non consente di associare operazioni alle entità: questo aspetto viene trattato separatemente e disperso nei programmi applicativi che verranno poi utilizzati. L'idea di unire la *OOP* con le basi di dati, ha avuto un rapido sviluppo solo a partire dagli anni '90 quando sono apparsi i primi esempi di *Object Oriented Database Management System*. Questi sistemi non gestiscono basi di dati nel senso tradizionale ma -appunto- basi di dati orientate agli oggetti. Da tempo esistono molti prodotti con tale classificazione, ma con caratteristiche troppo diverse da precludere anche dei confronti. Si sente tutt'oggi la necessità di disporre di un modello concettuale indipendente dall'*OODBMS* che svolga la funzione del modello *ER*.

1.3 SCELTA SVILUPPO DEL DBMS NEL LINGUAGGIO JAVA

La motivazione principale che ha portato alla scelta dello sviluppo di un un dbms è quella di poter estrapolare i dati dal software di pianificazione con la semplicità tipica di un linguaggio dichiarativo come SQL. Il progetto è nato pertanto con l'intento di aumentare le possibilità di recupero dei dati già presenti nel codice, limitate però al solo utilizzo dei metodi degli oggetti *Java* implementati. È stato naturale far cadere la scelta sul lingauggio appena menzionato, a causa non solo degli indubbi vantaggi che fornisce, ma anche per il fatto che la parte del software di MBM™ con cui si ha la necessità di interfacciarsi è implementato in Java. Si è deciso inoltre di sviluppare parallelamente un driver JDBC di tipo quattro per l'accesso al

dbms, così da poter sfruttare l'enorme quantità di utility disponibili in commercio come Squirrel.

1.4 FUNZIONALITÀ DA IMPLEMENTARE

Prima di iniziare lo sviluppo di un'applicazione è necessario decidere quali funzionalità e caratteristiche dovranno essere presenti e tenere a mente le necessità di sviluppi futuri, atti ad ampliare le *feature* dello stesso. Bisogna inoltre cercare di riutilizzare per quanto possibile il codice già pronto presente in altri software liberi, prendendo anche spunto per qualche idea implementativa. Infine è utile considerare il tempo a disposizione ed eventualmente decidere di posticipare lo sviluppo di alcuni componenti. Si è scelto inizialmente di dotare il dbms delle seguenti caratteristiche:

1. eseguire in forma semplificata le interrogazioni fondamentali in SQL;
2. gestione delle sessioni;
3. gestione degli utenti;
4. gestione dei cataloghi e schemi secondo lo standard;
5. gestione delle eccezioni rispettando lo standard;
6. driver JDBC di tipo quattro;

Si è infine deciso, per la natura dei dati, di mantenere in memoria *RAM* ogni informazione (tabelle, viste, schemi etc.) contenuta nel *dbms*. Quindi non è stata pensata alcuna funzionalità o componente per la rappresentazione delle tabelle nella memoria di massa.

Ovviamente sono stati presi in considerazione altri punti, ad esempio la concorrenza e l'aggiunta di un DDL. Sono elencati più dettagliatamente in 11.

2 STRUMENTI DI SVILUPPO

> La filosofia è scritta in questo grandissimo libro che continuamente ci sta aperto innanzi agli occhi (io dico l'universo), ma non si può intendere se prima non s'impara a intender la lingua, e conoscer i caratteri ne' quali è scritto. Egli è scritto in lingua matematica, e i caratteri son triangoli, cerchi, ed altre figure geometriche, senza i quali mezzi è impossibile intenderne umanamente parola; senza questi è un aggirarsi vanamente per un oscuro laberinto.
>
> -Galileo Galilei

Questo capitolo introduce brevemente gli strumenti adottati nel corso dello sviluppo del DBMS.

2.1 ECLIPSE

Eclipse è l'editor di testo più usato in Java, inizialmente sviluppato da IBM e successivamente rilasciato come software libero. Comprende nella sua variante più completa, un insieme di strumenti molto avanzati quali:

1. **SVN**;
2. **CVS**;
3. **TOMCAT**;
4. **DEBUGGER**;
5. **ANT**;

Il suo punto di forza è di poter essere eseguito in qualsiasi computer che possieda una Java Virtual Machine, proprio perchè sviluppato in Java.

2.2 CLIENT SQUIRREL

Squirrel è un programma appartenente alla grande famiglia dei client SQL sviluppati per interagire con qualsiasi dbms che fornisca un driver JDBC. Enfatizza le caratteristiche comuni a ogni database. Fornisce inoltre una collezione di plugin molto vasta permettendo tuttavia di scriverne di propri, in modo da utilizare funzionalità specifiche di un singolo dbms. Ecco come appare 2

2.3 JAVA RMI

Una spiegazione dettagliata è presente in 7.

2.4 SPRING

Spring (vedi [9]) è un framework open source nato con l'intento di gestire la complessità dello sviluppo di applicazioni enterprise. I suoi punti di forza principali sono quattro:

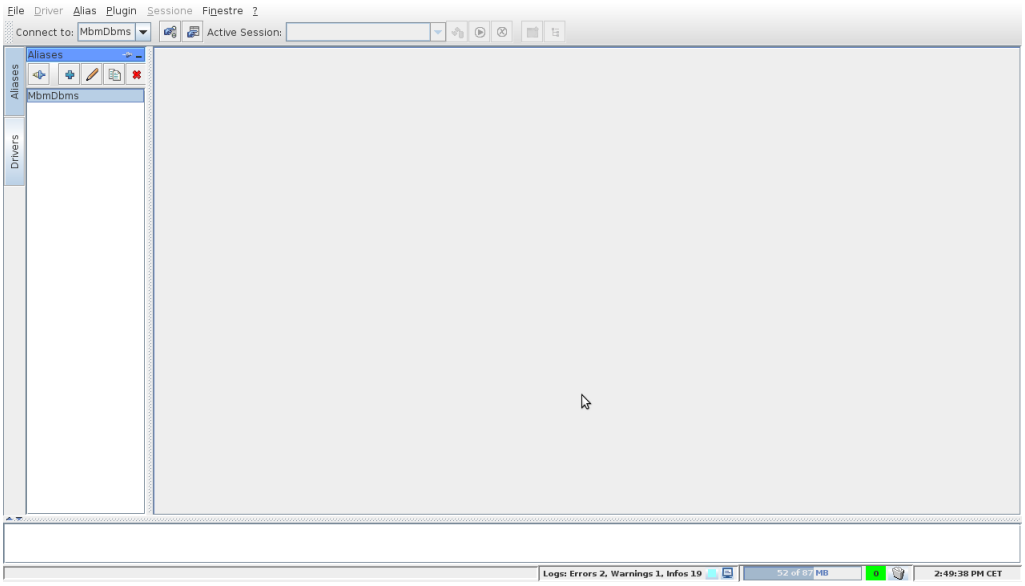
1. **Spring è un framework leggero**. In sostanza può essere integrato nel progetto per passi senza essere costretti a riscrivere tutto da capo;
2. **È un lightweight container**. propone un modello più semplice e leggero per lo sviluppo di entità *businnes*;
3. **Fornisce una serie completa di strumenti**;
4. **Test codice**. In Spring l'idea fondamentale è che il codice di qualità debba essere semplicemente testato;

2.4.1 Moduli di Spring

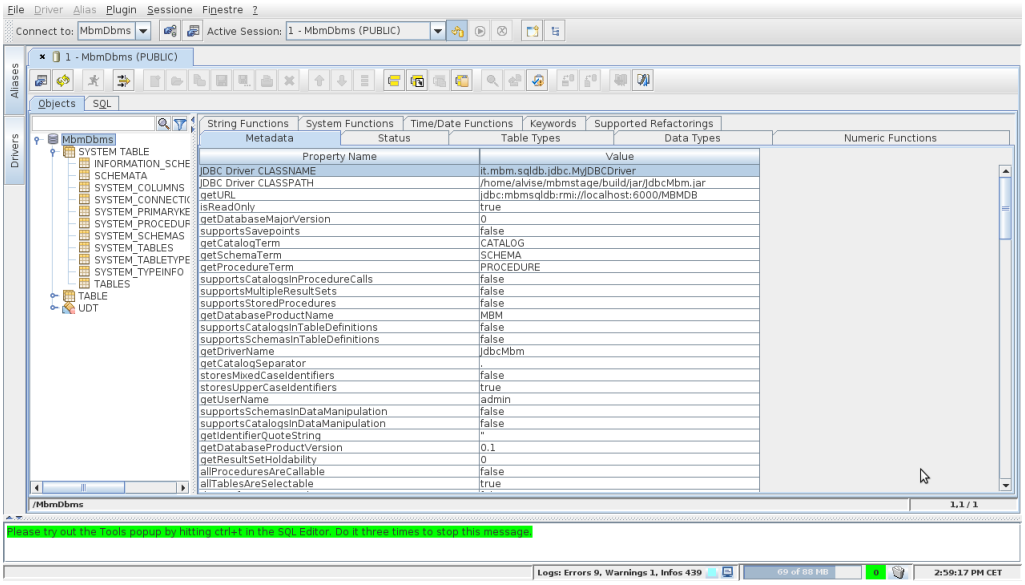
Spring consiste di una ventina di moduli come nella figura 3. Per semplificare, di molto, le cose, ogni modulo può essere pensato come una libreria per il linguaggio *Java* che aggiunge, sostituisce o modifica quanto già presente nel linguaggio.

2.4.2 Spring Inversion of Control

Per comprendere meglio il framework in questione bisogna descrivere i concetti di Inversion of control (IoC) e Dependency Injection (*DI*) . Nella programmazione classica il flusso del programma è esplicitamente definito dal programmatore come ad esempio le operazioni di creazione inizializzazione ed invocazione di metodi. In Spring



(a) *Squirrel client avviato.*



(b) *Squirrel client connesso.*

Figura 2.: Squirrel

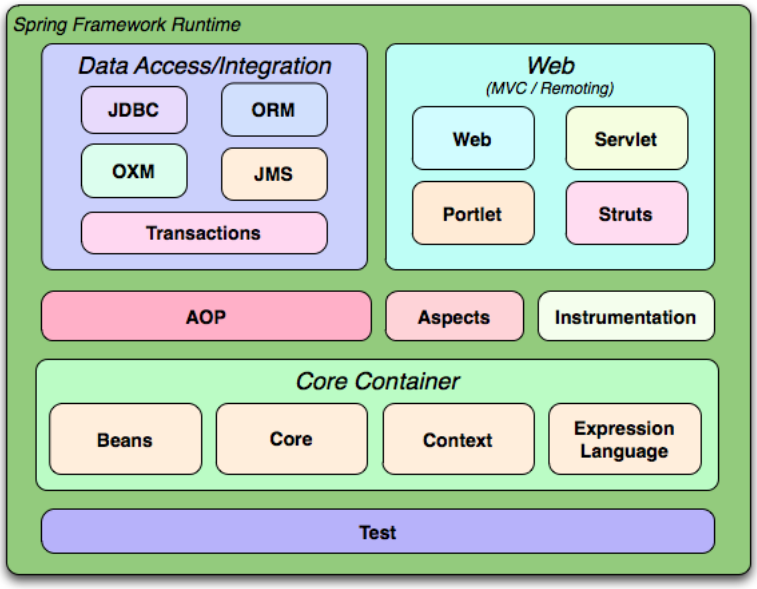


Figura 3.: Moduli Spring

l'*inversion of control* cerca, come dice il nome, di cambiare questi aspetti in modo che sia il framework stesso a preoccuparsene. *DI* invece non è un concetto separato ma proprio un esempio di IoC che riguarda le dipendenze. Nel caso della oop una classe A si dice dipendente dalla classe B se usa i suoi servizi, normalmente è compito del programmatore realizzare la logica per risolvere tale dipendenza, Spring ha l'ambizione di risolverla autonomamente (esempio di *DI*).

2.4.3 Spring AOP

La programmazione orientata agli aspetti (AOP) aiuta quella orienta agli oggetti (OOP) fornendo un altro modo di pensare alla struttura di un programma. Nel progetto del dbms viene usata per il logging. Più avanti sarà sfruttata anche per le transazioni. La comprensione di AOP passa per alcuni concetti fondamentali:

1. **Join Point**;
2. **Ascpect**;
3. **Croscutting Concern**;
4. **Advice**;
5. **Pointcut**;
6. **Target**;

Assieme all'IoC rappresenta uno dei motivi principali della popolarità di Spring.

2.4.4 Spring Test

Un punto molto importante del framework Spring è quello del testing. Tale framework fornisce una serie di strumenti utili ad implementare e automatizzare i test. Nel progetto, come si vedrà, questa parte è sfruttata per tutto quello che concerne il driver JDBC, proprio a causa dell'alto numero di metodi da implementare.

2.4.5 Spring WEB/MVC

Model-View-Controller (MVC) è il nome di una pratica informatica cui conviene utilizzare nelle applicazioni ogniqualvolta interagiscono con l'utente per mezzo di un'interfaccia grafica. Tale nuova architettura, infatti, incentiva la separazione dei ruoli e la suddivisone del codice. Con questo approccio si ottiene del software migliore, frutto di un codice più robusto più facile da evolvere e specialmente da mantenere. Spesso per implementare i pattern MVC si ricorre ad un framework esterno, in grado di potenziare la piattaforma di partenza. Questo è proprio il caso di Spring, dotato di una serie di classi, di supporto al programmatore. Prima di collegare il *DBMS* con *Spring* ho realizzato un piccolo applcativo *web* basato su tale *design pattern*.

2.5 HIBERNATE

Da Wikipedia: "*Hibernate* (spesso chiamato H8) è una soluzione *ORM* (*Object Relational Mapping*)" per il linguaggio di programmazione *Java*. È un *software free*, *open source* e distribuito sotto licenza *LGPL* (permette di utilizare la libreria dentro un programma chiuso). *Hibernate* fornisce un *framework* semplice da usare, che mappa un modello di dominio orientato agli oggetti in un classico database relazionale. In realtà non si occupa solo di mappare classi *Java* in tabelle, ma fornisce anche degli aiuti per le query di dati, il recupero di informazioni e riduce sicuramente il tempo da dedicare per lavorare manualmente in *SQL* e con il *driver JDBC* (vedi [16]).

2.6 LIBRERIE

In questa sezione saranno spiegate brevemente le librerie Java utilizzate per lo sviluppo.

Codice 1: Esempio uso libreria BeanUtils.

```
Object object = objectClazz.newInstance();
for (PropertyDescriptor propDesc :
         PropertyUtils.getPropertyDescriptors(object)) {
        String propName = propDesc.getName();
        System.out.println(propName);
}
```

2.6.1 BeanUtils

La libreria BeanUtils è un jar che permette di accedere dinamicamente alle propietà degli oggetti Java. Nel codice del dbms viene utilizzata ogniqualvolta sia necessario caricare una tabella dal software di pianificazione GPS. Nella figura 1 un piccolo esempio per capire il suo funzionamento.

2.6.2 Javassist

Questo strumento assieme alla reflection (vedi [6]), tecnica ormai consolidata in Java, è in grado di definire nuove classi dinamicamente. In realtà non si limita solo a questo ma permette di modificarle al volo aggiungendo metodi, campi e interfacce implementate. Tale libreria viene usata nel codice del dbms assieme a BeanUtils per definire nuovi tipi di tabelle in questi casi specifici:

1. **SELECT**;
2. **CREATE TABLE** (Quando verrà implementato);
3. **DROP** (Quando verrà implementato);

2.6.3 Log4J

Log4J (vedi [15]) è una libreria Java sviluppata dalla Apache Software Foundation che permette di sviluppare un sistema di logging evoluto. Il modo classico di configurare correttamente tale libreria è di scrivere un file di properties che, ovviamente, può essere scritto in formato xml. Il file è costituito da due componenti:

1. **Logger**;
2. **Appender**;

A ciascun Logger viene associato un livello di log:

1. **DEBUG**;
2. **INFO**;

3. **WARN**;
4. **ERROR**;
5. **FATAL**;

Gli Appender invece sono un flusso di output. Log4J è dotato di diversi Appender i cui nomi sono autoesplicativi:

1. **Console Appender**. L'appender di default;
2. **File Appender**;
3. **RollingFileAppender**;
4. **DailyRollingFileAppender**;
5. **Socket Appender**;
6. **JMS Appender**;
7. **SMTP Appender**;

Bisogna considerare che ogni Appender necessità di essere configurato con alcuni parametri specifici. Ad esempio a ciascuno di loro è possibile associare un Layout.

2.7 LINGUAGGIO JAVA

Abbiamo anticipato la decisione di scegliere *Java* come linguaggio per tutto lo sviluppo del *dbms*. Spesso sono state mosse delle critiche pesanti a causa della sua lentezza. *Java*, chiaramente, non può competere con linguaggi di più basso livello come l'*assembly* o il *C* ma da questo punto di vista, negli anni, si è trasformato portando sostanziali migliorie da una verione all'altra. I motivi di questo cambiamento sono alcuni componenti, per così dire nascosti come:

1. **JIT**. compilazione *Just In Time*;
2. **HS**. *Hot Spot*;
3. **GC**. *Garbage Collector*;

2.7.1 Java Hot Spot

Questo componente individua i punti più critici del codice per poterli ottimizzare e rendere, così, il processo di traduzione (*bytecode*) ed esecuzione il più veloce possibile. Come riesca la *JVM* ad individuare tali punti (caldi) è compito degli algoritmi adattivi di intelligenza artificiale.

2.7.2 Compilazione JIT

La compilazione *Just In Time* effettua un lavoro di ottimizzazione per eseguire un pezzo di codice compilandolo solo quando veramente necessario.

2.7.3 Il Garbage Collector

Come dice il nome effetua proprio un lavoro di pulizia, un'altra attività automatica di *Java*, il cui compito è verificare se un'istanza di una classe sia ancora necessaria. In linguaggi come il *C* deve essere il programmatore a liberare la memoria allocata nello *heap* quando non più utilizzate, altrimenti sono dietro l'angolo i cosidetti *memory leak*.

2.7.4 Javadoc documentazione

Lo strumento *Javadoc* integrato in tutte le installazioni dell'ambiente *Java* permette di generare della documentazione in formato *HTML* delle classi sviluppate, sul modello della documentazione dello standard *Java*. Si può generare la documentazione solo per quelle classi dichiarate *public*. Tra i vari strumenti è l'unico che non è stato utilizzato in maniera corretta, per ora la documentazione è scarna e deficitaria. Andrà decisamente usato meglio per costruire una documentazione che aiuti chiunque sia, per qualsiasi motivo, costretto a manipolare il codice sorgente.

3 PROGETTO

> Ma in mente mi vien, così alla grossa, che questo sia presagio di chissà quale turbamento per il nostro stato.
>
> -Shakespeare

Si è scelto di basare lo sviluppo su quanto già fatto dal *software* libero *HSQLDB*; un noto dbms scritto interamente in Java utilizzato ad esempio da *Open Office*. Appoggiarsi a un simile progetto ha reso possibile un' estrema velocizzazione (nella stesura del codice) e di ottenere già in anticipo un *lower bound*, sufficientemente preciso, del tempo neccessario per qualcosa di funzionante, che ponesse le basi ad ulteriori sviluppi. Ha permesso infine di comprendere alcune soluzioni da adottare per i problemi comuni che inevitabilmente sorgono nella costruzione di un dbms.

3.1 LA SRUTTURA

Lo sviluppo di un' applicazione di medio-grandi dimensioni deve essere strutturata in *Java* con l'uso dei *package*. Risulta chiaro che è umanamente impossibile pensare di riunire tutto il codice in un unico file, per quanto in realtà il linguaggio consenta di farlo.

3.1.1 I package di Java

Un package è una collezione di classi e interfacce correlate che fornisce uno spazio dei nomi ed un controllo sugli accessi. Aiutano ad organizzare le classi in una struttura ad albero e rendono facile la loro localizzazione. Il linguaggio è dotato di una vastissima libreria già organizzata con l'uso di questo strumento. Alcuni esempi:

1. *java.lang*;
2. *java.util*;
3. *java.io*;
4. *java.math*;

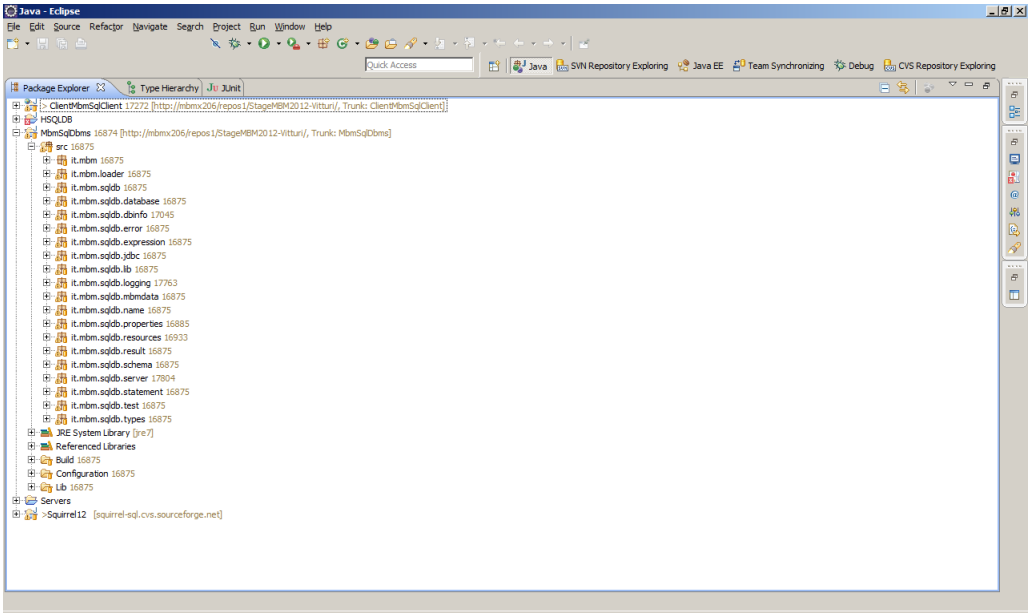


Figura 4.: Package progetto.

5. *java.sql*;
6. *java.security*;

3.1.2 Organizzazione in package

Un'applicativo *software* è costituito da uno svariato numero di componenti che devono interagire tra loro. Si è deciso di creare un *package* apposito per uno specifico componente nel caso fosse costituito da un collezione di classi. Pertanto è possibile che alcune parti si trovino in uno stesso package anche se sono, in realtà, separate, questo per evitare una proliferazione eccessiva degli stessi. Riportiamo qui 4 l'insieme dei package che fotografa la struttura di quanto sviluppato.

3.1.3 Le componenti

Decrivere dettagliatamente ogni componente del dbms non è possibile, diamo innanzitutto un elenco delle sue componenti principali:

1. **Gestore delle sessioni**;
2. **Gestore dei nomi**;
3. **Gestore dei *database***;
4. **Gestore delle risorse**;
5. ***Factory* delle eccezzioni** 9;
6. **Analizzatore lessicale** 5;

7. **Parser** 6;
8. ***Driver JDBC* tipo quattro** 8;
9. ***Server* per caricamento classi a *runtime*** 7;
10. **La componente di test** 10;

Sono presenti anche tutta una serie di classi usate internamente per codificarvi importanti informazioni. Ad esempio:

1. ***DatabaseInformation***. Usata per memorizzarvi le tabelle aggiuntive di sistema descritte in C;
2. ***MBMDataRecords***. La classe per le tabelle del dbms, la descrizione di come sono strutturate in 4;
3. ***SchemaTable***. Per istanziare gli oggetti che rappresentano gli schemi delle tabelle;
4. ***Expression***. Usata per le espressioni nelle condizioni di *where* e *having* vedi 4;
5. ***DataType***. Ogni oggetto contiene informazioni di un particolare tipo di dato per il linguaggio *SQL*;
6. **RequestResponse**. Ogni istanza contiene le informazioni di una richiesta o una risposta tra il *driver jdbc* e l'applicativo.

Nel proseguire di questo capitolo sono trattati, in modo semplificato, i componenti a cui non è dedicato un capitolo a sè stante.

3.2 I GESTORI

Definire all'interno dell'applicazione più gestori aiuta lo sviluppatore per svariati motivi. Migliora la:

1. leggibilità del codice;
2. manutentabilità;
3. divisione dei compiti;

3.2.1 Il gestore dei database

Il più importante ed imprescindibile è sicuramente quello per la gestione delle basi di dati presenti nel dbms. Possiamo riassumere le sue funzioni nelle seguenti:

1. aggiunta di un database nel sistema con relativa creazione di un nome e un identificativo;

Codice 2: Interfaccia DatabaseManager.

```
public  interface DatabaseManager {
        Database getDatabase(Integer idDataBase){};
        Database getDatabase(String databaseName){};
        Database getDatabaseOfIdSession(Integer idSession){};
        void addDatabase(Database database, String databaseName){};
}
```

Codice 3: Interfaccia SessionManager.

```
public interface SessionManager {
        synchronized Session newSession(Database db){};
        Session getSession(Integer idSession){};
        synchronized Session[] getAllSessions(){};
        void closeAllSessions(){};
        Session getSysSession(){};
        void addSession(Session session){};
        void removeSession(Integer idSession){};
}
```

2. metodi per la restituzione dei database presenti;

Una possibile interfaccia semplificata vedi 2.

3.2.2 Il gestore delle sessioni

Per permettere che più client possano collegarsi al *dbms*, è di notevole utilità l'introduzione, nel codice, di un gestore per le sessioni. Ogniqualvota, appunto, si apre una connessione deve esserle associata una sessione dove mantenere un lista di dati temporanei. La classe del gestore potrebbe ad esempio implementare l'interfaccia 3.

3.2.3 Il gestore dei nomi

Data la grande quantità di oggetti di cui si ha la necessità di associare un nome, viene spontaneo dotare il *dbms* di un tale gestore. Tabelle, viste, e altro ancora sono solo alcune entità che possiedono un nome e, il compito del gestore è quello di assegnarne uno a ciascuno di loro, prestando attenzione alle collisioni. Qui 4 l'interfaccia di un possibile gestore.

Nel progetto possono essere istanziati più gestori, uno per ogni catalogo. Di default si usa quello associato al catalogo *PUBLIC*, istanziato staticamente all'interno della classe. I cataloghi vanno pensati in modo del tutto analogo ai *namespace* di C++. Due tabelle possono avere lo stesso nome in due cataloghi differenti.

Codice 4: Interfaccia NameManager.

```
public interface NameManager{
  MbmsqlName newMbmsqlName(String name, MbmsqlName parent,
                           MbmsqlName schemaName, int type){};
  MbmsqlName newMbmsqlName(MbmsqlName schema, String name,int type){};
  MbmsqlName getCatalogName(){};
  MbmsqlName newInfoSchemaTableName(String name){};
  MbmsqlName newInfoSchemaObjectName(String name, int type){};
  MbmsqlName newInfoSchemaColumnName(String name, MbmsqlName table){};
}
```

3.2.4 Il gestore delle risorse

Il gestore delle risorse ha lo scopo di localizzare e restituire una risorsa che sia stata richiesta dal dbms. La natura delle risorse è varia; potrebbe trattarsi anche di un' informazione codificata all'interno di un file, quali ad esempio, i messaggi di errore per una specifica eccezione. La classe, ridotta all'osso, usata nel dbms sviluppato in 5.

3.3 NECESSITÀ DI CARICARE ALTRE CLASSI

Sorge spontaneo chiedersi la motivazione che ha portato allo sviluppo di un *Server* di caricamento classi. Qui possiamo solo anticipare che è conseguenza della gestione interna dei dati fatta dal *dbms*, costretto a creare nuovi classi durante la sua esecuzione. D'altro canto il driver *jdbc* (vedi 8) non possiede la definizione di tali classi e *Java* richiede pertanto un meccanismo che possa fornirgliele. Il componente *ClassServer* serve semplicemente a questo. Rimandiamo al capitolo 4 per le strutture dati, e al 7 per una descrizione di un *ClassServer*.

3.4 COMUNICAZIONE QUESTA SCONOSCIUTA

Una domanda ancora lasciata in sospeso è: come avvengono le connessioni? Per rispondere a questo, conviene anticipare quali alternative si sono presentate nello sviluppo:

1. sfruttare *Java RMI*;
2. sfruttare *CORBA*;
3. utilizzare i più famosi protocolli di comunicazione come *http* o *ftp*;
4. progettare un protocollo personale;

Ognuna di esse presenta vantaggi come svantaggi, diverse difficoltà nello sviluppo e prestazioni diverse. Nel progetto è stato deciso di tralasciare un'analisi dettagliata e approfondita sull'argomento e lanciarsi nell'uso di *Java RMI*, perchè già usato da altri applicativi all'interno dell'azienda (MBM).

3.4.1 Le richieste e risposte

Una volta deciso lo strumento bisogna decidere come sfruttarlo. Lo strumento *Java RMI* si presta bene allo scopo, considerando che sia le richieste che le risposte, possono essere inserite all'interno degli argomenti e dei risultati dei metodi remoti. Mostriamo qui 6 l'interfaccia dell'oggetto gestore di tutte le richieste fatte al *dbms* da parte del *driver jdbc*. Nella classe *RequestResponse* vi sono sempre tutte le informazioni per l'uno o per l'altro tipo di pacchetti. È valido, in sostanza, per entrambi i tipi.

Codice 6: Interfaccia del gestore delle richieste.

```
public interface Executor extends Remote {
        public RequestResponse execute(RequestResponse request)
                                        throws RemoteException;
        public RequestResponse connect(RequestResponse request)
                                        throws RemoteException;
        public void disconnect(RequestResponse request)
                                    throws RemoteException;
}
```

Codice 5: Classe del gestore risorse.

```
public abstract Class BundleHandler {
  public static Locale getLocale() {
    synchronized (mutex) {
      return locale;
    }
  }
  public static void setLocale(Locale l)
               throws IllegalArgumentException {
    synchronized (mutex) {
      locale = l;
    }
  }
  private static Method getNewGetBundleMethod() {
    //Manca implementazione
  }
  public static int getBundleHandle(String name, ClassLoader cl) {
    Integer bundleHandle;
    ResourceBundle bundle;
    String bundleName;
    String bundleKey;
    bundleName = prefix + name;
    synchronized (mutex) {
      bundleKey = locale.toString() + bundleName;
      bundleHandle =
                  (Integer) bundleHandleMap.get(bundleKey);
    }
    return bundleHandle == null ? -1 : bundleHandle.intValue();
  }
  public static ResourceBundle getBundle(String name, Locale locale,
                       ClassLoader cl) throws NullPointerException,
                                              MissingResourceException {
                //Manca implementazione
  }
  //Metodo piu importante:
  public static String getString(int handle, String key) {
    ResourceBundle bundle;
    String s;
    synchronized (mutex) {
      if (handle < 0 ||
          handle >= bundleList.size() ||
          key == null) {
        bundle = null;
      } else {
        bundle = (ResourceBundle)
        bundleList.get(handle);
      }
    }
    if (bundle == null) {
      s = null;
    } else {
      try {
        s = bundle.getString(key);
      } catch (Exception e) {
        s = null;
      }
    }
    return s;
  }
}
```

4 ALGORITMI E STRUTTURE DATI

I secondi pensieri sono
sempre i più saggi.

-Euripide

4.1 STRUTTURE DATI

In questo capitolo vengono riportati i principali algoritmi e le strutture dati che rappresentanto il cuore del *dbms*.

4.1.1 Le righe

Il modo di descrivere le righe è ereditato dal software di pianificazione. Questo fatto rappresenta una scelta che in futuro potrà cambiare perchè impone la creazione di nuove classi durante l'esecuzione. Tale scelta è stata dettata per mantenere un' omegeneità tra le tabelle ottenute e il risultato di una query. Nel *dbms* pertanto una riga viene codificata per mezzo di un oggetto *Java* dove i suoi attributi sono le colonne. Risulta chiaro il bisogno dell'introspezione o *reflection* per ottenere la lista degli attributi; non è possibile altrimeti, non sapendo quali classi esistono a priori, accedere ai dati. Ad ogni tipo di riga è associata una classe come questa 7.

Codice 7: Esempio tipo di riga.

```
public class ExampleRecord implements Serializable {
        Integer index;
        String attribute0;
        String attribute1;
        String attribute2;

        public ExampleRecord(String attribute0, String attribute1,
                        String attribute2,nteger index) {
                setIndex(index);
                setAttribute0(attribute0);
                setAttribute1(attribute1);
                setAttribute2(attribute2);
        }
        public Integer getIndex() {
                return index;
        }
```

```
        public void setIndex(Integer index) {
                this.index = index;
        }
        public String getAttribute0() {
                return attribute0;
        }
        public String getAttribute1() {
                return attribute1;
        }
        public String getAttribute2() {
                return attribute2;
        }
        public void setAttribute0(String attribute0) {
                this.attribute0 = attribute0;
        }
        public void setAttribute1(String attribute1) {
                this.attribute1 = attribute1;
        }
        public void setAttribute2(String attribute2) {
                this.attribute2 = attribute2;
        }
}
```

Come si può vedere per ogni attributo sono implementati i metodi *getter* e *setter*, questo per rispettare la convenzione dei *JavaBean* utile alla libreria *BeanUtils*. Questa loro particolarità (JavaBean) li rende paragonabili proprio alle righe che si vogliono rappresentare. Come già anticipato quanto appena spiegato è cruciale per la comprensione di alcune scelte.

4.1.2 Le tabelle

Partendo dalle righe per riuscire a rappresentare le tabelle il passo è semplice, basta pensarle come collezione di righe, ottenendo così la struttura dati più adatta: l'*array*. A dire la verità nel *dbms* non viene usato l'*array* classico ma l' *ArrayList* di *Java*, si migliora la leggibilità del codice grazie alle funzioni aggiuntive di cui è dotato.

4.1.3 Metadati

Abbiamo anticipato che un moderno *dbms* non può evitare di contenere anche una collezione di informazioni aggiuntive, chiamate metadati (*metadata*). I metadati nel progetto possono essere concettualmente separati in due parti:

1. tabelle aggiuntive definite dallo standard di *SQL* (vedi C);
2. rappresentazione interna degli schemi di tabelle, viste, etc.

Per la prima si intuisce con facilità che si tratta di aggiungere tali tabelle all'interno del *dbms*, in modo che vengano trattate secondo le specifiche imposte. Per la seconda è utile definire un nuovo insieme di classi, per istanziare i singoli schemi che devono essere manipolati durante l'esecuzione dell'applicazione stessa. Il codice del *software HSQLDB* è stato preso come linea guida per la realizzazione della collezione appena menzionata ed è composta dalle seguenti classi:

1. ***Schema***. Classe usata per gli schemi di un database.
2. ***SchemaTable***. Rappresenta lo schema di una tabella.
3. ***SchemaColumn***. Utilizzato internamente per mantenetere informazioni aggiuntive sulle colonne.

4.1.4 Le espressioni

Probabilmente saprete che le *query SQL* da eseguire possono specificare delle condizioni. Tali condizioni devono essere verificate da ogni riga della tabella di *output*. Viene naturale dover realizzare una classe per la valutazione di queste espressioni compilate durante l'analisi sintattica. L'idea di base utilizzata è quella di una struttura ricorsiva simile ad un albero. Ci sono tre tipi diversi di espressioni:

1. Unaria;
2. Binaria;
3. Ternaria;

Un'altra tassonomia per le espressioni è la seguente:

1. ***LogicalExpression***, usate per rapresentare una qualunque condizione;
2. ***ColumnExpression***, usate per estrarre i valori da una singola colonna durante l'uso dell'algoritmo per il prodotto cartesiano tra tabelle;
3. ***AggregateExpression***, simili alle precedenti ma vi è al loro interno la logica per gestire le funzioni aggregate di *SQL* (vedi C);
4. ***LikeExpression***, per realizzare l'operatore like di *SQL*;
5. ***ValueExpression***, per valori come interi, date, numeri in virgola mobile, stringhe etc.

Analizzando il caso binario si nota che l'espressione è composta da due espressioni e da un operatore (esempi: *and*, *or*, *not*); le due sotto-espressioni possono essere a loro volta composte da uno, due o tre espressioni rispettivamente con le opportune operazioni o operatori associati. Riportiamo il caso semplificato delle espressioni logiche (vedi 8), un caso particolare ma abbastanza esplicativo del funzionamento in generale.

4.2 ALGORITMI

4.2.1 L'operatore like

L'operatore di confronto *LIKE* può essere usato per il confronto di stringhe rispetto a un modello (*pattern*). Le stringhe parziali sono specificate utilizzando caratteri riservati: il segno di percentuale (%), che sostituice un numero arbitrario di zero o più caratteri e il segno di sottolineatura (_, underscore), che sostituisce invece un singolo carattere. Possiamo pertanto pensare l' input dell'operatore *like* come un'espressione regolare vedi A.1.8. Se nella stringa è necessario un segno di sottolineatura o di percentuale, questo deve essere preceduto da un carattere di *escape*, che va specificato dopo la stringa usando la parola chiave *ESCAPE*. Qualsiasi carattere non usato nella stringa può essere usato come carattere di escape. Si rende d'obbligo anche una regola riguardante apostrofi o apici che devono essere inseriti in una stringa, perchè tali caratteri sono usati per iniziare e terminare le stringhe. Ad esempio se serve una virgoletta singola ('), va rappresentata con doppie virgolette (''), in modo che non venga considerata come fine stringa. Riportiamo una parte della classe che realizza l'analisi che una data stringa rispetti il pattern specificato 9

Codice 9: Algoritmo per l'operatore *like*.

```
private boolean compareAt(Object o, int i, int j, int iLen, int jLen,
                          char cLike[], int[] wildCardType) {
  for (; i < iLen; i++) {
    switch (wildCardType[i]) {
      case 0 :// general character
        if ((j >= jLen) ||
            (cLike[i] != ((String) o).charAt(j++))) {
          return false;
        }
        break;
      case UNDERSCORE_CHAR :    // underscore
        if (j++ >= jLen) {
          return false;
        }
        break;
      case PERCENT_CHAR :       // percent
```

```
        if (++i >= iLen) {
          return true;
        }
        while (j < jLen) {
          if ((cLike[i] == ((String) o).charAt(j))
               && compareAt(o, i, j, iLen,
                            jLen, cLike,
                            wildCardType)) {
            return true;
          }
          j++;
        }
        return false;
    }
  }
  if (j != jLen) {
    return false;
  }
  return true;
}
```

L'algoritmo è costituito da uno *switch* facile da capire perchè divide l'analisi in tre semplici casi. Nel primo si è nella situazione di dover verificare la presenza di un carattere in particolare. La seconda quando ci si trova a disposizione l'uso di un *underscore* che sarà consumato come il carattere della stringa di ingresso, nel caso non sia la fine altrimenti, viene restituito un'errore. L'ultimo deve trovare almeno un modo di eliminare una sequenza qualsiasi di caratteri dalla stringa rimanente in modo che dallo stato attuale si arrivi a riconoscere l'input, in caso contrario non viene riconosciuta. L'ulimo punto può essere visto come il comportamento tipico di un'automa a stati finiti non deterministico A.1.4.

4.2.2 Valutare le espressioni

Nel prossimo paragrafo viene spiegato un concetto di cui, forse, non siete a conoscenza.

Logica a tre valori

SQL ha svariate regole per gestire i valori *NULL*. Il valore in questione è usato per rappresentare un valore assente, che di solito, viene interpretato in tre modi differenti:

1. *Valore sconosciuto*. Un valore che un'entità possiede ma che non è attualmente conosciuto;

2. *Valore non disponibile*. L'esempio più semplice consiste in un dato che non si vuole (da parte del proprietario) sia reso pubblico;

AND	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

OR	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

NOT	
TRUE	FALSE
FALSE	TRUE
UNKNOWN	UNKNOWN

Figura 5.: Valori di verità nella logica a tre valori.

3. *Attributo non applicabile*. Un attributo Indirizzo è *NULL* per una persona senza abitazione, non essendo applicabile a quella persona;

Per un utente è impossibile stabilire in quale situazione rientri uno specifico caso, pertanto si è deciso che nella base di dati ciascun *NULL* venga considerato diversamente dagli altri. Il terzo valore (nella logica a tre valori) è *UNKNOWN* che codifica tutti i casi del valore *NULL*. Riportiamo nella tabella 5 i valori di verità della logica a tre valori. Saranno riportati in *output* in una *query* soltanto quelle righe che possiedono il valore di verità *TRUE*.

Il metodo di valutazione

Ricordando com'è formata un'espressione nel *dbms*, viene naturale valutare le espressioni logiche sfruttando la loro struttura ad albero. Si scende in profondità con la ricorsione e, dalle foglie (tipicamente *ValueExpression* e *ColumnExpression*), si risale fino alla radice. All'atto pratico è poco più di un semplice algoritmo di visita degli alberi. Riportiamo per completezza una versione ridotta del metodo usato 10.

Codice 10: Algoritmo per valutare le espressioni.

```
public Object getValue(Session session) {
  switch (eOtype) {
    case ExpressionOperationTypes.VALUE:
      return valueData;
    case ExpressionOperationTypes.NOT: {
      Boolean result = (Boolean) this.expressions[LEFT].getValue(session);
```

```
      return result == null ? null :
                       result.booleanValue() ? Boolean.FALSE :
                       Boolean.TRUE;
    }
    case ExpressionOperationTypes.AND: {
      Boolean r1 = (Boolean) this.expressions[LEFT].getValue(session);
      if (Boolean.FALSE.equals(r1)) {
        return Boolean.FALSE;
      }
      Boolean r2 = (Boolean) this.expressions[RIGHT].getValue(session);
      if (Boolean.FALSE.equals(r2)) {
        return Boolean.FALSE;
      }
      if (r1 == null || r2 == null) {
        return null;
      }
      return Boolean.TRUE;
    }
    case ExpressionOperationTypes.OR: {
      Boolean r1 = (Boolean) this.expressions[LEFT].getValue(session);
      if (Boolean.TRUE.equals(r1)) {
        return Boolean.TRUE;
      }
      Boolean r2 = (Boolean) this.expressions[RIGHT].getValue(session);
      if (Boolean.TRUE.equals(r2)) {
        return Boolean.TRUE;
      }
      if (r1 == null || r2 == null) {
        return null;
      }
      return Boolean.FALSE;
    }
    case ExpressionOperationTypes.EQUAL:
    case ExpressionOperationTypes.NOT_EQUALS:
    case ExpressionOperationTypes.LESS:
    case ExpressionOperationTypes.LESS_EQUALS:
    case ExpressionOperationTypes.GREATER:
    case ExpressionOperationTypes.GREATER_EQUALS:{
      Object o1 = this.expressions[LEFT].getValue(session);
      Object o2 = this.expressions[RIGHT].getValue(session);
      return compareValues(session, o1, o2);
    }
    default:
      throw Error.runtimeError(ErrorCode.NOT_IN_LOGICAL_EXPRESSION,
                                              "OPERATORE_SCONOSCIUTO");
    }
}
```

4.2.3 Il prodotto cartesiano

Nel modello relazionale l'operazione tra tabelle più usata, spesso abusata, è senza ombra di dubbio il prodotto cartesiano, punto di partenza anche per i *join*. La seguente definizone applicata agli insiemi chiarisce il risultato del prodotto cartesiano tra due tabelle, perchè basta vedere le tabelle come relazioni.

Definizione prodotto cartesiano 4.3. Siano A e B due insiemi non vuoti, si definisce prodotto cartesiano e si indica con $A \times B$ l'insieme formato da tutte le coppie ordinate tali che il primo elemento appartenga ad A e il secondo a B. In altro modo:

$$A \times B = \{(a, b) \mid a \in A \land b \in B\}$$

Per il prodotto tra più tabelle si procede con una definizione iterativa. Per effettuare il prodotto cartesiano nel codice è stato sviluppato un nuovo iteratore con, in aggiunta, la possibilità di effettuare un *reset* tornando al primo elemento. Consideriamo il caso più banale di sole due tabelle, risulta chiaro che la tabella a destra, la B, deve essere iterata un numero di volte pari alla cardinalità della tabella A estraendo così tutte le coppie per rispettare la definizione. Le condizioni sono verificate durante lo svolgimento dell'iterazione, non necessariamente alla fine. Il punto esatto di valutazione dipende da vari fattori, non ultimo l'attenzione posta da chi ha inserito la *query*. In sostanza le prestazioni dipendono fortemente anche da come sono strutturate le espressioni. Il risultato, in effetti, spesso è un sottoinsieme del prodotto cartesiano, mancando le righe che non rispettano le condizioni specificate nella *query*. Per una valutazione precisa delle prestazioni bisognerebbe tenere conto di molte situazioni ma per il caso in cui non vi siano condizioni il risultato è chiaro, il costo è $O(m \cdot n)$ dove n ed m sono le dimensioni (in righe) della prima e della seconda tabella rispettivamente. Semplificando ulteriormente nel caso entrambe le tabelle abbiano la stessa dimensione n si ha $O(n^2)$. Poco prima è stato usato il termine "abusato" riguardo a questa operazione, la motivazione deriva dal costo computazionale sicuramente più alto di molte altre operazioni in *SQL* (bisogna usarla se serve veramente). Qui di seguito l'algoritmo per eseguire il prodotto cartesiano, aggiungo solo che le espressioni vengono valutate nella classe *RangeIterator* e il metodo *next* cerca il prossimo elemento della tabella associata che verifica le condizioni, se esiste posizona il cursore su tale tupla restituendo *true*, altrimenti *false*.

Codice 11: Algoritmo prodotto cartesiano.

```
while (currentIndex >= 0) {
  RangeIterator ri = rangeIterators[currentIndex];
  boolean notStop = ri.next();
  if (notStop) {
```

```
    currentIndex++;
    if (currentIndex == iteratorSize) {
      Object[] data = new Object[this.indexLimitData];
      for(int i=0;i<this.indexStartAggregates;i++){
        Object value = this.columnExpressions[i].getValue(session);
        data[i] = value;//Da aggiungere il controllo di aggregazione
      }
      /* Aggiunta per group by e having */
      currentIndex--;
     }
  }else {
    currentIndex--;
  }
}
/* Inserimento dati nei JavaBean (le tuple vere) */
```

4.3.1 Clausola GROUP BY e funzioni di aggregazione

Essendo possibile in *SQL* effettuare raggruppamenti ed operazioni di aggregazione, vedi C, vi è la necessià di aggiungere tale possibilità nel *DBMS*. Per la clausola *GROUP BY* si procede in modo relativamente semplice, la riga corrente che verifica le condizioni dell'interrogazione viene inserita all'interno di una struttura dati *TreeMap* [11] (avendo avuto l'accortezza di impostare correttamente il *Comparator*). Se la riga che è stata prodotta non è presente, viene inserita nel risultato, altrimenti si prosegue nell'iterazione successiva. Il punto cruciale di questo metodo è rappresentato dal modo in cui vengono confrontate le tuple. Chiaramente se non si pone la dovuta precisone nella verifica, si ottengono prestazioni molto scadenti. Ad esempio sapere che due tuple uguali devono avere il medesimo *hashcode* permette di mgliorare, di molto, le prestazioni. Per quanto riguarda le funzioni di aggregazione la cosa si complica, bisogna cercare di comprendere l'idea di fondo tralasciando i dettagli specifici. Quello che è possibile capire è che la tupla viene inizialmente creata senza valori nelle colonne di aggregazione. Il passo sucessivo è aggiornare il valore della colonna ogni volta che nel punto del *GROUP BY* il metodo *getGroupData* restituisce una riga già presente nell'albero (il medesimo *TreeMap*) o settare il primo valore in caso non lo sia. Sostanzialmente le tuple vengono scorse ponendo attenzione alle classi di equivalenza a cui appartengono in modo da poterle raggruppare e, calcolare correttamente i valori delle colonne corrispondenti alle funzionni di aggregazione. Riportiamo in 12 il codice relativo a questi due fatti.

Codice 12: Porzione per *GROUP BY* e aggregazione.

```
Object groupData = null;
if (isAggregated || isGrouped) {
        groupData = navigator.getGroupData(data);
}
```

```
for (int i = indexStartAggregates; i < indexLimitExpressions; i++) {

  data[i] = exprColumns[i].updateAggregatingValue(session,
        data[i]);
}

if (groupData == null) {
        navigator.add(data);
}
```

4.3.2 Clausola ORDER BY

Per effettuare l'ordinamento di una tabella rispetto ad una collezione di attributi bisogna prestare la massima attenzione a quale algoritmo di ordinamento utilizzare. Consideriamo ad esempio il caso del *Merge Sort*, risulta chiaro come da un lato la complessità in termini di tempo sia ottima $O(n \log(n))$ dall' altro quella spaziale sia abbastanza scadente. Date le dimensioni delle tabelle in alcuni casi è fuori questione l'utilizzo di tale algoritmo all'interno del motore del dbms, al contrario l'algoritmo *Quiksort* rappresenta la scelta ideale in questo caso. Abbiamo deciso di utilizzare la versione implementata in *hsqldb* perchè compatibile con le strutture dati del dbms sviluppato.

Codice 8: Classe espressioni logiche.

```
public class LogicalExpression extends Expression {
  public LogicalExpression(int eOtype){
    super(eOtype);
    dataType = DataType.SQL_BOOLEAN;
  }
  public LogicalExpression(boolean b) {
    super(ExpressionOperationTypes.VALUE);
    dataType = DataType.SQL_BOOLEAN;
    valueData = b ? Boolean.TRUE : Boolean.FALSE;
  }
  /* Espressione logica per uguaglianza di colonne  */
  LogicalExpression(RangeVariable leftRangeVar, int colIndexLeft,
                    RangeVariable rightRangeVar, int colIndexRight) {
    //Codice per creare l' espressione sinistra e destra
    this.expressions[LEFT] = leftExpression;
    this.expressions[RIGHT] = rightExpression;
    setEqualityMode();
    dataType = DataType.SQL_BOOLEAN;
  }
  public LogicalExpression(int type, Expression e) {
    super(type);
    this.expressions = new Expression[UNARY];
    this.expressions[LEFT] = e;
    //Codice per alcune informazioni
  }
  /* Espressione per uguaglianza di espressioni
     ad esempio colonna e valore*/
  public LogicalExpression(Expression left, Expression right) {
    super(ExpressionOperationTypes.EQUAL);
    this.expressions = new Expression[BINARY];
    this.expressions[LEFT] = left;
    this.expressions[RIGHT] = right;
    setEqualityMode();
    dataType = DataType.SQL_BOOLEAN;
  }
  void setEqualityMode() {
    //Non riportato il codice
  }
  /*Ritorna il risultato dell'espressione*/
  public Object getValue(Session session) {
    //Non riportato il codice
  }
}
```

5 ANALIZZATORE LESSICALE

> Il linguaggio dell'uomo è simile agli arazzi finemente ricamati; deve essere disteso per rivelare i suo disegni ma quando è rinchiuso su se stesso li nasconde e li distorce.
>
> -Temistocle

5.1 SCOPO DELL'ANALIZZATORE LESSICALE

Come prima fase della compilazione l'analizzatore lessicale deve leggere la sequenza di caratteri che forma il programma sorgente, ragrupparli in lessemi e generare una sequenza di token corrispondente. Tale sequenza viene infine inviata all'analizzatore sintattico.

5.2 LESSEMI, TOKEN E PATTERN

Elenchiamo tre concetti dandone una descrizione generale e semplificata:

1. Un *token* è una coppia formata da un nome e un attributo, eventualmente opzionale. Il nome è qualcosa di astratto che identifica uno specifico tipo di unità lessicale, ad esempio una parola chiave o un identificatore;

2. Un *pattern* è una descrizione formale della forma che il lessema di un token assume;

3. Un *lessema* è una qualsiasi sequenza di caratteri della stringa d'ingresso che corrisponde al pattern di un token;

Di solito i pattern da riconoscere sono espressi mediante espressioni regolari **A.1**.

5.3 TOKEN

Quando un lessema soddisfa un determinato pattern, l'analizzatore deve fornire alle fasi successive non solo il lessema ma altre informazioni codificate mediante attributi sullo specifico lessema riconosciuto.

5.4 IL PROBLEMA DEL RICONOSCIMENTO DELLE PAROLE RISERVATE E DEGLI IDENTIFICATORI

Si può presentare un problema nel riconoscimento delle parole chiave e degli identificatori. In genere parole chiave come *if* e *then*, nel caso di un linguaggio classico, *select* e *from* in quello del linguaggio *SQL*, sono riservate. Per cui non possono in alcun modo essere degli identificatori. Ci sono due semplici modi per trattare tutte le parole chiavi che possono essere prefisso di un identificatore.

1. Installare fin dall'inizio le parole chiavi in una struttura dati, ad esempio una tabella delle parole chiavi, affinchè indichi che tali stringhe non possano mai essere identificatori;

2. Creare un differente diagramma di transizione per ogni parola chiave.

5.5 ERRORI LESSICALI

L'analizzatore lessicale può già riconoscere eventuali errori nel codice sorgente come un lessema che non verifica nessun pattern. D'altro canto però è impossibile da parte sua riconoscere errori più sintattici che dovranno essere riconosciuti invece nel parser. Nel progetto il cui codice sorgente è una stringa del linguaggio SQL, per l'analizzatore lessicale questa specifica stringa "SELECT FROM NOME_TABELLA" è corretta.

5.6 ANALIZZATORE LESSICALE PROGETTO

Dato che i pattern non sono altro che espressioni regolari si possono costruire automi a stati finiti in grado di riconoscerli. Per l'analizzatore sviluppato è stato scelto di utilizzare con alcune modifiche quanto è già stato fatto nel software di HSQLDB. In HSQLDB sostanzialmente non viene utilizzato un automa ma viene sfruttato lo stato implicito presente durante l'esecuzione di un programma (algoritmo). Ad ogni carattere letto si intraprendono delle scelte grazie ai costrutti

decisionali del linguaggio Java. Si poteva anche costruire un automa a stati finiti come descritto in [19] procedendo successivamente anche alla fase di minimizzazione descritta dettagliatamente in [14] ma il guadagno nel fare tutto ciò sarebbe stato minimo e avrebbe reso il codice meno leggibile a causa del formalismo. Viene riportato il metodo principale della clase *Scanner*.

Codice 13: Metodo principale analizzatore lessicale.

```
//Analizzatore Lessicale
void scanNextToken() {
  //Riporto l'analizzatore lessicale allo stato iniziale
  stateReset();

  //Leggo il carattere corrente
  int currentCharacter = charAt(currentPosition);

  switch (currentCharacter) {
    case '0':
    case '1':
    case '2':
    case '3':
    case '4':
    case '5':
    case '6':
    case '7':
    case '8':
    case '9':
    case '.':
      //Tipo di Token numerico (valore)
      nextToken.tokenType = Tokens.LES_VALUE; /
      scanTypeNumber();
      return; //Ha finito
    case '-' :
      //Non visualizzato
      return;
    case '*': //Serve per Select * from NomeTabella
      nextToken.tokenString = Tokens.S_RES_SQL_ASTERISK;
      nextToken.tokenType = Tokens.RES_SQL_ASTERISK;
      currentPosition++;
      nextToken.isDelimiter = true;
      return;
    case ',':
      nextToken.tokenString = Tokens.S_RES_COMMA;
      nextToken.tokenType = Tokens.RES_SQL_COMMA;
      currentPosition++;
      nextToken.isDelimiter = true;
      return;
    case '=':
      nextToken.tokenString = Tokens.S_RES_EQUAL;
      nextToken.tokenType = Tokens.OP_SQL_EQUAL;
      currentPosition++;
      nextToken.isDelimiter = true;
```

```
      return;
    case '\'':
      scanCharacterString();

      if (nextToken.isMalformed) {
        return;
      }
      nextToken.dataType = CharacterType.getCharacterType();
      nextToken.tokenType = Tokens.LES_VALUE;
      nextToken.isDelimiter = true;
      return;
    case '\"':
      nextToken.tokenType = Tokens.LES_IDENTIFIER_DELIMITED;
      break;
    case '(':
      nextToken.tokenString = Tokens.S_RES_OPENBRACKET;
      nextToken.tokenType = Tokens.RES_SQL_OPENBRACKET;
      currentPosition++;
      nextToken.isDelimiter = true;
      return;
    case ')':
      nextToken.tokenString = Tokens.S_RES_CLOSEBRACKET;
      nextToken.tokenType = Tokens.RES_SQL_CLOSEBRACKET;
      currentPosition++;
      nextToken.isDelimiter = true;
      return;
    case '<':
      //Non visualizzato
      return;
    case '>':
      //Non visualizzato
      return;
    }
    scanIdentifier();
    setIdentifier();
}
```

6 ANALIZZATORE SINTATTICO

> La cosa principale e più importante, almeno per gli scrittori di oggi, è ridurre la lingua all'essenziale, fino a mostrarne l'intima struttura.
>
> -Ernest Hemingway

6.1 SCOPO DEL PARSER

Il parser riceve come input una sequenza di Token dall'analizzatore lessicale, e verifica se tale sequenza può essere generata dalla grammatica. Per sequenze ben fomate costruisce un albero di parsing e lo passa alla parte restante del compilatore. Di fatto, non è necessario costruire l'albero esplicitamente, poichè le azioni di verifica e di traduzione possono essere fuse con il parsing.

6.2 ERRORI SINTATTICI

Come già compreso gli errori di programmazione più comuni si verificano a livelli diversi:

1. **Errori lessicali**. Errori di scrittura dei lessemi.
2. **Errori sintattici**. Si possono sempre pensare come scrittura di una stringa non appartenente al linguaggio.
3. **Errori logici**. Il sorgente del programmatore non corrisponde al suo intento.

La precisione dei metodi di parsing permette di rivelare molto efficientemente gli errori sintattici. In alcuni casi come nei parser LL e LR, si possono rivelare gli errori appena possibile cioè appena la stringa non può essere completata con un suffisso tale da renderla appartenente al linguaggio. Il gestore degli errori in un parser ha degli obbiettivi semplici ma complessi da realizzare:

1. segnalare la presenza di errori in modo chiaro e accurato;

2. recuperare rapidamente un errore in modo da poter rilevare eventuali errori successivi;

3. non appesantire l'elaborazione dei programmi corretti.

6.3 GRAMMATICA DI UN PARSER

Non tutta la sintassi dei linguaggi di programmazione può essere descritta dalle grammatiche libere dal contesto. Ad esempio il fatto che gli identificatori debbano essere dichiarati prima di essere usati. Le sequenze di token accettate da un parser sono un sovrainsieme di quelle del linguaggio di programmazione. Ci sono poi dei problemi da risolvere come l'ambiguità della grammatica etc.

6.4 TIPI DI PARSER

Ci sono vari tipi di parser, rappresentano in sostanza modi diversi di costruire l'albero sintattico. Ognuno con vantaggi, svantaggi e limiti.

6.4.1 Parsing top-down

Il parsing top-down ha come fine quello di costruire un albero sinattico corrispondente alla stringa d' ingresso, partendo dalla radice e creando i nodi secondo un ordinamento depth-first. In modo più formale significa trovare una derivazione sinistra della data stringa. A ogni passo del parsing il problema cruciale consiste nel determinare la produzione da applicare a un non terminale. La più generale forma di parsing top-down è detto parsing a discesa ricorsiva, che può richiedere la tecnica del *backtracking* cioè tornare indietro sulla propria scelta. Un caso particolare è il parsing predittivo che non richiede *backtracking*. La classe di grammatiche per le quali si può costruire un parser predittivo analizzando al più k simboli successivi a quello corrente è a volte indicata come $LL(k)$. Si può infine costruire un parser predittivo non ricorsivo gestendo uno stack esplicitamente, piuttosto che facendo affidamento sullo stack implicito dovuto alle chiamate ricorsive.

6.4.2 Parsing bottom-up

Il parsing bottom-up al contrario del top-down non parte dalla radice ma dalle foglie per arrivare solo alla fine alla radice. Si può pensare al parsing bottom-up come al processo di progressiva riduzione di una stringa di ingresso w fino al simbolo iniziale della grammati-

ca. Uno dei più importanti parser bottom-up è il parsing *shift-reduce* o impila riduci. Benchè le operazioni principali siano shift e reduce, vi sono in tutto quattro azioni che può effettuare:

1. **Shift**.
2. **Reduce**.
3. **Accept**.
4. **Error**.

Ci sono delle grammatiche libere dal contesto CFG non riconoscibili da alcun parser shift-reduce.

6.4.3 Parsing LR

Il parser più diffuso noto con il nome di parser LR(k) che usa la strategia di scorrere la stringa da sinistra a destra per formare una derivazione destra in ordine inverso. k invece sono i simboli di lookhead usati per prendere le decisioni. I parser LR, come spesso accade, si basano su tabelle. Una grammatica è riconoscibile da un parser LR che è in grado di riconoscere gli handle delle forme sentenziali destre sulla cima dello stack. È inoltre interessante per i seguenti motivi:

1. è possibile costruire un parser LR per sostanzialmente tutti i linguaggi di programmazione per cui esiste una CFG che li riconosca;
2. il parsing LR funziona come un parsing shift-reduce privo di backtracking;
3. ha la caratterisica di poter rilevare un errore appena possibile;
4. l'insieme delle grammatiche libere dal contesto che può trattare contiene sia quello dei metodi predittivi che LL;

Il suo svantaggio è la mole di lavoro che un programmatore deve svolgere per la costruzione di tale parser nel contesto di un comune linguaggio di programmazione. Per questo di solito viene sfruttato per delle parti o vengono usati strumenti automatizzati.

Un caso particolare LR(0)

Prima di entrare nel merito della costruzione di un esempio bisogna fare alcune definizioni e precisazioni.

Definizione item LR(0) 6.4.3.1. Un item $LR(0)$ di una grammatica G è una produzione nella quale compare un punto nel corpo. Ad esempio per la produzione $P \to ABC$ ci sono quattro item:

$$P \to \cdot ABC$$

$$P \to A \cdot BC$$

$$P \to AB \cdot C$$

$$P \to ABC \cdot$$

Per il caso particolare $P \to \epsilon$ esiste il solo item $P \to \cdot$.

Il concetto di item viene usato nel contesto del parser LR per indicare quale porzione di una produzione si è già analizzato. Il secondo strumento è una collezione di insiemi di item, chiamata collezione canonica, fornisce gli elementi con cui costruire un DFA (chiamato ora automa $LR(0)$) per le decisioni di parsing. Per la costruzione di questa collezione si definisce innanzitutto la grammatica aumentata di una grammatica G, la chiusura degli insiemi di item e la funzione GOTO.

Definizione grammatica aumentata 6.4.3.2. Sia G una grammatica, si definisce grammatica aumentata G' di G la grammatica con un nuovo simbolo iniziale S' e una nuova produzione $S' \to S$.

Viene usata solo per indicare la fine del parsing e segnalare o meno l'accettazione della stringa d'ingresso.

Definizione chiusura degli insiemi di item 6.4.3.3. Sia I l'insieme degli item di una grammatica G, allora $CLOSURE(I)$ è un nuovo insieme costruito a partire da I utilizzando queste due regole:

1. si aggiungono tutti gli elementi di I a $CLOSURE(I)$.
2. se $A \to \alpha \cdot B\beta \in CLOSURE(I)$ e $B \to \gamma$ è una produzione di G , si deve aggiungere l'item $B \to \cdot\gamma$ a $CLOSURE(I)$. Si ripete ciò finchè l'insieme $CLOSURE(I)$ non può avere ulteriori item.

Un semplice codice per il calcolo della chiusura 6

Definizione funzione $GOTO(I, X)$ 6.4.3.4. $GOTO(I, X)$ è la chiusura dell'insieme di tutti gli item $A \to \alpha X \cdot \beta$ tali che $A \to \alpha \cdot X\beta \in I$

Tale nuova funzione è usata per le transizioni dell' automa $LR(0)$. Siamo pronti per la costruzione della collezione canonica. Un semplice algorimo figura 7

```
function CLOUSURE(I)
    J = C;
    repeat
        for ogni item A → α · Bβ ∈ C do
            for ogni produzione B → γ di G do
                if B → ·γ non appartiene a C then
                    aggiungi B → ·γ a C;
                end if
            end for
        end for
    until nessun nuovo item aggiunto a J;
    return C;
end function
```

Figura 6.: Algoritmo per la chiusura.

```
function CANONIC(G′)
    CA = CLOSURE(S → ·S);
    repeat
        for ogni insieme di item I in CA do
            for ogni simbolo grammaticale X do
                if GOTO(I, X) non è vuoto e non è in CA then
                    aggiungi GOTO(I, X) a CA;
                end if
            end for
        end for
    until nessun nuovo item aggiunto a J;
    return CA;
end function
```

Figura 7.: Algoritmo per collezione canonica.

SLR 0 PARSING LR SEMPLICE Ora abbiamo tutti gli strumenti per mostrare il funzionamento di un parser LR(0) semplice o SLR. Si costruisce l'automa LR(0) nel seguente modo:

1. gli stati sono gli insiemi di item della collezione canonica;
2. le transizioni sono fornite dalla funzione GOTO;
3. lo stato iniziale dell'automa LR(0) è CLOSURE($S' \to \cdot S$);

Definizione tabella parsing SLR 6.4.3.5. Si definisce tabella di parsing SLR un tabella formata dalle seguenti funzioni:

1. la funzione **GOTO**;
2. la funzione **ACTION**;

La funzione di ACTION(i, a) assume le seguenti quattro forme:

1. Shift j, in cui j è uno stato;
2. Reduce $A \to \beta$;
3. Accept, il valore per indicare l'accettazione della stringa d'ingresso;
4. Error, quando viene scoperto un errore;

Con $C = \{I_0, I_1, ..., I_n\}$ la collezione degli insiemi di item LR(0) di G'. Le azioni di parsing per lo stato i sono determinate in questo modo:

1. se $A \to \alpha \cdot a\beta \in I_i$ e GOTO$(I_i, a) = I_i$. Si pone ACTION$(i, a) =$ Shift j;
2. se $A \to \alpha\cdot \in I_i$ si pone ACTION$(i, a) =$ Reduce $A \to \alpha$ per ogni $a \in$ FOLLOW(A) (vedi [19]);
3. se $S' \to S\cdot \in I_i$, allora si pone ACTION$(i, a) =$ Accept;

Si utilizza infine per il parsing l'algoritmo 8.

6.5 ANALIZZATORE SINTATTICO PROGETTO

Il parser sviluppato nel dbms non appartiene a nessuna delle classi appena descritte ma rappresenta un collage di alcune di queste. Inoltre non supporta tutte le funzioni di SQL neanche relative alla sola SELECT. La parte principale è costituita da un parser top-down ricorsivo con uso di caratteri di lookahead. La grammatica per cui è pensato è costituita dalle seguenti produzioni:

1. S → Q | (Q) | Q U S | Q I S | Q E S;

```
index = 0;
a = w[index];
while true do
    s = lo stato sulla cima dello stack;
    if ACTION(s, a) = Shift t then
        aggiungi t sullo stack;
        index = 1;
        a = w[index];
    else if ACTION(s, a) = ReduceA → β then
        rimuovi |β| simboli dallo stack;
        t = il nuovo stato in cima allo stack;
        aggiungi lo stato GOTO(t, A) sullo stack;
        stampa la produzione A → β;
    else if ACTION(s, a) = Accept then
        return
    else
        gestione degli errori;
    end if
end while
```

Figura 8.: Algoritmo parsing SLR.

2. $Q \rightarrow query \mid S$;

Chiaramente è una versione semplificata perchè è omessa la parte relativa alle subquery. Per quanto concerne le espressioni è stato cotruito un parser SLR per tale grammatica non ambigua:

1. $E \rightarrow E + T \mid T$;
2. $T \rightarrow T * F \mid F$;
3. $F \rightarrow (E) \mid id$;

Nella figura 9 la tabella di parsing SLR relativa a tale grammatica.

Stato	**id**	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1	s6					acc			
2		r2	s7		r2	r2			
3		r4	s4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Figura 9.: Tabella parsing SLR per le espressioni.

7 JAVA RMI

> Molte mani rendono il lavoro leggero.
>
> -John Heywood

Oggi, la costruzione dei sistemi distribuiti basati sugli oggetti è l'approccio dominante nell'area dello sviluppo software, specialmente nel caso di applicazioni aziendali. All'inizio, in questo contesto sono state sviluppate diverse tecnologie per permettere la comunicazione; assieme alle loro funzioni sono identificate con il nome di middleware. Nel frattempo linguaggi di programmazione come Java, hanno cominciato ad integrare nuovi costrutti per gli oggetti distribuiti. Questi sono gli esempi più importanti:

1. **Java RMI**. *Oracle Remote Method Invocation*;
2. **.NET COM+**. *Componenet Object Model Plus*;
3. **OMG CORBA**. *Common Object Request Broker Architecture*;

7.1 SISTEMI DISTRIBUITI

Informalmente un sistema distribuito consiste in un numero di componenti software più o meno autonomi che possono essere eseguiti su differenti computer. Le motivazioni storiche che hanno portato allo sviluppo di tale tecnologia sono:

1. l'esplosione delle capacità computazionali dei processori;
2. lo sviluppo di reti più veloci;
3. i miglioramenti del software;
4. la rinuncia di sistemi con struttura gerarchica;

Quelle, al contrario, più legate alle caratteristiche:

1. scalabilità;
2. trasparenza;
3. eterogeneità;

4. tolleranza ai guasti;

Lo strato del middleware è inserito tra quello di rete del sistema operativo e applicativo. Ci sono vari tipi di middleware:

1. **classici**. Middleware orientati alle transazioni; esempi sono *IBM CICS* o *BEA Tuxedo*;
2. **Message-Oriented Middleware (MOM)**. Permettono anche messaggi asincroni; tipici prodotti sono *IBM MQSeries* o *DEC MessageQueue*;
3. **RPC**. Sono middleware che permettono le chiamate a procedure remote.

La parte restante di questo capitolo si focalizza su java RMI (vedi [8]) confrontandolo in alcuni casi con CORBA (vedi [1]).

7.2 CARATTERISTICHE

Prima di entrare nel merito di quanto sviluppato è necessario descrivere brevemente quali siano le caratteristiche, le funzionalità, vantaggi e svantaggi presenti. Per il linguaggio Java non vi è via più semplice di *Java RMI* perchè è integrato direttamente nel linguaggio e permette di automatizzare, di molto, lo sviluppo di un'applicazione distribuita. Il suo competitor CORBA, ha invece il vantaggio di poter interagire con programmi scritti anche in altri linguaggi. In realtà l'uso dell'uno o dell'altro è motivato principalmente da questa differenza, rispondendo cioè a questioni di semplicità e integrazione da una parte oppure all'eterogeneità dall'altra. Chiaramente è stata fatta una forte semplificazione ma rispecchia ad esempio quanto accaduto nello sviluppo del dbms in questione; a proposito esiste anche la possibilià di far interagire le due tecnolgie. Ci sono altri dettagli che è meglio analizzare più in profondità, argomento dei prossimi paragrafi.

7.3 COMUNICAZIONE

Va compreso come avviene lo scambio degli oggetti, risultato di chiamate ai metodi remoti.

7.3.1 Serializzazione

La serializzazione è il processo che consiste nella conversione di una collezione di oggetti istanziati, contenenti riferimenti tra loro, in

uno *stream* di byte, i quali possono essere inviati per mezzo di un *socket* o manipolati come una stringa di dati. Un altro modo di vedere la serializzazione è come metodo usato da RMI per permettere lo scambio di oggetti tra più *Java Virtual Machine*; sia come argomenti in una chiamata a metodo remoto, sia come oggetto restituito nello stesso.

La necessità della serializzazione

Cosa significa per il *client* inviare un' istanza di una classe *A* come argomento? Come minimo deve significare che il *server* possa utilizzare i metodi pubblici dell'istanza. Per esempio immaginiamo che il client invii queste due informazioni:

1. il tipo dell'istanza, in questo caso *A*;
2. un identificatore unico per l'oggetto;

Java RMI lato *server* può utilizzare tali informazioni per costruire uno *stub* per l'istanza di *A*. Così quando il *server* chiama un metodo su qualcosa che pensa come un istanza di *A*, viene inoltrata nella rete. Questo approccio presenta tre svantaggi:

1. non si può acccedere ai campi degli oggetti passati;
2. possono risultare delle performance inaccettabili a causa della latenza nella rete sottostante;
3. rende l'applicazione più vulnerabile agli errori;

Questi tre punti sono la causa della necessità di effettuare la copia degli oggetti e del loro invio sulla rete.

Usare la serializzazione

Come già accennato, la serializzazione è un meccanisco integrato in Java, per trasformare un grafo di oggetti in uno *stream* di dati. Questo flusso di dati può essere allora manipolato, e una copia in profondità può essere svolta con il processo inverso conosciuto con il nome di deserializzazione. In particolare esistono tre usi della serializzazione:

1. come meccanismo di persistenza;
2. come meccanismo di copia;
3. come meccanismo di comunicazione;

Come rendere una classe serializzabile

Affinchè una classe sia serializzabile sono neccessarie le seguenti operazioni:

1. implementare l'interfaccia *Serializable*;
2. accertarsi che anche la super-classe sia serializzabile correttamente;
3. sovrascrivere i metodi *equals()* e *hashCode()*;

Se una classe è composta da un attributo anch'esso deve essere serializzabile affinchè lo sia la classe di partenza, proprio perchè consiste in una copia in profondità.

Problemi di performance

La serializzazione è costituita anche da un algoritmo di *marshalling* e di *demarshalling* con svariate possibilità di personalizzazione. Ogni programmatore esperto sa che questo comporta un livello prestazionale non eccelso, infatti la serializzazione non è un eccezione. A volte oltre ad essere lenta sovraccarica intensamente la rete di comunicazione. I problemi di prestazioni sono essenzialmente dovuti ai seguenti motivi:

1. dipende dalla *reflection*;
2. utilizza troppi dati di supporto per l'invio degli oggetti;
3. è molto semplice inviare più dati di quelli effettivamente necessari;

Anticipiamo che la dipendenza dalla reflection è un problema molto difficile da risolvere e non verrà trattato. Mentre per il problema dell'uso eccessivo di banda Java mette a disposizione uno strumento per limitarlo.

7.3.2 Esternalizzazaione

Per risolvere il problema delle prestazioni, il meccanismo di serializzazione permette di dichiarare una classe come *Externalizable* invece di *Serializable*. La grande differena tra i due è come alcuni metodi vengono usati. La serializzazione scrive sempre ad esempio le descrizioni di tutte le superclassi, inoltre scrive le informazioni associate all'istanza quando è necessario vederla come istanza di una superclasse (polimorfismo). L'esternalizzazione cerca di evitare per quanto possibile di inviare informazioni non necessarie supportata dal fatto che il programmatore deve prendersi carico di decidere molte più cose rispetto alla serializzazione.

7.4 CARICARE CLASSI AL VOLO

Una necessità sempre più richiesta nello sviluppo software, specialmente nel caso di sistemi distribuiti, è quella di poter caricare durante l'esecuzione delle nuove classi non integrate direttamente nell'applicazione.

7.4.1 Il Classloader

Sicuramente sarete a conoscenza che il sorgente Java viene compilato all'interno di file con estensione *.class*. Ogni *.class* contiene le istruzioni compilate per la *JVM* di una singola classe, chiamato comunemente *bytecode*. Probabilmente saprete anche che ogni classe viene compilata singolarmente e lincata dinamicamente a *runtime*.

Come le classi sono caricate

La macchina virtuale di Java, carica e valida individualmente ogni classe per mezzo di un oggetto detto caricatore di classi o *classloaders*. Il caricatore funziona nel seguente modo:

1. Quando la *JVM* viene avviata la prima volta, un singolo *classloader*, spesso chiamato *bootstrap classloader*, è creato. Questo è responsabile di caricare e creare molte delle classi usate dalla *JVM*;
2. Un applicazine può definire più di un *classloader*. Ogni *classloader* possiede ad eccezione del *bootstrap classloader* un *padre*. L'insieme dei *classloader* forma un albero dove il *bootstrap classloader* rappresenta la radice;
3. Quando una classe è necessaria, la *JVM* trova il *classloader* appropriato ed utilizza uno dei metodi *loadClass()*;

Esiste la possibilità che sia richiesta alla *JVM* una classe non presente dove indicato dalla varibile d'ambiente *classpath*. Modificando correttamente le impostazioni della *JVM* è possibile aggiungere, definendo opportunamente un gestore, questa evenienza.

7.5 LA CLASSE SERVER

Dopo quanto detto l'idea di base consiste nel fatto di poter caricare una classe dato un *url*. Quanto viene spiegato successivamente assume che l'*url* abbia la seguente forma *http://*. Esiste una classe astratta facilmente reperibile che fornisce l'interfaccia di come una classe *Server* dovrebbe essere realizzata per tale scopo. Riportiamo qui il suo codice 14.

Codice 14: Classe ClassServer per java RMI

```
/*
 * ClassServer.java -- a simple file server that can serve
 * Http get request in both clear and secure channel
 */

/**
 * Based on ClassServer.java in tutorial/rmi
 */
public abstract class ClassServer implements Runnable {

  private ServerSocket server = null;
  /**
   * Constructs a ClassServer based on <b>ss</b> and
   * obtains a file's bytecodes using the method <b>getBytes</b>.
   *
   */
  protected ClassServer(ServerSocket ss)
  {
    server = ss;
    newListener();
  }

  /**
   * Returns an array of bytes containing the bytes for
   * the file represented by the argument <b>path</b>.
   *
   * @return the bytes for the file
   * @exception FileNotFoundException if the file corresponding
   * to <b>path</b> could not be loaded.
   * @exception IOException if error occurs reading the class
   */
  public abstract byte[] getBytes(String path)
         throws IOException, FileNotFoundException;

  /**
   * The "listen" thread that accepts a connection to the
   * server, parses the header to obtain the file name
   * and sends back the bytes for the file (or error
   * if the file is not found or the response was malformed).
   */
  public void run()
  {
    Socket socket;
    // accept a connection
    try {
      socket = server.accept();
    } catch (IOException e) {
      System.out.println("Class␣Server␣died:␣" + e.getMessage());
      e.printStackTrace();
      return;
    }
```

```
  // create a new thread to accept the next connection
  newListener();
  try {
  OutputStream rawOut = socket.getOutputStream();
  PrintWriter out = new PrintWriter(new BufferedWriter(
                              new OutputStreamWriter(
                              rawOut)));
    try {
      // get path to class file from header
      BufferedReader in =
                    new BufferedReader(
                    new InputStreamReader(socket.getInputStream()));
      String path = getPath(in);
      // retrieve bytecodes
      byte[] bytecodes = getBytes(path);
      // send bytecodes in response (assumes HTTP/1.0 or later)
      try {
        out.print("HTTP/1.0 200 OK\r\n");
        out.print("Content-Length: " + bytecodes.length +
                  "\r\n");
        out.print("Content-Type: text/html\r\n\r\n");
        out.flush();
        rawOut.write(bytecodes);
        rawOut.flush();
      } catch (IOException ie) {
        ie.printStackTrace();
        return;
      }

    } catch (Exception e) {
      e.printStackTrace();
      // write out error response
      out.println("HTTP/1.0 400 " + e.getMessage() + "\r\n");
      out.println("Content-Type: text/html\r\n\r\n");
      out.flush();
    }
  } catch (IOException ex) {
    // eat exception (could log error to log file, but
    // write out to stdout for now).
    System.out.println("error writing response: " + ex.getMessage());
    ex.printStackTrace();
  } finally {
    try {
      socket.close();
    } catch (IOException e) {
    }
  }
}

/**
 * Create a new thread to listen.
 */
```

```
  private void newListener(){
    (new Thread(this)).start();
  }

  /**
   * Returns the path to the file obtained from
   * parsing the HTML header.
   */
  private static String getPath(BufferedReader in)
                throws IOException{
    String line = in.readLine();
    String path = "";
    // extract class from GET line
    if (line.startsWith("GET /")) {
      line = line.substring(5, line.length()-1).trim();
      int index = line.indexOf(' ');
      if (index != -1) {
        path = line.substring(0, index);
      }
    }
    // eat the rest of header
    do {
      line = in.readLine();
    } while ((line.length() != 0) &&
             (line.charAt(0) != '\r') && (line.charAt(0) != '\n'));

    if (path.length() != 0) {
      return path;
    } else {
      throw new IOException("Malformed Header");
    }
  }
}
```

Il passo successivo è la sua estensione specificando il comportamento voluto. Ad esempio si possono tenere le classi in memoria e non prenderle effettivamente dal *filesystem* locale, o remoto che sia. Abilitare tale nuova funzione implica di dover modificare le politiche di sicurezza della *JVM*, perchè per motivi di sicurezza, di default, è impedito l'uso di tali tecniche. Provate ad immaginare cosa accadrebbe nel qual caso un malintenzionato catturasse il flusso di byte per sostituirli con una propria classe malevola che implementa la medesima interfaccia.

7.6 SICUREZZA

Rendere un sistema distribuito, o comunque un'applicazione che usi le medesime tecniche, sicuro, è un compito molto arduo. Qui alcune tipologie di problemi riguardanti la sicurezza:

1. confidenzialità dei dati;
2. integrità dei dati;
3. validazione e autorizzazione;

7.6.1 Permessi

L'idea usata da Java per tali problemi risiede nell'introduzione di una lista di permessi associati ai vari pezzi di codice. Di solito un'applicazione è eseguita con i permessi più bassi, in un certo senso un' idea sfruttata anche nei *firewall* come *iptables*. Viene fornita la possibilità di ampliare le azioni disponibili da un'applicazione, ad esempio aprire un *socket*, una porta, etc. Tutto questo deve essere specificato in un file, chiamato *security policy*. I tipi di permessi:

1. *AWT*;
2. *Socket*;
3. *File*;
4. *Property*;

L'esempio classico del file di configurazione è quello che permette qualsiasi azione 15.

Codice 15: Policy di sicurezza per java.

```
grant
{
        permission java.security.AllPermission;
};
```

8 DRIVER JDBC

> Tu, che sei il modello più sagace della raffinatissima natura.
>
> -Shakespeare

8.1 COS'È IL DRIVER JDBC

Il driver JDBC (vedi [4] e [5]) non è altro che un' API per accedere virtualmente a qualsiasi tipo di database relazionale. L'API JDBC consiste in una collezione di classi e interfacce scritte in Java tali da permettere di sviluppare interamente applicazioni per database nel linguaggio di programmazione Java. In altri termni rende facili le seguenti tre cose:

1. stabilire una connessione con il database;
2. inviare query o operazioni di aggiornamento;
3. processare il risultato;

Un semplice esempio 16.

8.2 JDBC CONNECTION

Nel driver un oggetto di tipo Conncetion rappresenta una connessione con il database. Una singola applicazione (ad esempio Squir-

Codice 16: Esempio dei tre passi.

```
Connection con = DriverManager.getConnection("jdbc:myDriver:myCatalog",
                                             "myLogin",
                                             "myPassword");
Statement stm = con.createStatement();
ResultSet rs = stm.executeQuery("select␣a,b␣from␣table1");
while(rs.next()){
        String currentString = rs.getString("a");
        Integer currentInteger = rs.getInt("b");
}
```

Codice 17: Esempio apertura connessione.

```
String url = "jdbc:mbmsqldb:rmi://localhost:6000/MBMDB";
Connection conn = DriverManager.getConnection(url);
```

rel) possiede una o più connessioni con un singolo database e può essere connesso con molti database. Tale oggetto viene usato per reperire informazioni della stessa connessione per mezzo del metodo *Connection.getMetaData* e del database chiamando il metodo *Connection.getDatabaseMetaData*.

8.2.1 Apertura della connessione

Il modo classico per aprire una connessione consiste nel chiamare il metodo DriverManager.getConnection con un parametro di tipo stringa che rappresenta un URL. Il codice 17 semplifica la comprensione di quanto appena detto:

8.2.2 Interfaccia Connection

Interfaccia Connection: 18

8.3 JDBC STATEMENT

Un oggetto della classe Statement è usato per inviare query al database. Ci sono tre tipi di Statement:

1. Statement;
2. PreparedStatement;
3. CallableStatement;

Il più semplice è l'oggetto Statement usato per query senza parametri; PreparedStatement è usato per query precompilate con o senza parametri, mentre CallableStatement è usato per chiamare le "stored procedure".

8.3.1 Eseguire query usando l'oggetto Statement

L'interfaccia Statement fornisce tre differenti metodi per eseguire query:

1. executeQuery;
2. execute;

3. executeUpdate;

Il metodo executeQuery viene utilizato per query il cui risultato è un singolo ResultSet come per le *select*. Il metodo executeUpdate è usato per eseguire *insert*, *update* o *delate* ed anche *SQL DDL*. Infine il metodo execute è usato per eseguire query che restituiscono più di un ResultSet.

8.4 JDBC RESULTSET

Un oggetto di tipo ResultSet è grosso modo una tabella che contiene il risultato di una query SQL. In altri termini contiene le righe che soddisfano le condizioni della query. I dati contenuti si ottengono utilizzando l'insieme di metodi disponibili che permettono l'accesso alle colonne della riga corrente. Il metodo ResultSet.next serve per muoversi alla successiva riga rendendola la nuova corrente.

8.4.1 Tipi di ResultSet

I ResultSet hanno funzionalità differenti. Ad esempio alcuni possono muovere il cursore in entrambe le direzioni usando sia il metodo ResultSet.next che ResultSet.previous altri sono limitati solo al metoto ResultSet.next. Inoltre alcuni sono sensibili ai cambiamenti apportati dopo l'apertura come l'inserimento di una nuova riga, altri invece non riflettono tali cambiamenti. Le seguenti costanti definite nell'interfaccia ResultSet rappresentano le funzionalità disponibili, i nomi sono autoesplicativi:

1. TYPE_FORWARD_ONLY;
2. TYPE_SCROLL_INSENSITIVE;
3. TYPE_SCROLL_SENSITIVE;

8.5 JDBC METADATA

I database oltre a memorizzare le informazioni degli utenti, contengono dati che riguardano se stessi. Molti DBMS possiedono un insieme di tabelle di sistema per mantenere informazioni riguardo le tabelle, colonne, chiavi primarie, chiavi esterne, fuzioni e altro. Proprio per questo alcuni metodi delle classi del driver JDBC resituiscono, in funzione di tali dati le informazioni che riguardano non solo le tabelle disponibili ma anche le funzioni supportate dal driver specifico. Le principali interfacce implementate, in un driver, che forniscono tali dati sono:

1. java.sql.DatabaseMetaData;

2. java.sql.ResultSetMetaData;

3. java.sql.ParameterMetaData;

8.6 JDBC DATABASEMETADATA

L'interfaccia java.sql.DatabaseMetaData fornisce informazioni riguardo il database nel suo complesso. Più precisamente spiega come il driver e il DBMS lavorano assieme. Ad esempio molti metodi della classe che implementa DatabaseMetaData indicano in quale modo alcune funionalità siano supportate. Se sia il DMBS e il driver JDBC supportano una determinata funzionalità, il corrispondente metodo del driver restituisce true.

8.6.1 Oggetto di tipo ResultSet come valore di ritorno

Molti dei metodi presenti in DatabaseMetaData restituisco una lista di informazioni nella forma di un oggetto della classe che implementa ResultSet. Il modo di prelevare le informazioni non si distingue da quello classico utilizzato quando l'oggetto ResultSet rappresenta il risultato di una query. Un semplice esempio che illustra come prelevare le informazioni sugli schemi 19

Codice 19: Ottenere gli schemi.

```
/* Codice per aprire una connessione */
DatabaseMetaData dbmd = connection.getDatabaseMetaData();
ResultSet rs = dbmd.getSchemas();
while(rs.next()){
        String s = rs.getString(1);
        System.out.println("Nome␣schema:␣" + s);
}
```

8.7 JDBC RESULTSETMETADATA

Quando viene inviata una SELECT in un applicazione che utilizza il driver JDBC, si ottiene come risultato un oggetto di tipo ResultSet che soddisfa la richiesta. Si possono ottenere informazioni riguardo alle colonne presenti,i tipi di dato ecc ecc, creando un oggetto di tipo ResultSetMetaData ed invocando i suoi metodi.

8.8 JDBC E LE TRANSAZIONI

In *JDBC* tutti i servizi di supporto alla gestione delle transazioni sono concentrati a livello dell'interfaccia *java.sql.Connection*. Dando uno sguardo a 18, si possono individuare i seguenti metodi:

1. **setAutoCommit**;
2. **getAutoCommit**;
3. **commit**;
4. **rollback**;
5. **setTransactionIsolation**;
6. **getTransactionIsolation**;
7. **close**;
8. **isClosed**;

8.8.1 I metodi setAutoCommit e getAutoCommit

La specifica *JDBC* prevede, di default, che un oggetto di tipo *Connection* sia posto nello stato *auto-commit*. Questo significa che una connessione esegue automaticamente un operazione di *commit* di ogni *statement SQL*. In altre parle, per default, non viene concessa la possibilità di racchiudere più comandi *SQL* all'interno di una transazione. Il metodo *setAutoCommit* serve proprio per modificare tale comportamento, riceve un parametro di tipo *boolean* per:

1. il valore *true* serve per richiedere che l'oggetto di tipo *Connection* chiuda automaticamente le transazioni alla fine di ogni comando *SQL*, in sostanza è il comportamento *standard*;
2. con il valore *false* che il *commit* o l'*abort* delle transazioni avvenga in modo esplicito tramite l'invocazione dei metodi *commit* e *rollBack*;

L'uso del metodo *getAutoCommit* è semplice, serve solo per ottenere l'impostazione corrente.

8.8.2 I metodi commit e rollBack

Una volta impostato un oggetto nello stato non *auto-commit*, è possibile realizzare le transazioni. Chiaramente si aggiunge del carico allo sviluppatore, nel senso che dovrà occuparsi personalmente di decidere quando una transazione dovrà essere completata o meno. Occorre

fare una precisazione: l'isolamento transazionale è a livello di *Connection* cioè processi diversi che debbano avviare transazioni concorrenti devono utilizzare oggetti *Connection* diversi.

8.8.3 I metodi set/getTransactionIsolation

Va necessariamente aggiunto che sono possibili diversi livelli di isolamento per le transazioni, ognuno dei quali permette diversi gradi di protezione rispetto alle anomalie che possono accadere. Vi sono i seguenti livelli:

1. **TRANSACTION_NONE**. le transazioni non vengono supportate, equivalente al caso *setAutoCommit(true)*;
2. **TRANSACTION_READ_UNCOMMITTED**. nessun livello di isolamento è garantito;
3. **TRANSACTION_READ_COMMITTED**. vengono prevenute solo le perdite di aggiornamenti;
4. **TRANSACTION_REPEATABLE_READ**. si aggiunge al caso precedente la pssibilità di prevenire le anomalie riguardanti le letture non ripetibili;
5. **TRANSACTION_SERIALIZABLE**. è il massimo livello di isolamento, viene usato per prevenire qualsiasi anomalia;

Il metodo *getTransactionLevel*, come dice il nome, serve ad ottenere il livello imposto.

8.9 FUNZIONI DRIVER PROGETTO

Le funzioni sviluppate nel *driver jdbc* relativo al progetto possono essere riassunte con una frase: tutte quelle necessarie affinchè il client *Squirrel* e il *framework Hibernate* possano funzionare. I punti del *driver* dove si è concentrato il maggior spazio sono i seguenti.

1. **Connection**;
2. **ResultSet**;
3. **PreparedStatement**;
4. **DatabaseMetaData**;

Il metodo di sviluppo applicato in questo ambito è molto semplice, una funzionalità del *driver* viene progettata appena risulta possibile farlo. Nel senso che avendo avuto la necessità di utilizzarlo prima che

sia effettivamente completata l'intera applicazione, molte delle funzionalità previste (per mezzo di metodi) sono incompiute. I metodi non implementati possono essere suddivisi in due categorie:

1. metodi per cui è possibile che il *DBMS* non fornisca tale funzione;
2. metodi che devono essere sviluppati per rispettare lo standard;

Nel primo caso si può correttamente restituire l'eccezione che dichiara: tale funzionalità non è presente. I secondi obbligano in qualche modo a dover restituire un risultato, questo ha creato spesso problemi nel decidere quale valore di ritorno effettivamente restituire. Un consiglio di massima è di ritornare un risultato vuoto ma mai dei valori *null*, questo perchè il client potrebbe bloccarsi qualora venga chiamato un metodo su quel puntatore. Probabilmente avrete intuito che per quanto si possa stare attenti in questo stadio di sviluppo, non ancora conforme agli standard, non si ha la certezza della compatibilità con una specifica applicazione.

8.10 PER SVILUPPATORI DI DRIVER

Per avere un *driver* di qualità il consiglio è quello di implementare, inizialmente, tutte le interfacce della versione 1.0 del *driver jdbc*. Soltanto quando si sarà completato questo si potrà pensare alle specifiche più recenti e all'implementazione delle funzionalità opzionali.

8.10.1 Requisiti della versione 1.0

Per rispettare le specifiche della prima versione, uno sviluppatore deve fare quanto segue:

1. implementare completamente le seguenti interfacce:
   a) java.sql.Driver;
   b) java.sql.DatabaseMetaData, ad eccezione dei metodi aggiuntivi delle versioni successive;
   c) java.sql.ResultSetMetaData, ad eccezione dei metodi aggiuntivi delle versioni successive;
2. includere le sequenti interfacce:
   a) java.sql.CallableStatement;
   b) java.sql.Connection;
   c) java.sql.PreparedStatement;
   d) java.sql.ResultSet;
   e) java.sql.Statement;

Codice 18: Interfaccia Connection.

```
package java.sql;
public inteface Connection{
/*=================================
        Metodi per la creazione degli statemet
  =================================*/
  Statement createStatement() throws SQLException;
  PreparedStatement prepareStatement(String sql) throws SQLException;
  CallableStatement prepareCall(String sql) throws SQLException;
  Statement createStatement(int resultSetType,
            int resultSetConcurrency) throws SQLException;
  PreparedStatement prepareStatement(String sql,int resultSetType,
                    int resultSetConcurrency) throws SQLException;
  CallableStatement prepareCall(String sql,int resultSetType,
                    int resultSetConcurrency) throws SQLException;

/*=================================
                Metodi relativi alle transazioni
  =================================*/
   void setAutoCommit(boolean enableAutoCommit) throws SQLException;
   boolean getAutoCommt() throws SQLException;
   void commit() throws SQLException;
   void rollback();
   void close();
   boolean isClosed() throws SQLException;

/*=================================
            Funzioni avanzate
  =================================*/

 DatabaseMetaData getMetaData() throws SQLException;
 void setReadOnly(boolean readOnly) throws SQLException;
 boolean isReadOnly() throws SQLException;
 void setCatalog(String catalog) throws SQLException;
 String getCatalog() throws SQLException;
 SQLWarning getWarnings() throws SQLException;
 void clearWarnings()throws SQLException;
 void setTypeMap(java.util.Map map) throws SQLException;
 java.util.Map getTypeMap()throws SQLException;
 void setTransactionIsolation() throws SQLException;
 int getTransactionIsolation() throws SQLException;
 String nativeSQL() throws SQLException;

/*=================================
            Funzioni JDBC 3.0
  =================================*/
 void setHoldability(int holdability) throws SQLException;
 int getHoldability(int holdability) throws SQLException;
 Savepoint setSavapoint() throws SQLException;
 Savepoint setSavepoint(String name) throws SQLException;
 void rollback(Savepoint savepoint) throws SQLException;
 void releaseSavepoint(Savepoint savepoint) throws SQLException;
 Statement createStatement(int resultSetType,
           int resultSetConcurrency,
           int resultSetHoldability) throws SQLException;
}
```

9 LE ECCEZIONI

> Come è glorioso, e anche
> doloroso, essere un'eccezione.
>
> -Alfred de Musset

In Java esiste una classe di nome *Exception* e le eccezioni sono un' istanza di tale classe permettendoci così di trattarle come un qualsiasi oggetto Java. Quando un eccezione si verifica in un nostro programma , all'interno di un contesto di errore che abbiamo o è stato previsto, possiamo gestirla. Al contrario, se un'eccezione si verifica in un contesto imprevisto, non posiamo gestirla e quindi causerà l'uscita dal blocco di codice in cui si presenta. Se allo stesso tempo un blocco di codice più esterno l'ha prevista, potrà gestirla, altrimenti si uscirà dal programma stesso. Le eccezioni ricadono in due categorie, eccezioni controllate, in inglese *checked* e non controllate *uncheked*. Quando viene lanciata un'eccezione controllata, bisogna dire al compilatore cosa fare; il compilatore verifica attentamente che l'eccezione non venga ignorata. Al contrario non c'è alcuna necessità per il compilatore di tenere traccia di quelle non controllate. Vi è un principio da seguire nella gestione delle eccezioni: lanciare presto, catturare tardi.

9.1 LE ECCEZIONI DEL DBMS

Sono contenuti svariati punti nel dbms dove ha senso la creazione di un'eccezione. La maggioranza di questi si trova nei seguenti contesti:

1. l'analizzatore sintattico;
2. il parser;
3. la comunicazione tra client e server;
4. accesso al disco per la scrittura delle classi generate;
5. errori riguardanti il *DBMS* come la mancanza di permessi, funzionalità non implementate etc.;

Si è sentito il bisogno di definire una classe che utilizzasse il *design pattern* del *factory method*, per la creazione di specifiche eccezioni

Codice 20: Metodo lancio eccezioni analizzatore lessicale.

```
public void read() {
        sc.scanNext();
        token = sc.getToken();
        if (token.isMalformed)
                throw Error.parseError(ErrorCode.X_42582,
                                       "TOKEN SCONOSCIUTO",
                                        sc.getLineNumber());
}
```

durante l'esecuzione dell'applicativo. Riportiamo qui 24 una classe ridotta rispetto a quella sviluppata.

Quando si individua una condizione d'errore, il compito è semplice: lanciare (*throw*) un oggetto appropriato di tipo eccezione.

9.1.1 Le eccezioni dell'analizzatore sintattico

In tutti i casi dove la stringa in ingresso contiene un lessema non valido, cioè non verifica alcun *pattern*, si presenta il bisogno di creare un eccezione. Il metodo per il lancio delle eccezioni per lessemi non validi 20.

9.1.2 Le eccezioni del parser

Nel parser esistono molte più situazioni di errore, perchè vi sono un'infinità di modi di generare una stringa che non appartenga alla sua grammatica. Sono lanciate nelle condizioni di errore presentate in 6. I metodi che ne racchiudono alcune sono: 21.

9.1.3 Le eccezioni del driver JDBC

Tutte le eccezioni che vengono lanciate all'interno del *DBMS* devono essere restituite al programma *Client* che ha effettuato la richiesta. Nelle specifiche del *driver* sono definite le eccezioni per qualsiasi situazione di errore a cui si può andare incontro. L'insieme delle classi che formano il *package* delle eccezioni non sono i realtà le eccezioni generate effettivamente nel *DBMS* ma quelle da lanciare nel contesto del *driver*. Deve, pertanto, esistere una classe atta ad effettuare la trasformazione dal tipo di eccezione gestita all'interno del *DBMS* a quella corrispondente all'errore che ha avuto luogo così da rispettare quanto definito nel *driver*.

Codice 21: Metodi lancio eccezioni analizzatore sintattico.

```
void readThis(int tokenId) {
        if (token.tokenType != tokenId) {
                String required = Tokens.getKeyword(tokenId);
                throw Error.parseError(ErrorCode.X_42581,
                                       "TOKEN_INATTESO",
                                       sc.getLineNumber());
        }
        read();
}

void checkIdentifier() {
        if (token.tokenType != Tokens.LES_IDENTIFIER) {
                throw Error.parseError(ErrorCode.X_42581,
                                       "TOKEN_INATTESO",
                                       sc.getLineNumber());
        }
}
```

9.2 LA TRADUZIONE DELLE ECCEZIONI INTERNE

Consideriamo il caso di una richiesta fatta mediante il *driver jdbc*. Sia lanciata ad esempio un' eccezione perchè la stringa di ingresso non rispetta la grammatica. Il dbms gestisce questa evenienza inserendo all'interno della risposta l'oggetto di tipo eccezione appena creato. Sarà poi compito del *driver jdbc* tradurre l'eccezione interna nel corrispondente del *package java.sql.SQLException* relativo allo standard del *driver JDBC*. Riportiamo un esempio di metodo che effettua la traduzione 23 e l'interfaccia della classe che effettua la traduzione 22. Questa parte dell'applicativo serve per rispettare lo standard degli errori, un applicativo necessità, nel caso una funzionalità non sia supportata o vi siano degli errori di ricevere la giusta eccezione, altrimenti si potrebbe arrivare a un blocco dell'applicativo lato *client*. Per esempio durante lo sviluppo l'applicativo *Squirrel* non riusciva nella costruzione degli schemi per un'eccezione restituita in modo errato.

Codice 22: Classe di trasformazione eccezioni.

```
public interface Util {
        void throwError(MbmException e) throws SQLException{};
        SQLException sqlException(String msg, String sqlstate,
                                  int code, Throwable cause){};
        SQLException sqlException(int error){};
        SQLException sqlException(MbmException e){};
        SQLException notSupported(){};
        SQLException sqlExceptionSQL(int id){};
        SQLException sqlException(int id, String message){};
        SQLException nullArgument(String name){};
        SQLException connectionClosedException(){};
}
```

Codice 23: Un metodo dove avviene la trasformazione.

```
@Override
public ResultSet executeQuery(String sql) throws SQLException {
        RequestResponse response = null;
        RequestResponse request = new RequestResponse();
        request.sql = sql;
        request.idSession = connection.sessionClient.getIdSession();
        response = connection.sessionClient.execute(request);
        if (response.result.getException() != null)
                Util.throwError(response.result.getException());
        currentResultSet = new MbmSqlDbResultSet(response);
        return getResultSet();
}
```

Codice 24: *Factory Class* per le eccezioni.

```
public class Error {
  private static final String errPropsName = "sql-state-messages";
  private static final int bundleHandle =
                      BundleHandler.getBundleHandle(errPropsName, null);
  private static final int SQL_STATE_DIGITS = 5;
  private static final int SQL_CODE_DIGITS = 4;
  private static final int ERROR_CODE_BASE = 11;

  public static MbmException error(Throwable t, int code, String add) {
    String s = getMessage(code);
    if (add != null) {
      s += ": " + add.toString();
    }
    return new MbmException(t, s.substring(SQL_STATE_DIGITS + 1),
                            s.substring(0, SQL_STATE_DIGITS), code);
  }
  public static MbmException error(int code, String add) {
    return error((Throwable) null, code, add);
  }
  public static MbmException error(int code)
  {
    return error((Throwable) null, code, "");
  }
  public static MbmException parseError(int code, String add, int lineNumber) {
    String s = getMessage(code);
    if (add != null) {
      s = s + ": " + add;
    }
    if (lineNumber > 1) {
      add = getMessage(ErrorCode.M_parse_line);
      s = s + " :" + add + String.valueOf(lineNumber);
    }
    return new MbmException(null, s.substring(SQL_STATE_DIGITS + 1),
                            s.substring(0, SQL_STATE_DIGITS), code);
  }
  public static RuntimeException runtimeError(int code, String add) {
    MbmException e = error(code, add);
    return new RuntimeException(e.getMessage());
  }
  public static MbmException urlError(int code) {
    return new MbmException(null, "Errore url", "url", code);
  }

  public static String getMessage(final int errorCode) {
    return getResourceString(errorCode);
  }

  public static String getResourceString(int code) {
    String key = StringUtil.toZeroPaddedString(code, SQL_CODE_DIGITS,
                                               SQL_CODE_DIGITS);
    return BundleHandler.getString(bundleHandle, key);
  }
}
```

10 TEST

> Istruzioni sanguinose che,
> dopo essere state apprese,
> ritornano ad affliggere
> l'inventore.
>
> -Shakespeare

10.1 PERCHÈ TESTARE?

Le motivazioni che portano alla fase di test sono molteplici, prima fra tutte quella di avere un certo grado di sicurezza nella correttezza di quanto sviluppato. Risulta chiaro che lo sviluppo deve essere caratterizzato da vincoli sulla qualtà, robustezza, affidabilità e, da non sottovalutare, la chiarezza e la facilità di estensione a nuove funzionalità. L'esperienza maturata nello sviluppo ha cambiato radicalmente il peso della fase di test tanto da diventare una delle parti più importanti dello sviluppo, facendo nascere quello che viene definito sviluppo guidato dal test.

10.1.1 Sviluppo guidato dai test

La scrittura di software *test driven*([13]) pone forte enfasi sullo scrivere i test mentre si scrive il codice o, meglio ancora, prima di scrivere il codice. Si tratta di un particolare importante: scrivere i test successivamente, quando ormai si è sviluppata l'applicazione, diventa un'impresa decisamente faticosa e probabilmente non esente da errori. Scrivere poche righe di test durante lo sviluppo è molto più semplice e aiuta a modificare prima che sia troppo tardi parte dell'architettura. Il fatto che non si sappia scrivere un test poichè non si ha confidenza con il comportamento del codice evidenzia caratteristiche negative del codice scritto.

10.2 AMBIENTI DI TEST

Si può utilizzare un approccio al collaudo estremamente semplice: si realizza il test mediante il metodo *main*, il cui scopo è calcolare i valori e visualizzarli a video, assieme a quelli previsti. Gli ambienti

per il collaudo di unità sono stati progettati per valutare pacchetti di prove.

10.2.1 Test con JUnit

JUnit (vedi [10]) è un'ambiente che permette un netto salto di qualità rispetto al metodo decritto precedentemente. In linea di massima va progettata una classe ausilaria per ogni classe che viene sviluppata, più correttamente va realizzata per testare una specifica funzionalità che potrebbe essere anche trasversale a più classi. Esistono due versioni attualmente in uso *JUnit3* e *JUnit4*, ma, per il progetto è stata abbracciata inizialmente la versione quattro, per poi decidere di integrare ad esso l'utilizzo di *Spring*. Al contrario della versione tre la classe non deve derivare da alcuna, ed è possibile scegliere liberamente i nomi dei metodi. Da notare l'obbligo di contrassegnare i metodi di collaudo con una annotazione (***@Test***).

10.2.2 Test con Spring

Quanto offerto dal *framework* in questione non è un ambiente di collaudo alternativo, ma fornisce un'insieme di funzionalità atte ad ampliare le possibilità e la facilità del testing. Aiuta a razionalizzare ulteriormente tale fase fornendo all'avvio dell'unità di test uno specifico contesto inteso come collezione di oggetti *bean* (da non confondere con i *java bean*) da testare o comunque di supporto ad essi.

10.3 TESTARE L'RDBMS

Nello sviluppo del *dbms* è stato deciso di testare in modo approfondito i seguenti componenti:

1. il driver *jdbc*;
2. l'analizzatore lessicale;
3. l'analizzatore sintattico;
4. il metodo statico per la generazione delle classi a runtime;
5. la valutazione delle espressioni;

Altri componenti sono stati testati in modo più approsimativo, cioè implicitamente, perchè forniscono supporto ai componenti citati. Questa scelta cambierà perchè ogni componente deve avere la sua unità di collaudo, altrimenti esiste il rischio di controllare del codice corretto il cui comportamento sembra difforme dalle specifiche invece del codice in errore. Prima di affrontare il caso specifico del progetto

si devono indrodurre alcuni concetti utilizzati, mirati a rendere più leggibile la fase di collaudo.

10.3.1 Design Pattern DAO

Lo scopo del *pattern DAO* o, anche *Data Access Object* (vedi [17]) è di disaccopiare la logica di business dalla logica di accesso ai dati. Tutto ciò si ottiene spostanto la logica di accesso ai dati dai componenti di business ad una classe *DAO*. Ciò apparentemente sembra provocare un maggior lavoro per lo sviluppatore, ma in realtà offre due grandi vantaggi descritti di seguito:

1. nelle applicazioni lavorate soltanto con l'interfaccia *DAO* l'implementazione concreta rimane nascosta. Ciò implica una manutenzione delle applicazioni più facile, poichè la classe di accesso ai dati concreti può essere modificata senza che il codce applicativo debba essere rivisto;

2. gli eventuali accessi al *database* sono mediati da una classe centrale. Se il modello dei dati nel *database* cambia, l'adattamento riguarderà soltanto questa classe centrale: le chiamate ai metodi nei vari elementi dell'applicazione non richiederanno modifiche;

10.3.2 ORM

L'*ORM* (*Object Relational Mapping*) è implementato da vari *framework* (tra cui hibernate) e consiste nel mappare le entità viste nelle programmazione ad oggetti (come classi) alle tabelle del *database* relazionale. Molti soluzioni che implementano questo *mapping* forniscono, oltre ad un linguaggio di *query* alternativo, la generazione in automatico di istruzioni in *SQL* per svolgere i compiti principali di manipolazione della base di dati.

10.3.3 Il codice di collaudo

Il codice che sarà presentato in questo breve paragrafo va inteso come esempio di quanto fatto per lo sviluppo vero e proprio. Sono state sviluppate due classi per il collaudo del *driver jdbc*, dove una utilizza i metodi direttamente, l'altra sfrutta sia il *DAO* che l'*ORM* per mascherare le chiamate in modo diretto. Dato il forte legame con il *software* libero *HSQLDB*, per metterlo alla prova si è scelto di usare come *framework* per l'*ORM Hibernate*. Per quanto concerne le altre componenti da testare si è ritenuto sufficiente la creazione (per ognuno) di una classe di collaudo con *JUnit* senza l'aggiunta di altri componenti.

Spring e Hibernate un caso pratico

Spring permette di approntare l'ambiente adeguato all'esecuzione di *Hibernate* (non è l'unico modo) mettendone a disposizione le risorse. La *factory* di *Spring* attraverso le informazioni definite in un file *xml*, può gestire la localizzazione e la configurazione della *SessionFactory* di *Hibernate* e del *DataSource* utilizzato anche dalla *SessionFactory*. Mostriamo in 25 un esempio di configurazione.

Viene innanzitutto definito il *bean dataSource*, fornendo tutte le informazioni per la connessione e poi la *bean sessionFactory*. Quest'ultima è configurata specificando i vari file di *mapping*, la tipologia del database utilizzata nell'esempio è specificato:

org.hibernate.dialect.HSQLDialect.

HibernateTamplate è una classe che implementa il *pattern Template Method* le cui funzioni sono:

1. fornisce accesso alla sessione di *Hibernate*;
2. si assicura che la *session* sia correttamente aperta e chiusa;
3. viene utilizzato per le transazioni;
4. fornisce tutte le fnzionalità standard di accesso ai dati;
5. i metodi di cui sopra delegano all'implementazione di determinate interfacce le logiche di persistenza;

In questo modo l'implementazione di un generico *DAO* può utilizzare la classe *HibernateTemplate* per eseguire le comuni operazioni. Mostriamo l'esempio più semplice, basato sull'entità la cui riga è un'istanza (*JavaBean*) della classe *ExampleRecord*. Si hanno ora tutti gli strumenti per creare una classe di collaudo con *Spring* per testare il funzionamento dei metodi implementati in questo specifico *DAO*. Ovviamente quanto fatto si può ripetere per qualunque tabella (anche quelle di sistema), così, durante lo sviluppo, si ha l'opportunità di verificare che le ultime modifiche non abbiano rovinato quanto già funzionante.

Codice 25: Configurazione di *Hibernate* con *Spring*.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
<!-- services -->

 <bean id="dataSource"
    class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="it.mbm.sqldb.jdbc.MyJDBCDriver" />
    <property name="url" value="jdbc:mbmsqldb:rmi://localhost:6000/MBMDB" />
    <property name="username" value="" />
    <property name="password" value="" />
 </bean>

 <bean id="sessionFactory"
        class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="mappingResources">
      <list>
        <value>it/mbm/sqldb/resources/examplerecord.hbm.xml</value>
      </list>
    </property>
    <property name="hibernateProperties">
      <props>
        <prop key="hibernate.dialect">
                              org.hibernate.dialect.HSQLDialect
                              </prop>
        <prop key="hibernate.show_sql">true</prop>
        <prop key="hibernate.hbm2ddl.auto">update</prop>
      </props>
    </property>
  </bean>

  <bean id="transactionManager"
 class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory" />
  </bean>

  <bean id="hibernateTemplate"
                class="org.springframework.orm.hibernate3.HibernateTemplate">
        <property name="sessionFactory" ref="sessionFactory" />
  </bean>

  <bean id="exampleRecordDao"
        class="it.mbm.sqldb.dao.ExampleRecordHibernateDaoSupport">
    <property name="hibernateTemplate" ref="hibernateTemplate" />
  </bean>
</beans>
```

Codice 26: Esempio *DAO* con *Hibernate*.

```
public class ExampleRecordHibernateDaoSupport
 extends HibernateDaoSupport implements ExampleRecordDao{
        @Override
        public void insert(ExampleRecord exampleRecord) {
                // TODO Auto-generated method stub

        }
        @Override
        public void update(ExampleRecord exampleRecord) {
                // TODO Auto-generated method stub
        }
        @Override
        public void delete(int id) {
                // TODO Auto-generated method stub

        }
        @Override
        public List<ExampleRecord> findAllExampleRecord() {
                HibernateTemplate ht = getHibernateTemplate();
                List<ExampleRecord> erList =
                        ht.find("from␣ExampleRecord");
                return erList;
        }
        @Override
        public int exampleRecordCount() {
                // TODO Auto-generated method stub
                return 0;
        }

}
```

11 CONCLUSIONI

> Perchè non esiste un lavoro sulla terra che il mendicante non conosca o non faccia.
>
> -Kipling

11.1 SVILUPPI FUTURI

Rispetto a un progetto più libero da vincoli temporali, la prima versione testata dell'applicativo, di solito, possiede tutte le funzionalità dei *software* simili. Vista l'impossibilità, almeno da parte mia, di sviluppare un intero *DBMS* nella sua completezza, molte parti, principali, sono state rimandate. Possiamo pertanto dividere ciò che dovrà essere affrontato in futuro in due insiemi.

1. **componenti assenti nel *DBMS***. linguaggio *ddl*, vincoli di integrità, transazioni, trigger, ottimizzazione del codice e aumento del numero di tipi di dato riconosciuti;

2. **funzionalità avanzate**. raggiungimento di standard *SQL* più recenti, aggiunta di vari protocolli di comunicazione;

Bisogna precisare che le funzionalità sviluppate del linguaggio *DML* sono un sottoinsieme che può essere chiamato *DQL*, cioè con funzionalità esclusive di reperimento. Il passo immediatamente successivo è, appunto, la realizzazione delle parti mancati, per comprendere tutto il *DML*.

11.1.1 Linguaggio *DDL*

Fino ad ora la mancanza di questa caratteristica può sembrare un paradosso, ma non lo è perchè le tabelle pricipali da interrogare sono prese dal *software* di pianificazione. Viene ora preclusa la possibilità di definire viste, nuove tabelle ed altre funzionalità che risultano essenziali.

11.1.2 Indici

In un *DBMS* moderno ci sono almeno i seguenti tre modi di accedere ai dati di un *database*.

1. **accesso sequenziale**;
2. **accesso con indice**;
3. **accesso con hash**;

Il primo metodo già sviluppato presenta grossi problemi di prestazioni. Supponiamo ad esempio che sia richiesta una ricerca, questa, nel modo sequenziale impiegherebbe un tempo lineare cioè $O(n)$ perchè, in media, dovremmo scorrere metà tabella. Risulta evidente che è estremamente inefficiente, si dovrebbe arrivare a una complessità di tipo logaritmico. Ecco quindi che sono stati introdotti degli oggetti atti, appunto, a ridurre questi tempi: gli indici. Un indice è un oggetto che fa parte integrante di una base di dati e che viene associato ad una delle sue tabelle rispetto ad uno o più attributi che costituiscono una chiave di ricerca. Se gli attributi sono i medesimi di quelli della chiave primaria (vedi B) si parla di indice primario, altrimenti di indice alternativo (secondario) (definizione usata nei *DBMS*). Gli indici non necessariamente contengono una superchiave, pertanto potrebbe venir meno la proprietà di univocità (vedi B). Il caso dell'indice primario si presta bene a rispondere alla seguente domanda:

quale tupla è associata alla chiave k ?

Risulta conveniente in altri termini consultare l'indice: esso ci dirà dove si trova la tupla in una frazione di tempo rispetto alla ricerca sequenziale. Ad una tabella possono ovviamente essere associati più indici, ma tutti devono necessariamente essere aggiornati quando viene effettuata un'operazione di modifica della tabella corrispondente. La strutura dati analizzata per questa funzione è una variante dell'albero di ricerca binaria chiamato *AVL* (vedi [7]) perchè possiede una propietà aggiuntiva che permette di effettuare la ricerca (nel caso peggiore) in $O(\log(n))$. Gli indici non sono utili solo alla ricerca ma anche a ridurre considerevolmente il tempo medio necessario ad eseguire l'operazione di *join*. Ad esempio nel caso due tabelle abbiano dimensioni sostanziali (dell'ordine di decine di migliaia) l'esecuzione di un prodotto cartesiano richiederebbe un tempo molto elevato. Gli indici sono uno strumento essenziale senza i quali il *throughput* delle query sarebbe insufficiente all'utilizzo nell'ambito *businnes*.

11.1.3 Procedure permanenti e trigger

Ormai risulta chiaro che l'idea di vedere *SQL* come un sottolinguaggio per l'accesso ai dati che può solo essere ospitato in diver-

si linguaggi di programmazione o essere usato con delle limitazioni computazionali (vedi C) sia stato un grave errore di valutazione. Dallo standard *SQL3* hanno arricchito il linguaggio di manipolazione dei dati fino a farlo diventare un vero linguaggio di programmazione. Secondo quanto detto lo schema di una base dati non sarebbe più costituito solo da schemi relazionali e vincoli di integrità ma anche da procedure e funzioni raccolte in moduli. I sottoprogrammi contenuti nei moduli si chiamano procedure permanenti, in inglese *stored procedures* giocando inoltre un ruolo essenziale ai fini dell'efficienza. In quanto sviluppato è stata prevista anche questa funzionalità ma non è ancora stata sviluppata. Assieme a questo molti dialetti (tra cui *HSQLDB*) permettono la possibilità di associare dei *trigger*, una tecnica per indicare determinati tipi di regole attive. I trigger sono utilizzati per diversi scopi nella progettazione di un database, e principalmente:

1. per mantenere l'integrità dei dati della singola tabella;
2. per mantenere l'integrità referenziale tra le varie tabelle;
3. per monitorare i campi di una tabella ed eventualmente generare eventi specifici;

Ho utilizzato una programmazione *top down*, in modo che si possano, per quanto possibile, implementare in futuro.

11.1.4 Transazioni

Le transazioni ovvero le unità logiche svolte da un'applicazione sono una delle caratteristiche principali di un *database* relazionale. Sono, infatti, quel meccanismo che consente di mantenere, durante la vita della nostra base dati, tutte le nostre informazioni consistenti. Una volta completato il linguaggio *DML* in modo da permettere anche l'inserimento si dovranno realizzare tutti i livelli di transazione presentati in 8. Come spiegato precedentemente durante lo sviluppo va mantenuto un occhio di riguardo a quello che sarà fatto successivamente. Nel codice le tuple sono, per così dire, incapsulate dentro un'altra classe definita proprio per gestire una riga, memorizzando tutte le informaioni riguardo le transazioni in atto. Quanto appena detto non modifica, assolutamente, il funzionamento di ciò che è stato spiegato nei capitoli precedenti, è stato tralasciato per non pesare sulla comprensione degli algoritmi etc.

11.1.5 Protocolli di comunicazione

Abbiamo già anticipato la possibilità di aggiungere altri protocolli di comunicazione. Per quanto *Java RMI* sia risultato utile alcune

circostanze potrebbero portare alla sua sostituzione con *CORBA* data, come già detto, la sua maggiore flessibilità in ambienti eterogenei. Potrebbe, in aggiunta, risultare utile utilizzare il classico protocollo *http*. Non è giusto nascondere che queste scelte siano intimamente legate con quella di generare delle classi per così dire al volo. Nel caso ad esempio si realizzi una modifica al driver *jdbc* per poter utilizzare un altro protocollo si avrebbe la necessità di risolvere nuovamente il problema delle classi che non sono presenti lato client. Una soluzione banale a questo problema è quella di eseguire la copia della tabella in una tabella formata da strutture dati note. Un'altra, invece, di eliminare il concetto delle classi utilizzate tutt'oggi. Non è ancora stata presa una decisone riguardo questi argomenti.

11.2 L'ANALISI DELLE PRESTAZIONI ?

L'analisi delle prestazioni completa, di solito parte fondamentale dello sviluppo, è stata rimandata per dei motivi molto semplici. Il principale riguarda l'impossibilità di implementare, in alcuni casi, algoritmi efficienti, i quali potranno essere integrati soltanto nel momento in cui lo sviluppo sarà in uno stadio più avanzato. Nei capitoli precedenti sono state affrontate solo quelle parti che hanno già la struttura definitiva come: il calcolo delle espressioni, il prodotto cartesiano e l'operatore *like*. Di fondamentale importanza nelle prestazioni è la capacità del *parser* di generare le interrogazioni nel miglior modo possibile. Realizzare d'altro canto un *parser* ottimo è molto difficile, anzi, se per ottimo si intende che sia in grado di eseguire sempre la *query* ottima data una in ingresso, ci si scontra con un problema *NP*-completo. Il parser pertanto subirà pesanti variazioni atte al miglioramento, non solo delle sue funzionalità, ma anche nel creare interrogazioni il più possibile vicine a quelle ottime. Rimarrà sempre, in realtà, la responsabilità dell'utente in questo contesto. Ad esempio se l'utente fornisce una interrogazione già efficiente questo, chiaramente, facilita il compito del parser e si ha la certezza di ottenere dei tempi di risposta soddisfacenti. Il secondo motivo è come spesso accade dovuto allla mancanza cronica di tempo. Le funzionalità da implementare sono molte e si è deciso di dare più importanza al numero effettivo di queste sviluppate che alla loro efficienza. Un unico fatto può, in parte, alleviare il peso di questa scelta: di essersi basati per molte scelte su quanto già sviluppato nel *DBMS* HSQLDB.

A LINGUAGGI

Questa appendice presenta una breve introduzione ai linguaggi formali utili per la comprensione dell'analizzatore lessicale, sintattico e per le scelte di progetto fatte. I linguaggi trattati sono di due tipi:

1. regolari;
2. liberi dal contesto;

Definizione linguaggio A.0.1. Sia L un insieme. Allora

$$L \subseteq \Sigma^* \Rightarrow L \text{ è un linguaggio}$$

A.1 LINGUAGGI REGOLARI

I linguaggi regolari sono quelli che possono essere descritti mediante gli automi a stati finiti o equivalentemente con le espressioni regolari. Ci sono tre tipi di automi a stati finiti:

1. automi a stati finiti deterministici o DFA;
2. automi a stati finiti non deterministici o NFA;
3. automi a stati finiti non deterministici con epsilon transizioni o $\epsilon - \text{NFA}$

Definizione DFA A.1.1. Si dice automa a stati finiti deterministico una quintupla $A = (Q, \Sigma, \delta, q_o, F)$ dove:

1. Q è un insieme di stati;
2. Σ è un insieme di simboli detto anche alfabeto;
3. δ è una funzione di tansizione;
4. q_o è lo stato iniziale;
5. $F \subseteq Q$ è l'insieme degli stati finali o accettanti;

La funzione δ deve avere questa forma

$$\begin{aligned} \delta \colon Q \times \Sigma &\to Q \\ q &\mapsto \delta(q, \sigma) \end{aligned}$$

Per poter consumare più simboli conviene introdurre una nuova funzione in tale modo:

Definizione DFA A.1.2. Definiamo una funzione $\hat{\delta}\colon Q \times \Sigma^* \to Q$ per cui

1. $\hat{\delta}(q, \epsilon) = q$ e
2. $\forall w \in \Sigma^*$ e $a \in \Sigma$,

$$\hat{\delta}(q, wa) = \delta(\hat{\delta}(q, w), a)$$

Linguaggio DFA A.1.3. Il linguaggio di un DFA $A = (Q, \Sigma, \delta, q_0, F)$ è definito da

$$L(A) = \{w | \hat{\delta}(q_0, w) \in F\}$$

Definizione NFA A.1.4. Si dice automa a stati finiti non deterministico una quintupla $A = (Q, \Sigma, \delta, q_o, F)$ dove:

1. Q è un insieme di stati;
2. Σ è un insieme di simboli detto anche alfabeto;
3. δ è una funzione di transizione;
4. q_o è lo stato iniziale;
5. $F \subseteq Q$ è l'insieme degli stati finali o accettanti;

La funzione δ deve avere questa forma

$$\begin{aligned} \delta\colon Q \times \Sigma &\to \mathcal{P}(Q) \\ q &\mapsto \delta(q, \sigma) \end{aligned}$$

Anche in questo caso la funzione δ può essere estesa a una funzione $\hat{\delta}$ che mappa $Q \times \Sigma^*$ a $\mathcal{P}(Q)$ in questo modo:

1. $\hat{\delta}(q, \epsilon) = q$ e
2. $\hat{\delta}(q, wa) = \{p : \exists\, r \in \hat{\delta}(q, w), p \in \delta(r, a)\}$

Linguaggio NFA A.1.5. Il linguaggio di un NFA $N = (Q, \Sigma, \delta, q_0, F)$ è definito da

$$L(N) = \{w | \hat{\delta}(q_0, w) \cap F \neq \emptyset\}$$

L'automa apparentemente più potente è $\epsilon - NFA$ che ammette transizioni sulla stringa vuota. È come se l'NFA potesse compiere una transizione spontaneamente, senza aver ricevuto alcun simbolo di input.

Definizione ϵ-NFA A.1.6. Si dice automa a stati finiti non deterministico con epsilon transizioni una quintupla $A = (Q, \Sigma, \delta, q_0, F)$ dove:

1. Q è un insieme di stati;
2. Σ è un insieme di simboli detto anche alfabeto;
3. δ è una funzione di transizione;
4. q_0 è lo stato iniziale;
5. $F \subseteq Q$ è l'insieme degli stati finali o accettanti;

La funzione δ deve avere questa forma

$$\begin{gathered}\delta \colon Q \times \Sigma \cup \{\epsilon\} \to \mathcal{P}(Q) \\ q \mapsto \delta(q, \sigma)\end{gathered}$$

Per fare quanto fatto nei due casi precedenti vi è la necessità di definire la ϵ chiusura di un insieme P di stati. Noi utilizziamo $\epsilon - \text{CLOSURE}(q)$ per identificare l'insieme dei vertici p per cui esiste un percorso da q a p di sole ϵ. Naturalmente viene spontaneo definire il caso più generale così:

$$\epsilon - \text{CLOSURE}(P) = \bigcup_{q \in P} \epsilon - \text{CLOSURE}(q)$$

Si può ora definire $\hat{\delta}$ nel seguente modo:

1. $\hat{\delta}(q, \epsilon) = \epsilon - \text{CLOSURE}(q)$;
2. $\forall w \in \Sigma^* \wedge a \in \Sigma, \hat{\delta}(q, wa) = \epsilon - \text{CLOSURE}(P)$, dove $P = \{p : \exists r \in \hat{\delta}(q, w), p \in \delta(r, a)\}$

Linguaggio ϵ-NFA A.1.7. Il linguaggio di un ϵ-NFA $N = (Q, \Sigma, \delta, q_0, F)$ è definito da

$$L(N) = \{w | \hat{\delta}(q_0, w) \cap F \neq \emptyset\}$$

Ora un modo di definire i linguaggi regolari utile nell'ambito della ricerca di testi o come componenti di un compilatore. Le espressioni regolari sono sostanzialmente degli automi a stati finiti non deterministici. Risulterà chiara questa affermazione più avanti.

Espressioni regolari A.1.8. Sia Σ un'alfabeto. Le espressioni regolari su Σ sono definite induttivamente nel seguente modo:

1. $\emptyset$ è un'espressione regolare e denota il linguaggio vuoto;
2. ϵ è un'espressione regolare e denota il linguaggio $\{\epsilon\}$;

3. per ogni $a \in \Sigma$, **a** è un'espressione regolare e denota il linguaggio $\{a\}$;

4. se **r** e **s** sono espressioni regolari che denotano i linguaggi **R** ed **S** rispettivamente, allora $(\mathbf{r} + \mathbf{s})$,**rs** e $(\mathbf{r}^*)$ sono espressioni regolari che denotano rispettivamente i linguaggi $R \cup S$, RS, R^*;

Teorema linguaggi regolari A.1.9. *Sia L un linguaggio riconosciuto da uno qualsiasi degli automi o espressioni regolari. Allora per ognuno degli altri esiste un automa o espressione regolare che lo riconosce.*

A.2 LINGUAGGI LIBERI DAL CONTESTO

I linguaggi liberi dal contesto sono una classe più ampia di quelli regolari. Al posto degli automi a stati finiti e delle espressioni regolari ci sono automi a pila e grammatiche libere dal contesto. Tali descrizioni vengono usate spesso nei compilatori proprio perchè i linguaggi di programmazione rientrano in questa categoria.

A.2.1 Grammatihe libere dal contesto

Grammatiche libere dal contesto A.2.2. Una grammatica libera dal contesto è una quadrupla $G = (V, T, P, S)$, dove V e T sono insiemi finiti rispettivamente di variabili e terminali. Si assume per semplicità che $V \cap T = \emptyset$. P è un insieme finito di produzioni della forma $A \to \alpha$, dove A è una variabile e α è una stringa di $(V \cup T)^*$. S è una variabile speciale chiamata simbolo iniziale.

Per poter definire il linguaggio definito da una grammatica è necessario innanzitutto introdurre due relazioni. Se $A \to \beta$ è una produzione di P e α, γ sono stringhe di $(V \cup T)^*$, allora $\alpha A \gamma \Rightarrow \alpha\beta\gamma$. Si supponga siano $\alpha_1, \alpha_2, ..., \alpha_m$ stringhe in $(V \cup T)^*$, $m \geqslant 1$ e

$$\alpha_1 \Rightarrow \alpha_2, \alpha_1 \Rightarrow \alpha_2, ..., \alpha_{m-1} \Rightarrow \alpha_m$$

Allora si dice che $\alpha_1 \overset{*}{\Rightarrow} \alpha_m$ oppure α_1 deriva α_m in G.

Linguaggio CFG A.2.3. Il linguaggio generato da G $L(G)$ è così definito:

$$L(G) = \{w | w \in T^* e S \overset{*}{\Rightarrow} w\}$$

Molto utile nei parser è il concetto di albero sintattico. Data una stringa di una grammatica libera dal contesto (non ambigua) permette di associarle una struttura univoca, importante per la sua analisi.

Albero sintattico A.2.4. Sia G una grammatica libera dal contesto. Un albero si dice sintattico per G se :

1. ogni vertice ha un etichetta in $V \cup T \cup \epsilon$;

2. l'etichetta del nodo radice è S;
3. se un vertice è interno e ha etichetta A, allora A deve essere in V;
4. se n ha etichetta A e i vertici $n_1, n_2, ..., n_k$ sono i suoi figli in ordine da sinistra, con etichette rispettivamente $X_1, X_2, ..., X_k$, allora
$$A \to X_1 X_2 ... X_k$$
deve essere una produzione in P;
5. se un vertice n ha etichetta ϵ, allora n deve essere una foglia e inoltre, essere l'unico figlio del padre;

Relazione tra produzioni e alberi A.2.5. *Sia* $G = (V, T, P, S)$ *una CFG. Allora*
$$S \overset{*}{\Rightarrow} \alpha \Leftrightarrow \exists \, \mathrm{un\ albero\ sintattico\ con\ resa}\ \alpha$$

Ambiguità

Se ad ogni passo di una derivazione è scelta la produzione della variabile più a sinistra, allora la derivazione è detta derivazione sinistra (simmetrico per la destra). Questa definizione sarà utile per l'ambiguità.

Grammatica ambigua A.2.5.1. Una CFG dove qualche parola ha più di un albero sintattico è detta ambigua.

Linguaggio ambiguo A.2.5.2. Un linguaggio riconosciuto solo da grammatiche ambigue si dice inerentemente ambiguo.

Vale la pena fare in questo contesto delle precisazioni. In sostanza può essere che un linguaggio sia espresso mediante una grammatica ambigua ma, se esiste una grammatica non ambigua che lo riconosce a sua volta, non è un problema. L'ambiguità è grave quando non esiste un modo di descrivere il linguaggio per eliminare l'ambiguità. Pertanto è intrinsecamente ambiguo.

ESPRIMERE L'AMBIGUITÀ

Le derivazioni non sono uniche, anche in grammatiche non ambigue, ma come diremo nel teorema in grammatiche non ambigue sono uniche le derivazioni sinistre e destre.

Legame ambiguità derivazioni sinistre e destre A.2.5.2.1. *Per ogni grammatica* $G = (V, T, P, S)$ *e per ogni stringa* $w \in T^*$, *w ha due alberi sintattici distinti se e solo se ha due distinte derivazioni a sinistra(destra) da S.*

Per approfondire quanto riportato, si consiglia la lettura di [18].

B | IL MODELLO RELEAZIONALE

In questo paragrafo ci occuperemo del modello usato per la fase di progettazione logica. Il *DBMS* sviluppato si basa essenzialmente su questo modello logico.

B.1 RELAZIONI E TUPLE

Il modello relazionale (vedi [2]) prevede un unico meccanismo di strutturazione dei dati costituito, appunto, dalla relazione. Questa va intesa in senso matematico, un concetto presente in molti testi elementari di matematica.

Definizione relazione B.1.1. Una relazione R su una sequenza di insiemi (domini) $D_1, D_2, ..., Dn$ è un sottoinsieme del loro prodotto cartesiano:

$$R \subseteq D_1 \times D_2 \times ... \times D_n$$

A differenza delle relazioni matematiche, nel modello relazionale, ad ogni dominio di una relazione viene associato un nome, detto attributo (o campo). Gli elementi di una relazione sono delle *ennuple*, meglio note come tuple. Si osservi che, coerentemente con la definizione data non vi possono essere duplicati. Nella realtà dei *DBMS* le relazioni vengono viste come multinsiemi per ovviare a tale vincolo. Come vedremo, una base di dati relazionale è un insieme di tabelle.

B.2 CHIAVI DI UNA RELAZIONE

Il modello relazionale è molto limitato nella possibilità di esprimere vincoli. Il più importante tra questi è sicuramente il vincolo di chiave. Il concetto di chiave può essere espresso formalmente come segue:

Definizione chiave B.2.1. Una chiave K di una relazione R è un insieme $K = \{x_1, x_2, ..., x_n\} \neq \emptyset$ di nomi di attributi di R che gode delle sequenti propietà:

1. **Univocità**. data una qualsiasi estensione di K, cioè un'assegnazione di valori agli attributi di K, esiste al massimo una tupla di R che ha tali valori in corrispondenza degli attributi di K;

2. **Non-ridondanza**. la propietà precedente non vale più nel caso si tolga uno qualsiasi degli attributi di K;

In altre parole, gli attributi di una chiave sono tutti necessari e sufficienti per identificare una tupla di una relazione. Un insieme di attributi che verifica la prima propietà viene detto superchiave della relazione, qundi una chiave è una superchiave ma non è vero necessariamente il contrario. Chiaramente una relazione può essere caratterizzata da più chiavi, fra queste ne viene scelta una, generalmente con il minor numero degli attributi chiamata chiave primaria. Per questioni di efficienza spesso si fa in modo che la chiave primaria sia costituita da un solo attributo e che sia a bassa occupazione di memoria. Per definire, invece, l'integrità referenziale (vincolo tra due relazioni) definiamo dapprima il concetto di chiave esterna vedi [3].

Definizione chiave esterna B.2.2. Un insieme di attributi FK nello schema di relazione R_1 è una chiave esterna di R_1 che riferisce la relazione R_2 se soddisfa queste due regole:

1. gli attributi presenti in FK hanno gli stessi domini degli attributi di chiave di R_2; si dice che gli attributi FK riferiscono o fanno riferimento alla relazione R_2;
2. un valore di FK in una tupla t_1 dell'estensione corrente di R_1 o è presente come valore di K di R_2 in una tupla t_2 nell'estensione corrente di R_2 o è nullo. Nel primo caso si ha $t_1[FK] = t_2[K]$ e si dice che la tupla t_1 riferisce o fa riferimento alla tuple t_2.

Un vincolo di integrità rferenziale consite solo nel dire che ogni estensione dello schema di relazione R_1 deve avere come chiave esterna rispetto a R_2 (ad esempio) un'insieme di attributi FK. Anche in questo caso possono esistere più vincoli di chiave esterna.

B.3 SCHEMI DI RELAZIONE, SCHEMI RELAZIONALI E BASI DI DATI RELAZIONALI

L'intensione di una relazione è lo schema di relazione costituito dai seguenti componenti:

1. elenco degli attributi della relazione, con il loro tipo;
2. l'insieme degli attributi che costituisce la chiave primaria;
3. l'insieme delle chiavi esterne;
4. il nome della relazione;

Siamo adesso in grado di definire lo schema relazionale.

Definizione schema relazionale B.3.1. Uno schema relazionale è una collezione di schemi di relazione.

Definizione base di dati relazionale B.3.2. Una base di dati relazionale è un insieme di relazioni le cui occorrenze sono estensioni di uno schema relazionale.

B.4 ALGEBRA RELAZIONALE

Il modello qui accennato è stato introdotto pensando di manipolare in modo elegante i dati da un punto di vista matematico. Sono stati proposti due approcci:

1. **algebrico**. in cui il risultato di un'interrogazione è una relazione ottenuta mediante l'utilizzo degli operatori dell'algebra relazionale;
2. **logico**. in cui il risultato di un interrogazione viene descritta come la relazione che soddisfa una formula del linguaggio logico (noto come calcolo relazionale);

Entrambi questi approcci sono equivalenti in riferimento all'espressività, cioè per un'interrogazione fatta in un approccio ne esiste una per l'altro.

B.4.1 Operazioni elementari

Qui viene proposta una breve descrizione delle operazioni più comuni dell'algebra relazionale. Partiamo dall'unione.

Definizione unione di relazioni B.4.2. Siano R ed S due relazioni compatibili (definite, in ordine, sugli stessi attributi), l'unione di R con S è la relazione ottenuta dall'unione insiemistica delle due relazioni:

$$R \cup S = \{t | \, t \in R \vee t \in S\}$$

Definizione differenza di relazioni B.4.3. Siano R, S due relazioni compatibili, la differenza fra R ed S è la relazione ottenuta dalla differenza insiemistica delle due relazioni:

$$R - S = \{t | \, t \in R \wedge t \notin S\}$$

Definizione proiezione di una relazione B.4.4. Questa operazione permette di selezionare solo determinate colonne da una tabella. Sia data una relazione R ed un sottoinsieme $A = a_1, a_2, ..., a_k, |A| = k$ dei suoi attributi, si definisce proiezione R su A la relazione di grado

k che si ottiene da R tralasciando le colonne (attributi) non presenti in A. Le eventuali nuove tuple duplicate vengono elimintate (nel modello stretto). Attualmente tale operazione viene indicata nel seguente modo.

$$\pi_{a_1,a_2,\ldots,a_k}(R)$$

Definizione restrizione di una relazione B.4.5. L'operazione di restrizione, detta anche selezione, consente di estrarre delle tuple che rispettano una determinata condizione (o predicato) P. Viene indicata :

$$\sigma_P(R)$$

B.4.6 Operazioni composte

Tra le operazioni che è possibile derivare da quelle appena citate, quelle che rivestono maggiore utilità sono quelle di giunzione. Sono considerate le operazioni che più rendono il modello relazionale utile. Ciò che è possibile fare con tali operazioni in altri linguaggi sono espressi mediante algoritmi, con un pesante uso di puntatori.

Definizione Equigiunzione B.4.7. L'equigiunzione (*equijoin*) di due relazioni R, S, rispetto ad un attributo A di R e B di S, è così definita:

$$R \underset{A=B}{\bowtie} S = \sigma_{A=B}(R \times S)$$

Viene per prima cosa eseguito il prodotto cartesiano per poi mantenere solo quelle righe in cui risultano uguali i due attributi A e B. L'equigiunzione è un caso particolare di una operazione più generale detta θ-join dove θ è un operatore di confronto ($<, >, =$, etc.). A sua volta il θ-join è un caso particolare di giunzione interna (*inner join*), dove qui il criterio di selezione è una qualsiasi condizione.

B.4.8 Le giunzioni esterne

Si è sentita la necessità di introdurre delle nuove giunzioni che permettano di preservare nel risutato tutte le informazioni presenti nelle tabelle di partenza. I campi per il quale, a ragione, non esiste il valore vengono riempiti con il valore *null* (ricordiamo che *null* possiede più significati).

B.5 CALCOLO RELAZIONALE

Molti linguaggi commerciali e, in parte, lo stesso *SQL* si basano sul calcolo relazionale piuttosto che sull'algebra relazionale. È un linguaggio di interrogazione dichiarativo: specifica le proprietà del

risultato dell'interrogazione. Esistono due versioni del calcolo relazionale:

1. calcolo relazionale sulle tuple (SQL);
2. calcolo relazionale sui domini (QBE);

Un'espressione generale del calcolo relazionale su tuple ha la seguente forma:

$$\{t_1.A_j, t_2.A_k, ..., t_n.A_m \mid COND(t_1, t_2, ..., t_n, t_{n+1}, ..., t_{n+m})\}$$

dove $t_1, t_2, ..., t_n, t_{n+1}, ..., t_m$ sono variabili di tupla, ogni A_i è un attributo della relazione su cui il suo t_u prende valori e COND è una condizione del calcolo relazionale su tuple.
Il calcolo su domini differisce dal calcolo su tuple per il tipo di variabili usato nelle formule: anzichè avere alcune variabili che prendono valori sulle tuple, le variabili prendono valori singoli sui domini degli attributi. Un'espressione del calcolo su domini è quindi del tipo:

$$\{x_1, x_2, ..., x_n \mid COND(x_1, x_2, ..., x_n, x_{n+1}, ..., x_{n+m})\}$$

Come si può facilmente notare, entrambi i metodi sono dichiarativi in quanto specificano il risultato di un'interrogazione senza indicare come produrlo. È un modo completamente diverso rispetto all'algebra relazionale, molto lontano anche dai linguaggi di programmazione usati più comunemente.Tra i due tipi di calcolo relazionale quello che ha preso più piede è sicuramente quello sulle tuple, infatti SQL può essere, proprio, considerato tale.

B.6 FORME NORMALI

Un approccio troppo semplicistico nello sviluppo di una base di dati basata sul modello relazionale porta ad avere degli schemi che risultano inefficienti nei confronti dello spazio occupato e che presentano delle difficoltà sulle operazioni di modifica della base di dati. Un primo parametro, non formale, per capire se una base di dati è stata sviluppata con criterio è la facilità con cui è possibile esprimere le interrogazioni. Chiaramente è qualcosa da prendere con le molle perchè, facile, non è un termine adatto in un contesto matematico, potrebbe esistere una *query* difficile per sua natura. È consuetudine riferirsi a questa situazione dicendo che lo schema non è normalizzato. Prima di definire le forme normali più spesso usate in letteratura bisogna introdurre le dipendenze funzionali.

B.6.1 Dipendenze funzionali

Di fondamentale importanza nel modello relazionale è il concetto di dipendenza funzionale.

Definizione dipendenza funzionale B.6.2. Siano $X = \{X_1, X_2, ..., X_k\}$ e $Y = \{Y_1, Y_2, ..., Y_n\}$ due insiemi di attributi di uno schema di relazione R. Diremo che Y dipende funzionalmente da X nel seguente modo:

$$X_1, X_2, ..., X_k \rightarrow Y_1, Y_2, ..., Y_n$$

se e solo se, per tutte le tuple di R se due hanno i medesimi valori degli attributi di X allora hanno anche gli stessi di Y.

Imporre come vincolo che valga una dipendenza funzionale significa che: per ogni possibile estensione di R Y deve dipendere funzionalemente da X. Si osservi che per definizione di chiave, ogni attributo non chiave di una relazione dipende funzionalmente dalla chiave. In sostanza i vincoli di chiave e superchiave sono dei casi particolari di dipendenze funzionali. Si usa distinguere tra dipendenza parziale e completa di Y da un insieme X. Nel primo significa che

$$\exists X' \subset X | X' \rightarrow Y$$

nell'altro no, cioè se da X viene tolto anche un solo attributo la dipendenza funzionale non è più vera.

B.6.3 Le forme principali

Sono state proposte diverse forme normali corrispondenti a diversi standard di normalizzazione. Lo standard che viene, ormai, considerato come minimo è la terza forma normale (**3NF**). La prima forma normale (**1NF**) è quella in cui gli attributi sono di tipo semplice, non composti da più parti, cioè la forma in cui si devono trovare tutte le relazioni secondo il modello classico. Le altre due vengono qui definite.

Definizione 2NF B.6.4. Lo schema di una relazione è in seconda forma normale quando:

1. R è in prima forma normale;
2. per ogni chiave di R non esistono attributi non chiave che dipendono parzialmente dalla chiave;

Definizione 3NF B.6.5. Lo schema di una relazione R è in terza forma normale quando:

1. R è in seconda forma normale (da notare 2NF $\Rightarrow$ 1NF);

2. per ogni dipendenza funzionale $X \rightarrow Y$ è vera una delle seguenti due condizioni:
   a) X è una superchiave della relazione;
   b) Y è un membro di una chiave delle relazione;

C STANDARD SQL2

C.1 INTRODUZIONE

Lo standard *SQL2* è stato approvato nel 1992. Ha lo scopo di definire le strutture e le operazioni di base sui dati. Questo standard internazionale specifica la sintassi e la semantica di un linguaggio (*SQL*) per i *database*. Si deve chiarire che lascia agli sviluppatori molte scelte riguardo al tipo di comunicazione, ai componenti dell'applicativo e all'ottimizzazione delle performance. In questa appendice viene riportato un riassunto delle parti più importanti che riguardano quanto fatto nel progetto. Per tutto il resto si rimanda al documento ufficiale. Ci sono state delle evoluzioni in standard più recenti ma trattano argomenti più avanzati non ancora sviluppati nel *dbms*. Lo standard usa i termini tabella, riga e colonna in luogo, rispettivamente, dei termini relazione, tupla e attributo. Noi consideriamo questi termini come sinonimi, cioè intercambiabili.

C.2 TIPI DI DATI

I principali tipi di dati disponibili per gli attributi includono tipi numerici, stringhe di caratteri, stringhe di bit, booleani, date e ora.

1. ***Tipi numerici***. Comprendono numeri interi di diverse dimensioni (*SMALLINT*, *INT*) e numeri in virgola mobile con diversa precisione (*FLOAT*, *DOUBLE*);

2. ***Tipi per caratteri e stringhe***. Comprendono quelli a lunghezza fissa (*CHAR(n)*, *CHARACTER(n)*), in cui *n* è il numero di caratteri e a lunghezza variabile (*VARCHAR(n)*, *CHAR VARYING(n)*), in cui *n* è, questa volta, il numero massimo di caratteri. Da totare come le stringhe siano *case sensitive* al contrario del linguaggio in sè, che non distingue *select* da *SELECT*;

3. ***Tipi per stringhe di bit***. Come per le stringhe vi sono sia quelli a lunghezza fissa *n*, *BIT(n)*, che a lunghezza variabile *BIT VARYING(n)*, in cui *n* è il numero massimo di bit;

4. ***Il tipo booleano***. Assume solo i valori *TRUE* e *FALSE*. Ricordiamo della presenza dei valori *NULL* e della logica a tre valori;

5. ***Tipi per date e ora***. A *SQL2* sono stati aggiunti dei nuovi tipi per data e ora. Il tipo *DATE* ha dieci posizioni e i suoi componenti sono *YEAR*, *MONTH* e *DAY*. Mentre il tipo di dati *TIME* ha minimo otto posizioni, con i componenti *HOUR*, *MINUTE* e infine *SECOND*;

6. ***Un tipo timestamp***. Comprende sia il tipo *DATE* che *TIME*, in aggiunta vi sono sei posizioni per le frazioni decimali di secondo;

C.3 IL LINGUAGGIO SQL

Il linguaggio descritto dallo standard contiene da un punto di vista concettuale più linguaggi, riportiamo in questa appendice solo tratti della grammatica relativa a quello chiamato *DML*.

C.3.1 Interrogazioni fondamentali

In *SQL* è presente un'istruzione per il recupero delle informazioni da una base di dati: l'istruzione ***SELECT***. Anticipiamo una differenza tra una tabella in *SQL* e il concetto di relazione usato nel modello relazionale, una relazione essendo un insieme non può contenere *tuple* uguali, mentre, per una tabella questo è possibile. Sarebbe pertanto più corretto dire che una tabella è un multinsieme (anticipato precedentemente). In *SQL* per evitare che una tabella contenga duplicati bisogna utilizzare la parola chiave *DISTINCT*. L'intera specifica della *SELECT* nello standard è molto lunga, anche perchè vi sarebbe la necessità di introddurre una notazione per descrivere la grammatica, si rimanda sempre alla documentazone dello standard per approfondimenti. Qui viene riportata in modo informale, molto più facile da leggere. Una base dell'istruzione *SELECT* è così composta:

SELECT	<elenco attributi e funzioni>
FROM	<elenco tabelle>
WHERE	<condizione>
GROUP BY	<attributi di raggruppamento>
HAVING	<condizioni di raggruppamento>
ORDER BY	<attributi di ordinamento>;

Le due componenti che non possono mancare sono *SELECT* e *FROM*, le restanti sono opzionali. L'ordine dei sottoparafrafi non è casuale, segue un possibile ordine di lettura della query. Quanto segue vuole solo introdurre allo *statement SELECT*.

Lista tabelle nella FROM

Nella lista delle tabelle della clausola *FROM*, si deve inserire una lista di nomi di tabelle intervallate dalla virgola. Vi è anche un'altra possibilità, quella di scrivere delle ulteriori *query* tra le virgole considerandole a tutti gli effetti una tabella. Si può procedere anche alla rinominazione di una tabella all'interno di questa lista, permettendo nelle parti successive di utilizzare tale nome al posto del precedente. Per farlo si può utilizzare la parola chiave *AS* seguita dal nuovo nome oppure solo il nome, ciò che conta è che ogni tabella sia separata dalla virgola.

Condizione di WHERE

Nella clausola *WHERE* si inseriscono delle espressioni nelle quali si possono specificare uguaglianze, disuguaglainze ed altro tra colonne, colonna e valore, appartenenza a un'insieme etc. Rimandiamo alla documentazione ufficiale per capire con che regole scrivere la *WHERE*.

Lista attributi SELECT

Nell'elenco atributi della *SELECT*, si devono inserire i nomi delle colonne per eseguire la proiezione. Vi sono vari modi per riferirsi ad una colonna, quello più completo è anteporre il nome della tabella (NOME_TABELLA.NOME_ATTRIBUTO) o altrimenti soltanto il nome prestando attenzione che non vi siano tabelle nella *FROM* con il medesimo attributo. Nel caso un'interrogazione si riferisca a due o più attributi con lo stesso nome, ad esse, bisogna riferirsi obbligatoriamente soltanto con il primo modo. Per ogni attributo *SQL* permette di utilizzare la parola chiave *AS* per rinominare l'attributo nella tabella risultato. Si possono inserire anche delle funzioni aggregate, spiegate più avanti.

La clausola GROUP BY

Non poteva mancare nel linguaggio la possibilità di eseguire un raggruppamento. Il concetto può essere visto come la creazione di classi di equivalenza, dove un'elemento si dice equivalente ad un'altro se presenta gli stessi valori negli attributi specificati nella clausola *GROUP BY*. Alla fine verranno mostrate le classi di equivalenza codificate mediante una tupla fatta con la proiezione di alcuni attributi della tabella di partenza.

Le funzioni aggregate

Poichè il raggruppamento e l'aggregazione sono richiesti in molte applicazioni di basi di dati, *SQL* comprende funzioni che

implementano tali concetti, tra cui alcune funzioni integrate (*built-in*).

1. *COUNT*;
2. *SUM*;
3. *MAX*;
4. *MIN*;
5. *AVG*;

La funzione *COUNT* restituisce il numero di tuple o valori individuati da un'interrogazione. Le funzioni *SUM*, *MAX*, *MIN* e *AVG* sono applicate a un insieme di valori numerici e restituiscono rispettivamente la somma, il valore massimo, il valore minimo e la media dei valori presenti. Le funzioni *MAX* e *MIN* possono essere utilizzate anche come attributi che hanno domini non necessariamente numerici, con condizione che in tale dominio vi sia definito un ordine totale. Sono nella maggior parte dei casi usate assieme al *GROUP BY*.

La condizione HAVING

Talvolta si vogliono recuperare i valori di raggruppamenti che soddisfano certe condizioni. Se per esempio si vogliono prendere soltanto i gruppi composti da più di un certo numero di componenti, *SQL* mette a disposizione la clausola *HAVING* per poter esprimere tale condizione.

Ordinamento del risultato

Il linguaggio consente di ordinare le *tuple* di un'interrogazione rispetto al valore di uno o più attributi usando la clausola *ORDER BY*. L'ordine è lessicografico da sinistra a destra. L'ordine predefinito è l'ordine crescente dei valori.

Unione, intersezione e differenza in SQL

Come si è detto precedentemente, *SQL* di solito considera una tabella non come un insieme, ma piuttosto come un multinsieme. *SQL* non elimina automaticamente le tuple duplicate nei risultati delle interrogazioni per i seguenti motivi:

1. l'eliminazione dei duplicati è un' operazione costosa;
2. l'utente può voler vedere le tuple duplicate nel risultato di un'interrogazione;

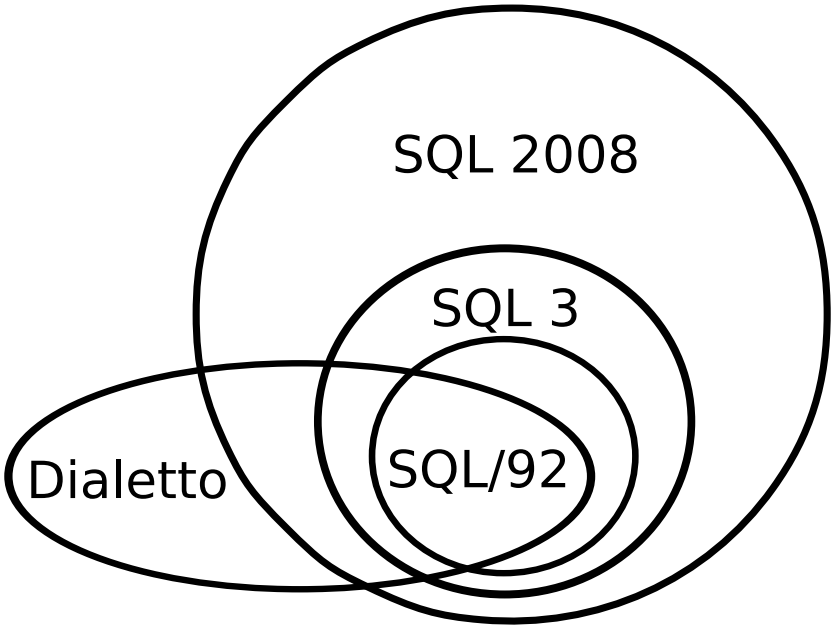


Figura 10.: Relazione tra tipi di SQL

3. quando una funzione di aggregazione è applicata alle tuple, nella maggior parte dei casi non si vogliono eliminare i duplicati;

Alcune operazioni insiemistiche dell'algebra relazionale sono state incorporate in *SQL*. Vi sono tra tutte, le operazioni di unione (*UNION*), differenza (*EXCEPT*) e intersezione (*INTERSECT*). I risultati come relazioni che ne derivano sono insiemi di tuple, cioè le tuple duplicate sono eliminate dal risultato. Poichè tali operazioni si applicano solo alle relazioni compatibili all'unione, ci si deve assicurare che le due relazioni a cui si applica un'operazione insiemistica abbiano gli stessi atributi e nel medesimo ordine. *SQL* ha in aggiunta le corrispondenti operazioni multinsieme, che vengono scritte postponendo la parola chiave *ALL* all'operazione scelta.

C.3.2 Non un solo SQL

Nella realtà è assodato che non esiste un unico *SQL* (tralasciando i vari standard) perchè ogni produttore di *DBMS* propone un proprio dialetto. Questo sarà più o meno rispettoso degli standard **ANSI** che vengono con costanza aggiornati. Si troverà sicuramente in questa situazione 10.
La figura 10 serve a rappresentare il fatto che molti dialetti di *SQL* si discostano sempre di più dagli standard via, via più recenti, accettandoli solo in parte e aggiungendo delle funzionalità proprie che risultano, comunque, molto utili. Oggigiorno la parte comune è

quasi l'intero *SQL/92*. Questo rappresenta lo scoglio maggiore alla portabilità, porta cioè al paradosso di dover utilizzare uno standard, ormai, vecchio decenni se da un lato si vuole tale caratteristica o di rinunciarvi se si ritengono più utili le possibilità specifiche. L'augurio, come sempre, consiste in uno standard completo che possa comprendere tutte le miliori funzionalità dei vari dialetti.

C.3.3 Limiti computazionali di SQL/92

Spesso è utile chiedersi se esistono dei limiti, da parte di un linguaggio, sul potere espressivo. Tutti i linguaggi hanno dei limiti, qui la parola limite significa di non poter raggiungere il massimo potere espressivo che un linguaggio, a livello teorico, può avere. Poniamoci inizialmente le seguenti due domande.

esistono delle interrogazioni che si possono esprimere nell'algebra relazionale, ma non in *SQL* ?

Fortunatamente a tale domanda vi è una risposta negativa: *SQL* consente di rappresentare ciascuna delle operazioni primitive dell'algebra relazionale.
Ora la domanda simmetrica.

esistono delle interrogazioni che si possono esprimere n *SQL*, ma non nell'algebra relazionale ?

Qui la risposta è affermativa, in quanto l'algebra relazionale non possiede l'operatore di raggruppamento e nemmeno le funzioni aggregate. Fino a questo momento si è solo capito che l'algebra relazionale ha dei limiti, resta da capire se anche *SQL/92* presenta o meno dei limiti. Per quanto sia scomodo vi sono delle interrogazioni che non si possono esprimere nemmeno in *SQL/92*, questo fatto risulta molto grave dato che anche *query* intuitive presentano questo limite. Un esempio è rappresentato dalla chiusura transitiva di una relazione, spesso, infatti, capita di voler ottenere proprio la chiusura di una tabella da una interrogazione. In termini più formali si dice che *SQL/92* non è un linguaggio completo. Non c'è alcuna ragione per disperarsi, perchè esistono due modi di ovviare a questo problema.

1. utilizzare uno standard più recente, cioè da *SQL3* in poi;

2. ricorrere ad un linguaggio di programmazione (universale) quali *Java*, C, C++, *Python*, *Perl* etc.

Riportiamo in un'immagine quanto appena detto 11.

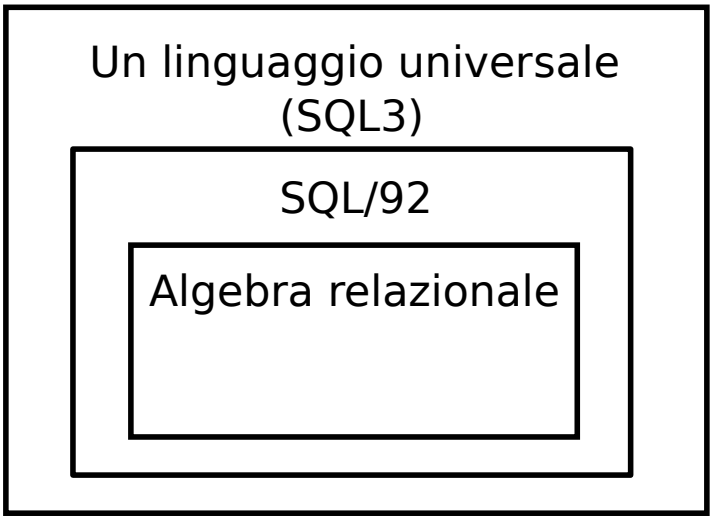


Figura 11.: Espressività

C.4 LE TABELLE AGGIUNTIVE

In precedenza si è detto che un *DBMS* contiene, oltre alle tabelle dei database definiti, delle informazioni aggiuntive. Lo standard *SQL* prevede l'aggiunta di una collezzione di tabelle all'interno di uno schema chiamato *INFORMATION_SCHEMA*. Questo permette a un utente di estrapolare le informazioni aggiuntive utilizzando le interrogazioni classiche di *SQL*. Riportiamo un breve elenco di tali tabelle compresa la loro definizione. Va ricordato l'importanza dell'ordine delle colonne presenti, anche solo scambiarle comporta gravi problemi a tutti gli applicativi che ne fanno uso, perchè di solito l'accesso avviene per indice, non per nome. Quanto viene proposto tralascia lo schema *DEFINITION_SCHEMA*, questo implica di dover definire in maniere differente le tabelle. Quello che conta è soltanto il risultato, non come in realtà internamente venga gestito.

C.4.1 Tabella CATALOG_NAME

All'interno di questa tabella va inserito il nome del catalogo che contiene lo schema *INFORMATION_SCHEMA*. La definizione di tale tabella in 27:

Codice 27: Definizione CATALOG_NAME.

```
CREATE TABLE INFORMATION_SCHEMA_CATALOG_NAME
(CATALOG_NAME SQL_IDENTIFIER,
CONSTRAINT INFORMATION_SCHEMA_CATALOG_NAME_PRIMARY_KEY
PRIMARY KEY (CATALOG_NAME)
)
```

C.4.2 Tabella TABLES

Vi è una riga per ogni tabella, incluse le viste. La definizione di tale tabella tramite *SQL*:

Codice 28: Definizione TABLES.

```
CREATE TABLE TABLES
(TABLE_CATALOG INFORMATION_SCHEMA.SQL_IDENTIFIER,
TABLE_SCHEMA INFORMATION_SCHEMA.SQL_IDENTIFIER,
TABLE_NAME INFORMATION_SCHEMA.SQL_IDENTIFIER,
TABLE_TYPE INFORMATION_SCHEMA.CHARACTER_DATA
/* Aggiunta di vincoli di chiave e chiavi esterne */
)
```

C.4.3 Tabella VIEWS

Questa tabella in ogni riga descrive brevemente una vista presente nel *DBMS*. La definizione dello standard di tale tabella:

Codice 29: Definizione VIEWS.

```
CREATE TABLE VIEWS
(TABLE_CATALOG INFORMATION_SCHEMA.SQL_IDENTIFIER,
TABLE_SCHEMA INFORMATION_SCHEMA.SQL_IDENTIFIER,
TABLE_NAME INFORMATION_SCHEMA.SQL_IDENTIFIER,
VIEW_DEFINITION INFORMATION_SCHEMA.CHARACTER_DATA,
CHECK_OPTION INFORMATION_SCHEMA.CHARACTER_DATA
/* Aggiunta di vincoli di chiave e chiavi esterne */
)
```

C.4.4 Tabella COLUMNS

Tutte le colonne di ogni tabella sono elencate tramite una riga nella tabella *COLUMNS*. La sua definizione:

Codice 30: Definizione COLUMNS.

```
CREATE TABLE COLUMNS
(TABLE_CATALOG INFORMATION_SCHEMA.SQL_IDENTIFIER,
TABLE_SCHEMA INFORMATION_SCHEMA.SQL_IDENTIFIER,
TABLE_NAME INFORMATION_SCHEMA.SQL_IDENTIFIER,
COLUMN_NAME INFORMATION_SCHEMA.SQL_IDENTIFIER,
ORDINAL_POSITION INFORMATION_SCHEMA.CARDINAL_NUMBER,
DOMAIN_CATALOG INFORMATION_SCHEMA.SQL_IDENTIFIER,
DOMAIN_SCHEMA INFORMATION_SCHEMA.SQL_IDENTIFIER,
DOMAIN_NAME INFORMATION_SCHEMA.SQL_IDENTIFIER,
COLUMN_DEFAULT INFORMATION_SCHEMA.CHARACTER_DATA.
IS_NULLABLE INFORMATION_SCHEMA.CHARACTER_DATA
/* Aggiunta di vincoli di chiave e chiavi esterne */
)
```

Come già affermato lo standard prevede una grande quantità di tabelle (qui per motivi di spazio non riportate) che riguardano i vincoli, le viste, e altre informazioni su tutti i dati presenti. Tutte queste devono essere presenti nel *DBMS* affinchè si possano estrapolare le informazioni utilizzate dagli applicativi.

INDICE ANALITICO

BIBLIOGRAFIA

[1] M Alesky. *Implementing Distributed System With Java and Corba*. Springer, London, 2006.

[2] P Atzeni. *Modelli e linguaggi di interrogazione*. McGraw-Hill, Boston, 2009.

[3] R Elmasri. *Sistemi di basi di dati. Fondamenti*. Pearson Addison Wesley, Madrid, 2011.

[4] M Fisher. *JDBC API Tutorial and Reference volume 1*. Addison Wesley, Boston, 2003.

[5] M Fisher. *JDBC API Tutorial and Reference, volume 2*. Addison Wesley, Boston, 2003.

[6] I Forman. *Java Reflection in Action*. Wenning, New York, 2004.

[7] M Goodrich. *Data Structures and Algorithms in Java*. John Wiley and Sons, New York, 2006.

[8] W Grosso. *Java RMI*. O'Relly, London, 2008.

[9] R Harrop. *Pro Spring 3*. Apress, New York, 2012.

[10] C Horstman. *Concetti di informatica e fondamenti di Java*. Apogeo, London, 2006.

[11] S Kartsen. *Java - Dai fondamenti alla programmazione avanzata*. Hoepli, New York, 2005.

[12] B Korte. *Ottimizzazione combinatoria, Teoria e Algoritmi*. Springer, London, 2007.

[13] J Langr. *Agile Java: Crafting Code with Test-Driven Development*. Prentice Hall, London, 2005.

[14] R Motwani. *Automi, linguaggi e calcolabilità*. Pearson Addison Wesley, New York, 2009.

[15] S Perry. *Log4J*. O'Relly, New York, 2010.

[16] A.R Seddighi. *Spring persistance with Hibernate*. Packt, New York, 2009.

[17] J.M Smith. *Elemental Design Patterns*. Addison Wesley, London, 2000.

[18] J.D Ullman. *Introduction to Automata Theory, Languages, and Computation*. Pearson Addison Wesley, New York, 1979.

[19] J.D Ullman. *Compilatori*. Pearson Addison Wesley, New York, 2009.