Università degli studi di Torino
Dipartimento di informatica

Corso di laurea in informatica

Progettazione e implementazione di un DBMS relazionale

Relatore:
Prof. Ruggero G. Pensa

Candidato:
Leonardo Magliolo

Anno Accademico 2020/2021

Dichiaro di essere responsabile del contenuto dell'elaborato che presento al fine del conseguimento del titolo, di non avere plagiato in tutto o in parte il lavoro prodotto da altri e di aver citato le fonti originali in modo congruente alle normative vigenti in materia di plagio e di diritto d'autore. Sono inoltre consapevole che nel caso la mia dichiarazione risultasse mendace, potrei incorrere nelle sanzioni previste dalla legge e la mia ammissione alla prova finale potrebbe essere negata.

Ringraziamenti:

Desidero ringraziare i miei genitori per il fondamentale supporto morale ed economico offertomi lungo tutto il percorso di laurea.
Un ringraziamento speciale va a mia madre che, nonostante i numerosi problemi di salute, si è sempre mostrata disponibile affinché io potessi proseguire gli studi in tranquillità, nel limite del possibile.

Ringrazio, inoltre, gli amici e i compagni per la pazienza e il supporto prestatomi durante la stesura della tesi.
È stato particolarmente prezioso il contributo offerto da Athena, Cristian, Fabio, Greta e Marco al conseguimento del titolo.

Desidero ringraziare, in fine, il professore Pensa per avermi concesso la possibilità di lavorare sul progetto di laurea proposto, per la disponibilità e la professionalità con cui ha condotto il ruolo di relatore.

Indice

Capitolo 1: Introduzione

L’obiettivo della tesi consiste nella realizzazione di un Database Management System relazionale, portabile e dalle ridotte dimensioni, il cui funzionamento possa essere esaminato a fini didattici, aumentando la comprensione di come vengano progettati i moderni RDBMS in pratica.

La realizzazione del progetto prefigge, inoltre, lo scopo di consolidare ed ampliare le conoscenze apprese durante il corso di laurea di linguaggi formali, traduttori e basi di dati.

Nel capitolo dedicato ai linguaggi formali e traduttori verranno approfonditi meccanismi in grado di specificare e interpretare la sintassi del linguaggio SQL.

Nel capitolo dedicato alle basi di dati verrà svolta una breve introduzione al modello relazionale, alla specifica del linguaggio SQL e alle metodologie attuabili alla rappresentazione dei dati e all’esecuzione delle query da parte di un DBMS.

Nel capitolo dedicato al progetto verranno forniti dettagli aggiuntivi rispetto alle scelte implementative adottate, sfruttando la terminologia e le osservazioni proposte nei capitoli precedenti.

Capitolo 2: Linguaggi formali e traduttori

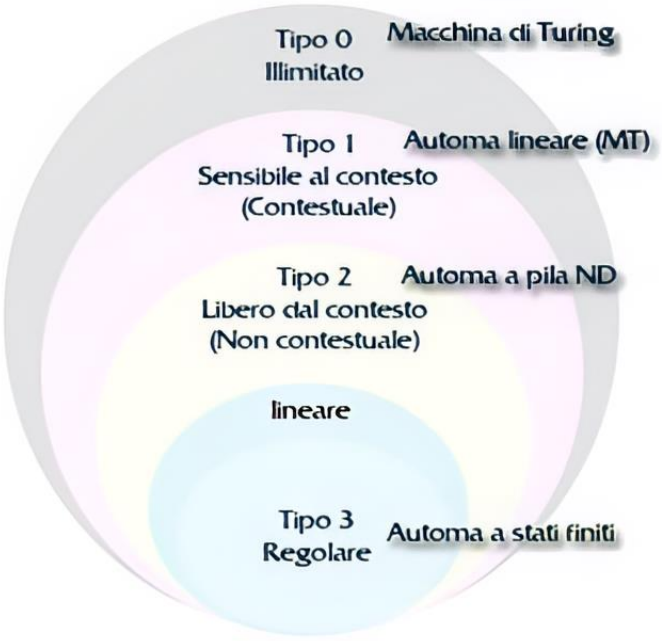
2.1 Linguaggi formali:

Nello studio dell'informatica e della linguistica, un linguaggio formale [1] consiste in un insieme di parole definite su un alfabeto in grado di rispettare uno specifico insieme di regole.

In particolare, un alfabeto (in gergo riconosciuto con **Σ**) consiste in un insieme di simboli, lettere o token.
Un insieme peculiare nell'ambito dello studio dei linguaggi formali è l'insieme di tutte le possibili stringhe generabili per mezzo dell'operazione di concatenazione all'interno di un alfabeto (in letteratura indicato con $\mathbf{\Sigma}^*$).
Considerando le precedenti definizioni è possibile osservare che, in effetti, per ogni alfabeto **Σ** e per ogni linguaggio formale **L** definito su **Σ** è sempre vera la relazione: $\mathbf{L} \subseteq \mathbf{\Sigma}^*$.

Nell'ottica di riuscire a stabilire quali caratteristiche dovesse avere un modello computazionale per riconoscere un determinato sottoinsieme di linguaggi formali, nel 1956 è stata introdotta nell'ambito dei linguaggi formali la gerarchia di Chomsky [2].
I modelli computazionali presi in considerazione dalla gerarchia sono rispettivamente:

- Automa a stati finiti
- Automa a pila non deterministico
- Automa lineare
- Macchina di Turing



Per quel che riguarda la stesura del progetto, sono stati utilizzati linguaggi regolari (di tipo 3) per identificare correttamente i token imposti dalla grammatica SQL sfruttando un analizzatore lessicale; è stato invece implementato un parser in grado di riconoscere la struttura sintattica SQL, sfruttando la categoria dei linguaggi formali di tipo 2.

Per queste ragioni, i paragrafi seguenti si concentreranno su una definizione più precisa di linguaggio regolare e linguaggio libero dal contesto.

2.1.1 Linguaggi regolari:

Un linguaggio regolare **R** è un linguaggio formale per il quale è possibile definire un automa a stati finiti *A* tale per cui l'insieme **L(*A*)** delle stringhe accettate da *A* rispetti la seguente condizione: $\mathbf{L}(\boldsymbol{A}) \subseteq \mathbf{R} \wedge \mathbf{R} \subseteq \mathbf{L}(\boldsymbol{A})$.

Un automa a stati finiti deterministico (DFA) è definibile con la quintupla:$< \mathbf{Q}, \mathbf{\Sigma}, \delta, q_0, \mathbf{F} >$ i quali elementi sono:

- Un insieme non vuoto di stati **Q.**
- Un insieme finito di simboli **Σ.**
- Una funzione di transizione δ: **Σ**x**Q** → **Q.**
- Uno stato iniziale $q_0 \in \mathbf{Q}$.
- Un insieme di stati finali **F** ⊆ **Q.**

É possibile notare come la funzione di transizione δ sia sufficiente a stabilire per ogni $s \in \mathbf{\Sigma}$ se è vero che $s \in \mathbf{L}(\mathbf{DFA})$.

Per estendere efficacemente la funzione di transizione, così che possa considerare ogni $w \in \mathbf{\Sigma}^*$, è necessario definire una funzione di transizione estesa $\hat{\delta}$: $\mathbf{\Sigma}^*$x**Q** → **Q** in modo tale che sfrutti ricorsivamente la funzione δ.

Ciò può esser fatto tenendo presente che è sempre vera la seguente condizione per ogni $w \in \mathbf{\Sigma}^*$: $w = w_1 w_2 \ldots w_n$, con $w_i \in \mathbf{\Sigma}$ per ogni i (la giustapposizione nel caso della sequenza dei w_i identifica l'operazione di concatenazione).

Pertanto la funzione $\hat{\delta}$ è definibile come segue:

- Passo base: $\hat{\delta}(q, \varepsilon) = q$ (ε in letteratura fa riferimento alla stringa vuota).

- Passo induttivo: $\hat{\delta}(q, w) = \hat{\delta}(q, w_1 w_2 \dots w_i) = \delta(\hat{\delta}(q, w_1 w_2 \dots w_{i-1}), w_i)$.

La definizione di $\hat{\delta}$ permette di descrivere in maniera rigorosa il linguaggio accettato dal precedente DFA: $\mathbf{L(DFA)} = \{ w \in \mathbf{\Sigma}^* | \hat{\delta}(q_0, w) \in \mathbf{F} \}$.

Sebbene i linguaggi regolari siano riconoscibili per mezzo di DFA, è possibile formulare una dimostrazione [3] strutturale sulla generica coppia di DFA *A* e *B* che provi che per i linguaggi regolari **L(*A*)**, **L(*B*)**:

- **L(*A*)** U **L(*B*)** è regolare.
- **L(*A*)L(*B*)** è regolare (la giustapposizione tra L(*A*) e L(*B*) identifica l'operazione di concatenazione).
- **L(*A*)*** e **L(*B*)*** sono linguaggi regolari (* fa riferimento alla chiusura di Kleene).

Diventa possibile, quindi, la definizione di linguaggi regolari partendo da elementi appartenenti a **Σ** per mezzo di espressioni concise che sfruttano gli operatori insiemistici elencati in precedenza; evitando così il ben più arduo compito di dover fornire esplicitamente la funzione di transizione, ma mantenendo la possibilità di poter computare il linguaggio accettato dall'espressione, generando algoritmicamente un DFA che accetti un linguaggio equivalente per poi calcolare i risultati della sua funzione di transizione estesa $\hat{\delta}$.
Le espressioni citate in precedenza prendono il nome di espressioni regolari [4], utilizzate nell'ambito di questo progetto per la definizione dei token SQL.

2.1.2 Linguaggi liberi dal contesto:

Un linguaggio libero dal contesto L è un linguaggio formale per il quale è possibile definire una grammatica libera dal contesto (CFG) *G* tale che l'insieme **L(*G*)** delle stringhe accettate da *G* rispetti la seguente condizione: $\mathbf{L}(\boldsymbol{G}) \subseteq \mathbf{L} \wedge \mathbf{L} \subseteq \mathbf{L}(\boldsymbol{G})$.

Una CFG *G* è definibile come una quadrupla:$< \mathbf{V}, \mathbf{T}, \mathbf{P}, S >$ i quali elementi sono:

- Un insieme di variabili **V.**
- Un insieme di terminali **T.**
- Un insieme di produzioni **P**, una produzione è definita come: $v \rightarrow c$ con $v \in \mathbf{V}$ e $c \in (\mathbf{V} \text{ U } \mathbf{T})^*$.

In particolare, v viene spesso indicata come "testa" della produzione e c come "corpo".

- Un simbolo iniziale $S \in \mathbf{V}$.

Ai fini del formalizzare quale linguaggio venga accettato da una CFG, è utile introdurre l'operatore di derivazione $\Rightarrow_{\boldsymbol{G}}$: $\alpha A\beta \Rightarrow_{\boldsymbol{G}} \alpha\gamma\beta \leftrightarrow (A \to \gamma) \in \mathbf{P}$ con $A \in \mathbf{V}$ **e** $\alpha, \beta, \gamma \in (\mathbf{V} \cup \mathbf{T})^*$

In modo simile, per come è stata estesa la funzione di transizione δ a $\hat{\delta}$, si desidera definire un nuovo operatore $\Rightarrow^*_{\boldsymbol{G}}$ su basi ricorsive che identifichi sequenze multiple di derivazione:

- Passo base: $\forall_{\alpha \in (\mathbf{V} \cup \mathbf{T})^*}$: $\alpha \Rightarrow^*_{\boldsymbol{G}} \alpha$.
- Passo induttivo: $\forall_{\alpha,\beta,\gamma \in (\mathbf{V} \cup \mathbf{T})^*}$: $\alpha \Rightarrow^*_{\boldsymbol{G}} \beta \ \wedge \ \beta \Rightarrow_{\boldsymbol{G}} \gamma \ allora \ \alpha \Rightarrow^*_{\boldsymbol{G}} \gamma$.

A questo punto è possibile specificare in maniera precisa quale sia il linguaggio accettato dalla grammatica:

$$\mathbf{L}(\boldsymbol{G}) = \{ w \in \mathbf{T}^* \mid S \Rightarrow^*_{\boldsymbol{G}} w \}.$$

È possibile notare come si possa costruire ricorsivamente un albero i cui nodi appartengono a $(\mathbf{V} \cup \mathbf{T})^*$ in modo tale che identifichi una sequenza di produzioni applicate sulla grammatica *G*:

- Passo base:

  $\forall_{\gamma \in \mathbf{T}*} \gamma \Rightarrow^*_{\boldsymbol{G}} \gamma$ viene associato un albero composto di una sola foglia contenente γ

- Passo induttivo:

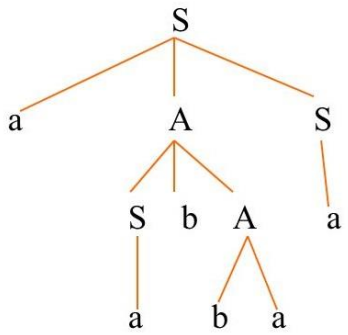
  $\forall_{\alpha,\beta_1,\ldots,\beta_n \in (\mathbf{V} \cup \mathbf{T})^*} \alpha \Rightarrow^*_{\boldsymbol{G}} \beta_1 \ldots \beta_n$ viene associato un albero la cui radice rappresenta un nodo contenente α avente esattamente n figli.
  Ogni i-esimo figlio è anch'esso un albero ricavato da $\beta_i \Rightarrow^*_{\boldsymbol{G}} \omega_1 \ldots \omega_m$.

In particolare, se $w \in \mathbf{L}(\boldsymbol{G})$ è possibile generare un albero sintattico per mezzo della procedura sopra descritta in modo tale che la concatenazione da sinistra verso destra di tutte le sue n foglie $w_i \in \mathbf{T}^*$ sia equivalente a w, o in altri termini: $w = w_1 \ldots w_n$.

L'albero così generato viene chiamato in letteratura "albero sintattico", la cui costruzione può esser sfruttata da un programma (parser) per determinare se $S \Rightarrow^*_{\boldsymbol{G}} w$ e dunque stabilire se $w \in \mathbf{L}(\boldsymbol{G})$.

Viene riportato di seguito un esempio di albero sintattico:

G = ({S,A} ,{a, b}, P, S)
Con P:
$S \rightarrow a\ A\ S$
$A \rightarrow S\ b\ A$
$A \rightarrow S\ S$
$S \rightarrow a$
$A \rightarrow b\ a$



Risultato dell'albero : **a a b b a a**

$S \rightarrow aAS \rightarrow aSbAS \rightarrow aabAS \rightarrow aabbaS \rightarrow aabbaa$

Per alcune grammatiche è possibile dimostrare l'esistenza di almeno una stringa w nel linguaggio accettato per cui esistono alberi sintattici differenti le cui foglie concatenate da sinistra verso destra sono comunque equivalenti a w.

Grammatiche che posseggono la proprietà sopra elencata vengono classificate come "ambigue" [5] e in molti casi rappresentano un ostacolo nella definizione di un parser qualora l'albero sintattico debba esser costruito osservando un determinato ordine di precedenza rispetto alle produzioni da applicare.

2.2 Traduttori:

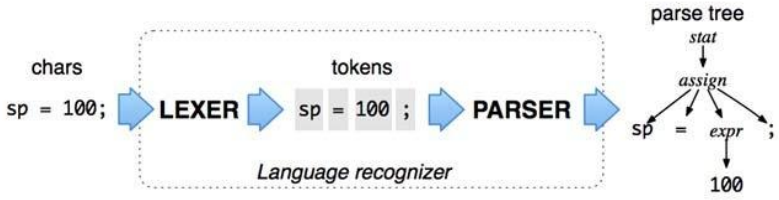
Traduttore, nell'ambito informatico, è un termine facente riferimento a programmi in grado di convertire codice scritto in uno specifico linguaggio di programmazione in un altro linguaggio [6].

Essendo i linguaggi di programmazione anche linguaggi formali, è possibile nella maggior parte dei casi applicare le metodologie illustrate nei paragrafi precedenti per la definizione lessicale e sintattica degli stessi.

In genere è possibile scomporre un traduttore in sottoprogrammi in grado di dialogare l'uno con l'altro, in modo tale da gestire efficacemente la complessità del codice e suddividerne le mansioni per una maggiore comprensione della logica di funzionamento.

I principali sottoprogrammi implementati durante la stesura di un traduttore sono un analizzatore lessicale (in gergo chiamato lexer) e un analizzatore sintattico (chiamato parser).

Di seguito viene mostrato un esempio intuitivo che illustra la suddivisione dei compiti tra lexer e parser (tratto dalla fonte [7]):



2.2.1 Analisi lessicale:

L'analisi lessicale è il processo in grado di convertire una sequenza di caratteri in una sequenza di token; un programma in grado di eseguire un'analisi lessicale prende il nome di lexer o tokenizer.

Un token è una stringa per la quale viene assegnato un ruolo all'interno del linguaggio formale; Più precisamente, è possibile definirlo come una coppia di stringhe: <Nome del token, Valore del token>.

Le più comuni tipologie di token all'interno dei linguaggi di programmazione sono:

- Identificatore: Nome associato a una variabile scelto dal programmatore.
- Parole chiave: Nome facente riferimento a costrutti tipici del linguaggio.
- Operatore: Simbolo in grado di prende in input determinati tipi di dato e restituire un risultato.
- Letterale: Rappresenta delle costanti per un particolare tipo di dato.
- Commento: Sequenza di caratteri arbitraria non interpretata dal traduttore.

Vengono considerati di particolare importanza caratteri identificati come separatori (in genere lo spazio e il ritorno a capo), in quanto utilizzati dall'analizzatore lessicale per determinare la fine della sequenza di caratteri a cui associare un token.

La maggior parte dei linguaggi di programmazione dispone di token in grado d'essere associati a sequenze di caratteri appartenenti alla famiglia dei linguaggi regolari, pertanto il riconoscimento dei token viene spesso delegato all'utilizzo di espressioni regolari.

2.2.2 Analisi sintattica:

L'analisi sintattica è il processo in grado di stabilire come una stringa di terminali possa essere generata da una grammatica; un programma in grado di eseguire un'analisi sintattica prende il nome di parser. Tipicamente, l'analisi sintattica effettuata da un parser sfrutta la generazione di un albero sintattico.

A seconda del tipo d'analisi utilizzata per la costruzione dell'albero sintattico sopra citato, i parser possono esser classificati come LR o LL:

- Analisi top-down: La costruzione dell'albero sintattico comincia ponendo come radice il simbolo iniziale della grammatica formale considerata, determinando progressivamente porzioni sempre più ristrette dell'albero, associate ai discendenti del nodo radice.
  I parser LL eseguono un'analisi di tipo top-down.
- Analisi bottom-up: La costruzione dell'albero sintattico comincia con la generazione delle foglie associate ai token della stringa considerata, determinando progressivamente porzioni sempre più ampie dell'albero, associate agli antenati dei nodi foglia.
  I parser LR eseguono un'analisi di tipo bottom-up.

Il parser utilizzato per il riconoscimento della sintassi SQL implementato per questo progetto è di tipo LL. Pertanto, il resto del paragrafo si concentrerà più in dettaglio sulle metodologie adottabili per la costruzione dell'albero sintattico utilizzando un'analisi di tipo top-down.

Un parser di tipo LL è in grado di generare l'albero sintattico per una particolare sequenza di token sfruttando una metodologia analoga a quella utilizzata dall'algoritmo Depth-First Search [8], il quale utilizza una pila come coda per esplorare i nodi dell'albero, in funzione della loro profondità.

Essendo gli odierni linguaggi di programmazione dotati di richiami a funzioni, implementati sfruttando una pila come struttura dati, è possibile realizzare un parser di tipo LL associando una funzione ad ogni variabile appartenente alla grammatica.
Ogni funzione eseguirà specifici richiami ad altre funzioni in relazione alla produzione da applicare, avente simbolo in testa uguale a quello della variabile associata alla funzione stessa.

È di notevole importanza nella stesura di un parser saper determinare, scelta una determinata variabile della grammatica, quale produzione applicare per mezzo della prossima derivazione. Le grammatiche che permettono parsing predittivo discendente sono chiamate LL(k), dove k è il numero di simboli necessari a individuare la prossima produzione da applicare senza ambiguità.

L'insieme **LL(k)** contiene tutti i e soli i linguaggi formali in grado d'esser definiti per mezzo di una grammatica LL(k), per un valore di k >= 1.

Un parser viene classificato come LL(*) se è in grado di parsificare una generica grammatica LL(K), per un qualsiasi k ∈ ℕ.

Per semplicità, si procede con l'illustrare il criterio secondo cui un parser LL è in grado di determinare quale produzione applicare senza ambiguità considerata una grammatica di tipo LL(1).

Data una CFG $G < \mathbf{V}, \mathbf{T}, \mathbf{P}, S >$, l'insieme **FIRST(α)** contenente tutti gli $a \in \mathbf{T}$ tale che $\alpha \Rightarrow^*_{\boldsymbol{G}} a$ è calcolabile ricorsivamente come segue:

- $\mathbf{FIRST}(\boldsymbol{\varepsilon}) = \{\boldsymbol{\varepsilon}\}$
- $\forall_{t \in \boldsymbol{T}} \forall_{\beta \in (\boldsymbol{T} \cup \mathbf{V})^*} \mathbf{FIRST}(\boldsymbol{t\beta}) = \{\boldsymbol{t}\}$
- $\forall_{\omega,\beta \in (\boldsymbol{T} \cup \mathbf{V})^*} \beta \Rightarrow^*_{\boldsymbol{G}} \varepsilon \rightarrow \mathbf{FIRST}(\boldsymbol{\beta\omega}) = \mathbf{FIRST}(\boldsymbol{\beta}) - \{\boldsymbol{\varepsilon}\} \cup \mathbf{FIRST}(\boldsymbol{\omega})$
- $\forall_{\omega,\beta \in (\boldsymbol{T} \cup \mathbf{V})^*} \neg(\beta \Rightarrow^*_{\boldsymbol{G}} \varepsilon) \rightarrow \mathbf{FIRST}(\boldsymbol{\beta\omega}) = \mathbf{FIRST}(\boldsymbol{\beta})$

L'insieme **FOLLOW(A)** con A ∈ **V** , contenente tutti gli $a \in \mathbf{T}$ con cui iniziano le stringhe che seguono A nelle derivazioni della grammatica *G* (definendo $ come carattere finale della stringa), è calcolabile come segue:

- $V = S \rightarrow \mathbf{FOLLOWSTART}(\boldsymbol{V}) = \{\$\}$
- $V \neq S \rightarrow \mathbf{FOLLOWSTART}(\boldsymbol{V}) = \emptyset$
- $$\mathbf{FOLLOW}(\boldsymbol{A}) = \left\{\bigcup_{B \rightarrow \alpha A \beta \in P} (\mathbf{FIRST}(\boldsymbol{\beta}) - \{\boldsymbol{\varepsilon}\})\right\} \cup \left\{\bigcup_{\substack{B \rightarrow \alpha A \beta \in \boldsymbol{P} \\ \wedge\, \beta \Rightarrow^*_{\boldsymbol{G}} \varepsilon \\ \wedge B \neq A}} \mathbf{FOLLOW}(\boldsymbol{B})\right\} \cup \mathbf{FOLLOWSTART}(\boldsymbol{A})$$

In fine considerando *G*, l'insieme **PREDICT($A \rightarrow \alpha$)** contenente tutti gli $a \in \mathbf{T}$ con cui iniziano le stringhe generabili da $\boldsymbol{A \rightarrow \alpha}$ è calcolabile come segue:

- $\neg(\alpha \Rightarrow^*_{\boldsymbol{G}} \varepsilon) \rightarrow$ **PREDICT($A \rightarrow \alpha$) = FIRST**(α)
- $\alpha \Rightarrow^*_{\boldsymbol{G}} \varepsilon \rightarrow$ **PREDICT($A \rightarrow \alpha$) =** $(\mathbf{FIRST}(\alpha) - \{\varepsilon\})$ U $\mathbf{FOLLOW}(A)$

Se **L(G)**$\in \mathbf{LL}(\mathbf{1})$ allora per ogni A $\in \mathbf{V}$: $\bigcap_{A \rightarrow \alpha \in \boldsymbol{P}} \mathbf{PREDICT}(\boldsymbol{A \rightarrow \alpha}) = \emptyset$

Per cui, per un parser riconoscitore di linguaggi LL(1) è sufficiente determinare il prossimo token da consumare 'next_token' per calcolare, in funzione della variabile *A* considerata, quale sarà la prossima produzione da applicare con *A* in testa: $A \rightarrow \alpha$ è applicabile se e solo se next_token $\in \mathbf{PREDICT}(\boldsymbol{A \rightarrow \alpha})$.

2.2.3 Analisi semantica:

L'analizzatore semantico utilizza l'albero sintattico, costruito dal parser, per assicurare che il programma sorgente rispetti i vincoli semantici definiti all'interno del linguaggio considerato. Inoltre, raccoglie informazioni sui tipi di dati delle variabili salvandole, in genere, all'interno di una tabella dei simboli, ispezionata successivamente per la generazione di codice intermedio , oppure, per attuare generiche azioni semantiche in corrispondenza di sottoinsiemi della sintassi del linguaggio.
Un esempio particolarmente importante di analisi semantica è il 'type checking' [9], analisi in grado di stabilire per ogni operatore se gli sono stati forniti gli operandi aventi corretto tipo di dato.

L'approccio più comune per la progettazione di un analizzatore semantico consiste nella definizione di:

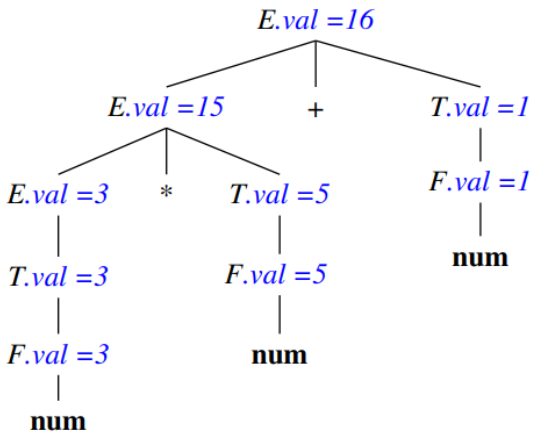
- Una serie di attributi (definibili come una tripla <Nome, Tipo di dato, Valore>) associati a variabili appartenenti alla grammatica formale considerata.
- Una definizione guidata dalla sintassi (Syntax-Directed Definition, abbreviata SDD), utilizzata per esplicitare le regole di valutazione da eseguire sugli attributi della grammatica in relazione alla produzione applicata.

La traduzione è il risultato della valutazione delle regole semantiche stabilite per mezzo di uno schema SDD.

In letteratura, un albero sintattico che riporta i valori degli attributi associati ai nodi viene chiamato albero sintattico annotato.

Viene riportato di seguito un esempio di SDD con annesso albero sintattico annotato (tratto dalla fonte [10]):

PRODUZIONI	REGOLE SEMANTICHE
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow E_1 - T$	$E.val = E_1.val - T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow T_1/F$	$T.val = T_1.val/F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow$ **num**	$F.val =$ **num**$.lexval$



(Sinistra: Esempio di SDD associato alle produzioni di una grammatica)

(Destra: Esempio di albero sintattico annotato, generato dalla grammatica riportata nell'esempio a sinistra, per l'input: 3 * 5 + 1)

Per accedere al valore di un attributo 'a' associato a una variabile della grammatica 'V' è stata utilizzata la notazione: 'V.a'.
Si noti, inoltre, come gli attributi associati a variabili terminali abbiano un valore fornito direttamente dall'analizzatore lessicale; negli schemi SDD non è possibile esplicitare regole semantiche per il calcolo dei valori degli attributi associati a variabili terminali.

In funzione di come l'attributo 'a' appartenente alla variabile 'V' è calcolabile a partire da un determinato nodo dell'albero sintattico n, viene classificato come:

- Sintetizzato: Se tutti gli attributi necessari al calcolo di 'a' sono direttamente associati al nodo n o ai suoi nodi figli.
  In questo caso, la variabile 'V' è presente nella testa della produzione considerata.

- Ereditato: Se tutti gli attributi necessari al calcolo di 'a' sono direttamente associati al nodo n, a suo nodo padre o ai suoi nodi fratelli.
  In questo caso, la variabile 'V' è presente nel corpo della produzione considerata.

A seconda dei tipi di attributi dichiarati all'interno di uno schema SSD, lo schema è classificabile come:

- S-attribuito: Se utilizza solo attributi sintetizzati
- L-attribuito, se per ogni attributo 'a' appartenente alla variabile 'V' si verifica una delle seguenti condizioni:
  - 'a' è sintetizzato
  - Se V.a è un attributo ereditato calcolato tramite una regola semantica associata alla produzione $\boldsymbol{A \rightarrow \omega_1 \ldots \omega_n V \omega_{n+2} \ldots \omega_m}$, allora la regola può utilizzare soltanto:
    - attributi ereditati di A.
    - attributi delle variabili $\boldsymbol{\omega_1 \ldots \omega_n}$.
    - attributi di V purché l'introduzione degli stessi renda comunque possibile l'esistenza di una sequenza di valutazione finita.
    - attributi di ω_i, con i compreso tra [n+2, m], purché ω_i sia terminale.

Sebbene gli schemi SDD specifichino quali azioni semantiche effettuare in relazione alla produzione applicata, non specificano necessariamente un ordine ammissibile con cui calcolare il valore degli attributi.
Per questa ragione vengono introdotti gli schemi di traduzione (Syntax-Directed Translation scheme, abbreviato SDT), in grado di specificare operazioni di traduzione da effettuare durante la parsificazione.

Uno schema di traduzione è una SDD, in cui le azioni semantiche, racchiuse tra parentesi graffe, vengono inserite nel corpo delle produzioni, in posizione tale che i valori degli attributi indicati all'interno delle azioni semantiche siano già stati calcolati.

$$
\begin{aligned}
&E \rightarrow E_1 + T \ \{E.val = E_1.val + T.val\} \\
&E \rightarrow E_1 - T \ \{E.val = E_1.val - T.val\} \\
&E \rightarrow T \ \{E.val = T.val\} \\
&T \rightarrow T_1 * F \{T.val = T_1.val * F.val\} \\
&T \rightarrow T_1 / F \{T.val = T_1.val / F.val\} \\
&T \rightarrow F \{T.val = F.val\} \\
&F \rightarrow (E \{F.val = E.val\}) \\
&F \rightarrow \mathbf{num} \{F.val = \mathbf{num}.lexval\}
\end{aligned}
$$

(Esempio di schema SDT, formulato sulla grammatica della SDD illustrata in precedenza)

In relazione alla tipologia di SDD, è possibile generare algoritmicamente uno schema SDT in grado di soddisfare i vincoli di disponibilità imposti sugli attributi all'interno delle azioni semantiche:

- Generazione di uno schema SDT partendo da una SDD S-attribuita:

  Vengono inserite le azioni, che calcolano gli attributi sintetizzati delle variabili in testa alle produzioni, alla fine delle produzioni stesse.
- Generazione di uno schema SDT partendo da una SDD L-attribuita:
  - Vengono inserite le azioni che calcolano gli attributi sintetizzati delle variabili in testa alle produzioni, alla fine delle produzioni stesse.
  - Vengono inserite le azioni che calcolano i valori degli attributi ereditati per un non terminale V immediatamente prima dell'occorrenza dello stesso, nel corpo delle produzioni interessate.
  - Qualora diversi attributi ereditati da V dipendessero l'uno dall'altro, verrebbero ordinate le valutazioni degli attributi interessati in modo da calcolare per primi quelli prima necessari.

Per ulteriori approfondimenti riguardo le SDD e gli schemi SDT si consiglia il consulto della fonte [11].

2.3 ANTLR:

ANTLR (ANother Tool for Language Recognition), è un generatore automatico di parser LL(*) per grammatiche arbitrarie di tipo LL(k).
In genere usato per la costruzione di interpreti e traduttori per linguaggi di dominio specifico (DSL), nella sua versione 4 è in grado di supportare numerosi linguaggi per la scrittura di lexer e parser (tra cui: Java, C#, C++ e Python).

Per mezzo della meta-sintassi EBNF (extended Backus–Naur form) [12] è possibile dichiarare, in maniera diretta, schemi di traduzione (SDT) utilizzati successivamente da ANTLR per generare automaticamente i corrispondenti traduttori.

2.3.1 Perché ANTLR?:

L'utilizzo di un generatore automatico di parser, qualora si intenda implementare un traduttore, comporta considerevoli benefici:

- Maggiore disaccoppiamento tra definizioni delle regole di produzione e relative azioni semantiche;
  per mezzo della meta-sintassi scelta dal generatore, utilizzata come linguaggio di rappresentazione intermedio, è possibile generare riconoscitori scritti in linguaggi sorgente differenti senza dover codificare manualmente lo stesso parser più volte in differenti linguaggi di programmazione.
- Tempistiche di sviluppo notevolmente ridotte.
- Report automatico di errori dovuti ai processi di analisi lessicale e analisi sintattica.
- Plugin di supporto IDE in grado di facilitare il debugging.

Sebbene i parser LR siano in grado di riconoscere una più ampia famiglia di linguaggi formali non riconoscibili dai parser LL, la tecnologia su cui i parser LR si basano è meno intuitiva rispetto a quella dei parser discendenti ricorsivi LL, rendendo difficoltose alcune operazioni necessarie al debug.

In fine, ANTLR gode di una documentazione esaustiva [13] e di una community di sviluppatori particolarmente attiva.
Sono stati utili, nello sviluppo del progetto, i file sorgenti messi a disposizione dalle repository ufficiali di Apache e ANTLR:

- Apache Cassandra (DBMS scritto in java con il supporto di ANTLR) [14]
- Un esempio di grammatica SQL scritta in ANTLR [15]

2.3.2 Definizione di un lexer:

Viene riportato di seguito un frammento di codice rappresentante un esempio di Lexer definito in ANTLR (tratto dalla fonte [16]):

```
1  /*
2   * Lexer Rules
3   */
4  fragment A          : ('A'|'a') ;
5  fragment S          : ('S'|'s') ;
6  fragment Y          : ('Y'|'y') ;
7  fragment H          : ('H'|'h') ;
8  fragment O          : ('O'|'o') ;
9  fragment U          : ('U'|'u') ;
10 fragment T          : ('T'|'t') ;
11 fragment LOWERCASE  : [a-z] ;
12 fragment UPPERCASE  : [A-Z] ;
13 SAYS                : S A Y S ;
14 SHOUTS              : S H O U T S;
15 WORD                : (LOWERCASE | UPPERCASE | '_')+ ;
16 WHITESPACE          : (' ' | '\t') ;
17 NEWLINE             : ('\r'? '\n' | '\r')+ ;
18 TEXT                : ~[\])]+ ;
```

L'esempio mostra come si possano definire sequenze di caratteri per la definizione dei token, utilizzando una sintassi intuitiva che combina elementi della notazione EBNF assieme alla definizione di espressioni regolari.

La parola chiave 'fragment' identifica regole riutilizzabili all'interno della definizione dei token, utilizzata in questo contesto per rendere case-insensitive le parole chiave dichiarate.

2.3.3 Definizione di un traduttore:

Viene riportato di seguito un frammento di codice contenente una porzione rappresentativa del traduttore implementato per questo progetto; l'esempio è stato scelto in modo tale che mostri la quasi totalità dei costrutti messi a disposizione da ANTLR per la definizione di schemi SDT:

```
cartesian returns [FromClauseNode nodeToReturn]:
    n1=join  n2=cartesian_1[$n1.nodeToReturn] {$nodeToReturn = $n2.nodeToReturn;};

cartesian_1[FromClauseNode n] returns [FromClauseNode nodeToReturn]:
    ',' n1=join n2=cartesian_1[new OperationTableNode(n, $n1.nodeToReturn, OperationTableNode.Op.CARTESIAN_PRODUCT , sessionID)] {$nodeToReturn =
$n2.nodeToReturn;}|{$nodeToReturn = n;};

join returns [FromClauseNode nodeToReturn]:
    n1=term n2=join_1[$n1.nodeToReturn]{$nodeToReturn = $n2.nodeToReturn;};

join_1[FromClauseNode n] returns [FromClauseNode nodeToReturn]:
    JOIN n1=term {OperationTableNode joinNode = new OperationTableNode(n, $n1.nodeToReturn, OperationTableNode.Op.JOIN , sessionID);
                  leaves = new ArrayList<>();} n2=join_1[joinNode] ON '(' constraint = booleanExpr ')'
    {
        WhereBooleanTree booleanTree = new WhereBooleanTree($constraint.node, leaves);
        joinNode.setConstraint(booleanTree);
        $nodeToReturn = $n2.nodeToReturn;
    }
    |{$nodeToReturn = n;};

```

In aggiunta alla notazione EBNF è possibile dichiarare:

- Azioni semantiche, racchiuse tra parentesi graffe.
- Valori di ritorno associati a variabili della grammatica, per mezzo della sintassi ‘nomeVariabile returns [tipoDiDato nomeValore]’.
- Parametri associati a variabili della grammatica, per mezzo della sintassi ‘nomeVariabile [tipoDiDato nomeParametro]’.
  La dichiarazione dei parametri permette di implementare attributi ereditati, e quindi di dichiarare schemi SDT ricavati da SDD L-Attribuite.
- Alias associati alle variabili della grammatica chiamate nelle produzioni.

Ogni variabile invocata all’interno di una produzione possiede determinati attributi, alcuni dichiarati all’interno della grammatica (è il caso dei parametri o dei valori di ritorno) e alcuni dichiarati implicitamente per ogni variabile, da ANTLR stesso (un attributo particolarmente importante è il ‘text’, il cui contenuto rappresenta la sequenza di caratteri associati alla produzione applicata sulla variabile chiamata).
Per accedere agli attributi di una variabile all’interno di un’azione semantica è necessario usare la sintassi: ‘$AliasVariabile.nomeAttributo’.

Per ulteriori approfondimenti riguardo la definizione di traduttori mediante ANTLR si consiglia il consulto della fonte [13].

Capitolo 3: Basi di dati

3.1 Modello relazionale:

Il modello relazionale è stato introdotto come modello logico per la rappresentazione e strutturazione dei dati da Edgar Codd nel 1970, permettendo una maggiore indipendenza dei dati e la semplificazione delle interrogazioni alle basi di dati.
Il modello relazionale sfrutta primariamente i concetti di relazione insiemistica e di tabella.

In particolare, per una relazione **R** tra n insiemi (in gergo chiamata n-aria): $\mathbf{I_1}, \mathbf{I_2}, \ldots, \mathbf{I_n}$ è sempre vero che $\mathbf{R} \subseteq \mathbf{I_1}$ x $\mathbf{I_2}$ x … x $\mathbf{I_n}$, pertanto **R** è un insieme i cui elementi sono definibili come collezioni ordinate $(\mathrm{i}_1, \mathrm{i}_2, \ldots, \mathrm{i}_\mathrm{n})$ dove $\forall_{j\in[1,n]}(\mathrm{i_j} \in \mathbf{I_j})$.
Gli elementi contenuti in **R** prendono il nome di tuple.

Sebbene il concetto di relazione risulti piuttosto semplice da definire, nell'ambito delle basi di dati presenta alcune problematiche.
Come fatto notare in precedenza, le tuple contenute all'interno di una relazione sono definite per mezzo di una notazione posizionale piuttosto che simbolica, particolarmente sconveniente da utilizzare per mezzo delle interrogazioni.
Pertanto sarebbe preferibile considerare delle tuple i cui valori vengano identificati per mezzo di attributi.

Si procede dunque a fornire una definizione formale alternativa in grado di risolvere il problema:

- Viene definito un insieme dei domini **D.**
- Viene definito un insieme di attributi **A.**
- Viene definita una nuova funzione *dom*, tale che: $\mathbf{A} \rightarrow \mathbf{D}$.
- Viene riformulato il concetto di tupla per mezzo di una funzione *t*, tale che:
  $\mathbf{A} \rightarrow dom(\boldsymbol{A})$.
- Viene riformulato il concetto di relazione, in modo tale che una relazione **R** definita su **I** è definibile come una collezione di tuple: $\{t \mid t(\mathbf{A})\}$.

Per accedere ai valori degli attributi **A** appartenenti ad una tupla t verrà utilizzata la seguente notazione: $t[\mathbf{A}]$.

Considerata la nuova riformulazione di relazione, è possibile rappresentare convenientemente la stessa per mezzo di una tabella, avente indici per le colonne appartenenti all'insieme degli

attributi della relazione e righe contenenti le tuple, rappresentate con gli attributi in ordine rispetto all'ordine scelto per la rappresentazione delle colonne.

La rappresentazione tabellare di una relazione, seppur conveniente, obbliga necessariamente a scegliere un ordine di presentazione degli attributi, non proprio della relazione così come definita in precedenza.
Pertanto è necessario tenere a mente che per una determinata relazione possono esistere n! tabelle diverse rappresentanti la stessa, con n equivalente alla cardinalità del suo insieme di attributi.

3.1.1 Vincoli di integrità:

Il vincolo di integrità è una proprietà in grado d'esser soddisfatta o meno, applicabile ad istanze della base di dati per esaminarne la consistenza.
In base alla tipologia di istanze coinvolte, i vincoli di integrità posso esser classificati in due categorie:

- Vincoli intra-relazionali: Scelta una relazione è possibile determinare se gli elementi che la compongono rispettano i vincoli relazionali, esaminando come istanze le tuple (nel caso di vincoli di tupla) oppure esaminando i valori degli attributi contenuti in alcune tuple (vincoli di dominio).
- Vincoli inter-relazionali: Vengono verificate le condizioni prendendo in considerazioni tuple di relazioni diverse (vincoli d'integrità referenziale).

Per semplicità verranno esaminati in dettaglio i soli vincoli d'integrità la cui conoscenza è fondamentale per la comprensione del DBMS implementato.

Affinché il modello relazionale possa esser utilizzato in pratica, è necessario saper identificare univocamente una determinata istanza appartenente a una relazione.
Ricordando che una relazione **R** non è altro che un insieme di tuple *t*, definite su un insieme di attributi **A**, siccome il contenuto di un insieme non ammette ripetizioni degli stessi elementi, non può esistere una coppia di tuple $t_1, t_2 \in \mathbf{R}$ tali che $t_1[\mathbf{A}] = t_2[\mathbf{A}]$ con $t_1 \neq t_2$.

Viene definito superchiave di **R** un insieme di attributi **SK** tale che:

$$\mathbf{SK} \subseteq \mathbf{A} \wedge \forall_{t_1, t_2 \in \mathbf{R}} (t_1[\mathbf{SK}] = t_2[\mathbf{SK}] \rightarrow t_1 = t_2).$$

Intuitivamente una superchiave basta ad identificare una qualsiasi istanza della relazione su cui è definita.
Si noti come, viste le considerazioni precedenti, per ogni **R**: **A** è superchiave di **R.**

Un vincolo intra-relazionale di chiave impone che scelta una relazione **R** e scelto un insieme di attributi **I** ⊆ **A** allora: **I** sia superchiave di **R.**
Qualora una determinata tupla rendesse falso il vincolo di chiave non potrebbe essere inserita all'interno della relazione **R.**

Un altro vincolo degno di nota viene chiamato d'integrità referenziale, è utilizzato per referenziare la tupla di una particolare relazione all'interno di una tupla di una relazione differente.
Il vincolo d'integrità referenziale definito sulle tuple di una relazione $\mathbf{R}_1$ verso le tuple di una relazione $\mathbf{R}_2$ impone che scelto un **I**, dato un vincolo di chiave definito su $\mathbf{R}_2$ avente superchiave **SK**:

$$\forall_{t_1 \in \mathbf{R}_1} \exists_{t_2 \in \mathbf{R}_2} (t_1[\mathbf{I}] = t_2[\mathbf{SK}]).$$

3.1.2 Vantaggi e svantaggi:

Rispetto ai principali modelli logici alternativi di rappresentazione dei dati (gerarchici, reticolari e ad oggetti), quello relazionale offre una più netta separazione tra il livello di presentazione dei dati (logico) e il livello di rappresentazione dei dati (fisico), spesso definito per mezzo di puntatori.
Ciò permette, dal punto di vista dell'utilizzatore, di concentrarsi maggiormente sulla natura dei dati indipendentemente dall'architettura fisica sottostante, semplificando notevolmente la definizione e l'interrogazione delle basi di dati.

Ad ogni modo, l'utilizzo del modello relazionale non è sempre giustificato a prescindere dal volume e dalle caratteristiche che la base di dati sarà in grado di possedere.
All'aumentare del numero di tabelle, infatti, molte interrogazioni presentano degradazioni dei tempi di risposta.
Inoltre, l'obbligo di dover definire la base di dati per mezzo di relazioni, per alcune applicazioni non risulta particolarmente efficiente in termini di prestazioni (ne sono un esempio la definizione di gerarchie per aggregazioni OLAP).

3.1.3 Algebra relazionale:

Affinché una base di dati possa esser interrogata, è necessario definire un linguaggio in modo tale che consideri un determinato sottoinsieme di relazioni contenute nella base di dati, e che restituisca una relazione contenente le tuple che soddisfino i criteri dell'interrogazione.

L'algebra relazionale risulta particolarmente importante per quel che riguarda la pianificazione delle query da parte del DBMS, vista la sua caratteristica d'esser un linguaggio procedurale [17].

Viene definita di seguito una lista di operatori appartenenti all'algebra relazionale, in modo tale che nei paragrafi successivi ci si possa ricondurre ad essa per spiegare alcune parti del funzionamento del DBMS:

- Prodotto cartesiano (o cross join): É possibile estendere il prodotto cartesiano alle relazioni per mezzo dell'operazione di giustapposizione applicabile alle tuple:
  $$\mathbf{R_1 x R_2} = \{t_1 t_2 |\, t_1 \in \mathbf{R_1} \wedge t_2 \in \mathbf{R_2}\}.$$
  Così facendo il nuovo insieme di attributi su cui le tuple sono definite diventa $\mathbf{A_1} \cup \mathbf{A_2}$
- Unione, intersezione e differenza: É possibile estendere questi operatori tipici dell'insiemistica alle relazioni come: $\mathbf{R_1}, \mathbf{R_2} \rightarrow \mathbf{R}$, purché $\mathbf{A_1} = \mathbf{A_2}$.
- Ridenominazione: Per via delle limitazioni imposte al punto precedente è necessario talvolta modificare i nomi degli attributi di una relazione, per farlo è possibile usare l'operatore di ridenominazione, intuitivamente:
  $$\rho_{newattr1,newattr2,\dots,newattrn \Leftarrow attr1,attr2,\dots,attrn}(\mathbf{R}).$$
- Selezione: Viene utilizzata la selezione per considerare solo un sottoinsieme di tuple appartenenti ad una relazione che rispetti una condizione booleana, genericamente composta da operatori d'ordine legati per mezzo di connettivi logici:
  $$\sigma_{booleanExp}(\mathbf{R}).$$
  Si noti come $|\sigma_{booleanExp}(\mathbf{R})| \leq |\mathbf{R}|$ $siccome$ $\sigma_{booleanExp}(\mathbf{R}) \subseteq \mathbf{R}$.
- Proiezione: Viene utilizzata la proiezione per considerare solo un sottoinsieme di attributi appartenenti ad una determinata relazione:
  $$\pi_{attr1,attr2,\dots,attrn}(\mathbf{R}), \text{ con } \{attr1, attr2, \dots, attrn\} \subseteq \mathbf{A}.$$
- Join naturale: Viene utilizzato per correlare il contenuto di più relazioni considerando il prodotto cartesiano delle tuple avente valore uguale per gli attributi in comune:
  $$\mathbf{R_1} \bowtie \mathbf{R_2} = \{t_1[\mathbf{A_1}] t_2[\mathbf{A_2} - \mathbf{A_1}] |\, t_1 \in \mathbf{R_1} \wedge t_2 \in \mathbf{R_2} \wedge t_1[\mathbf{A_1} \cap \mathbf{A_2}] = t_2[\mathbf{A_1} \cap \mathbf{A_2}]\}.$$

Inoltre $0 \leq |\mathbf{R}_1 \bowtie \mathbf{R}_2| \leq |\mathbf{R}_1| * |\mathbf{R}_2|$.

Essendo l'algebra relazionale appartenente alla famiglia dei linguaggi liberi dal contesto, ne consegue che sia possibile rappresentare una determinata espressione per mezzo del suo albero sintattico.

Mediante alcune proprietà associate all'applicazione consecutiva degli operatori, è possibile generare alberi sintattici che rappresentano espressioni sintatticamente differenti ma equivalenti dal punto di vista della relazione restituita.

Vengono elencate alcune proprietà fondamentali utilizzate dai DBMS relazionali per generare alberi sintattici alternativi a scopo d'ottimizzazione:

- $\pi_{attr1,attr2,\ldots,attrn}(\mathbf{R}) = \pi_{attr1,attr2,\ldots,attrn}\left(\pi_{attr1,attr2,\ldots,attrn,\ldots,attrm}(\mathbf{R})\right)$.
- $\sigma_{booleanExp1 \wedge booleanExp2}(\mathbf{R}) = \sigma_{booleanExp1}(\sigma_{booleanExp2}(\mathbf{R}))$.
- *proiezioni e selezioni godono infine della proprietà distributiva.*

Per ulteriori informazioni riguardo l'ottimizzazione logica applicabile alle query si consiglia il consulto della fonte [19].

3.2 Linguaggio SQL:

Structured Query Language è un linguaggio standardizzato per l'interazione di basi di dati relazionali, concepito per poter permettere le seguenti operazioni:

- Creare e modificare gli schemi della base di dati, per mezzo di operazioni classificate come di Data Definition Language (abbreviato DDL).
- Inserire e modificare i dati immessi all'interno degli schemi, per mezzo di operazioni classificate come di Data Manipulation Language (Abbreviato DML).
- Interrogare la base di dati per estrarre informazioni, per mezzo di operazioni classificate come di Data Query Language (abbreviato DQL).
- Creare e gestire strumenti di controllo e accesso ai dati, per mezzo di operazioni classificate come di Data Control Language (abbreviato DCL).

Avendo il linguaggio SQL caratteristiche tipiche sia dei linguaggi dichiarativi, sia di quelli procedurali, si dimostra particolarmente versatile a seconda delle esigenze d'utilizzo, tanto da

esser diventato il linguaggio standard de facto per l'interrogazione di una qualsiasi base di dati relazionale.

Dal 1974, SQL è stato aggiornano a differenti versioni, ampliando progressivamente la sintassi accettabile per svolgere via via operazioni sempre più complesse.
Per ragioni di praticità, la tesi tratterà in dettaglio solo un sottoinsieme di costrutti implementati nella versione SQL-92, che apporta dalla prima versione (SQL-86) l'introduzione dei vincoli d'integrità referenziale e la dichiarazione esplicita di join.

Per maggiori informazioni riguardo SQL si consiglia il consulto della fonte [18].

3.2.1 Sintassi DDL principale:

I principali comandi SQL di tipo DDL sono 'Create' e 'Drop', in grado rispettivamente di creare ed eliminare una relazione appartenente alla base di dati scelta.

Ogni relazione all'interno della base di dati viene identificata per mezzo del proprio nome, pertanto è necessario menzionarlo sia in 'Create' che in 'Drop'.
Inoltre, come definito dei paragrafi precedenti, una relazione è una collezione di tuple definite su un insieme di attributi aventi nome e dominio associato.

Viene riportato di seguito esempio intuitivo di sintassi per il comando 'Create':

```
CREATE TABLE nomeTabella
(
nomeAttributo1 tipo_di_dato(size),
nomeAttributo2 tipo_di_dato(size),
nomeAttributo3 tipo_di_dato(size),
FOREIGN KEY (nomeAttributo4) REFERENCES nome_Tabella2(nomeAttributo1)
PRIMARY KEY(nomeAttributo1, nomeAttributo4),
....
);
```

Si noti come FOREIGN KEY permetta di definire vincoli di integrità inter-relazionali come quello di integrità referenziale.

Il comando drop, se utilizzato per eliminare una relazione all'interno della base di dati, assume la seguente sintassi: DROP TABLE nome_Tabella.

3.2.2 Sintassi DML principale:

I principali comandi SQL di tipo DML sono ‘Insert’ e ‘Delete’, in grado rispettivamente di aggiungere e rimuovere delle tuple all’interno di una specifica relazione.

Per mezzo della sintassi seguente è possibile specificare un insieme di tuple da aggiungere ad una relazione:

```
INSERT INTO nomeTabella (Attributo1, Attributo3, Attributo4) VALUES
(Valore1, Valore2, Valore3)
(Valore1, Valore2, Valore3)
...
;
```

É possibile invece eliminare le tuple precedentemente inserite all’interno di una relazione come segue: DELETE FROM nomeTabella WHERE booleanExp.

3.2.3 Sintassi DQL principale:

Il principale comando SQL di tipo DQL è ‘Select’, in grado di interrogare la base di dati:

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

Solo le clausole Select e From sono necessarie alla dichiarazione della query.

In relazione al frammento di codice sopracitato, il funzionamento del comando ‘select’ è così descritto:

- Elenco di attributi: Viene utilizzato per dichiarare gli attributi delle tuple appartenenti alla relazione risultante e, simultaneamente, stabilire un ordine degli attributi per la generazione della tabella associata alla relazione.
  É possibile utilizzare il carattere ‘*’ per preservare tutti gli attributi appartenenti alle tuple della relazione.

- Elenco di funzioni: Viene utilizzato per esplicitare funzioni (talvolta d'aggregazione) applicate alla relazione risultante, generata per mezzo della query.
  Le funzioni più comunemente utilizzate sono: MIN, MAX, AVG, SUM e COUNT.
- Elenco di tabelle: Viene utilizzato per dichiarare sequenze di join di vario tipo.
  Nell'SQL92 i tipi di join più comunemente utilizzati sono i cross join, dichiarati per mezzo della sintassi: tabella_1, tabella_2, …, tabella_n.
  É possibile inoltre dichiarare tipologie di join differenti per mezzo della sintassi: tabella_1 JOIN tabella_2 ON (booleanExp_1) … JOIN ON (booleanExp_n).
  Così facendo verranno eseguite sequenze di prodotti cartesiani le cui tuple risultanti saranno in grado di soddisfare l'espressione: booleanExp_1 ∧ ... ∧ booleanExp_n.
  Inoltre, SQL permette la possibilità di indicare tabelle generate per mezzo di altre query, implementando dunque un meccanismo di subqueries.
- Condizione: Viene espressa una condizione tale per cui tutte le tuple indicate nella relazione restituita dalla query devono soddisfare, analogamente a come svolto dall'operatore di selezione in algebra relazionale.
  Similmente a come descritto per la clausola FROM, sono presenti funzionalità di subqueries in grado di ricavare valori da tabelle differenti rispetto a quelle citate nella clausola FROM.
- Attributi di raggruppamento: Nel caso si intenda utilizzare funzioni d'aggregazione (come AVG, SUM e COUNT), il contenuto della clausola GROUP BY esplicita gli attributi tali per cui se una coppia di tuple possiede quegli attributi uguali, allora rientrano nella stessa categoria d'aggregazione.
- Condizioni di raggruppamento: Le condizioni espresse all'interno della clausola HAVING vengono valutate dopo il calcolo degli attributi aggregati, permettendo una più ampia gamma di selezione non possibile all'interno della clausola WHERE.
- Attributi di ordinamento: É possibile dichiarare un ordine tra gli attributi della relazione (di tipo Ascendente ASC o discendente DESC) tale per cui le tuple della relazione verranno indicate all'interno della tabella.

3.3 RDBMS:

Un Database Management System (DBMS) è un sistema software finalizzato alla gestione delle basi di dati, fornendo tipicamente la possibilità di svolgere operazioni DDL, DML, DQL e DCL su di esse, per mezzo di un linguaggio specifico.
Nel caso in cui il DBMS si occupi di gestire basi di dati relazionali, allora ci si riferisce ad esso come Relational DBMS (abbreviato RDBMS).

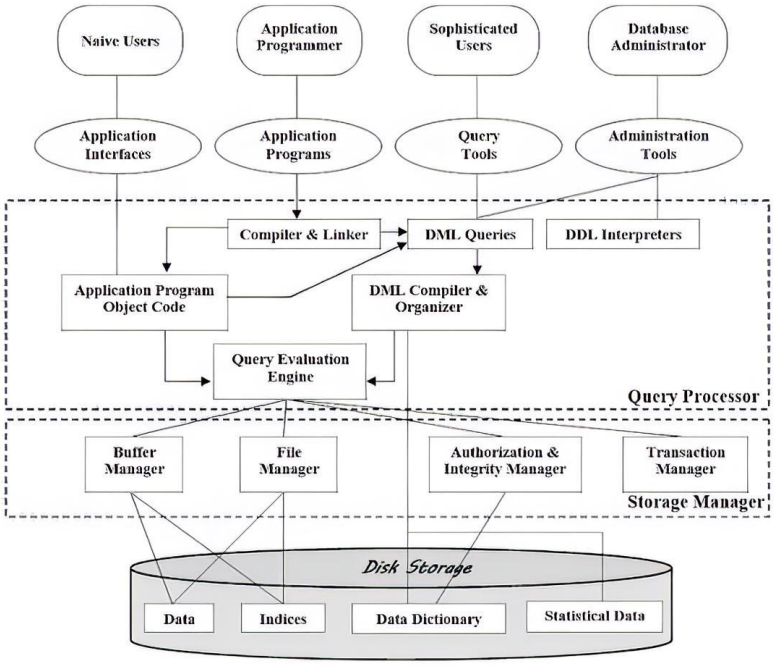
Affinché un DBMS possa essere utilizzato concretamente, è necessario che disponga di un meccanismo tale che permetta l'interrogazione della base di dati da parte di più programmi simultaneamente, la porzione di software incaricata di svolgere questo compito viene chiamata driver (o connettore).

Uno dei concetti cardine alla comprensione del funzionamento di un DBMS è la transazione.
Una transazione identifica una richiesta d'interazione con la base di dati da parte di un'applicazione, rappresentabile come una coppia T_i : <ID_Identificativo:i , query>.

Un DBMS si occupa di gestire transazioni garantendo, per ognuna di esse, il soddisfacimento di proprietà che in gergo vengono chiamate ACID:

- Atomicità: Per un DBMS l'unità indivisibile dei processi che opera deve essere la transazione, pertanto, le istruzioni associate alla query della transazione devono avere effetto sulla base di dati solo se eseguite tutte con successo.
- Coerenza: La base di dati deve rispettare i vincoli d'integrità definiti su di essa, prima e dopo la gestione di una qualsiasi transazione.
- Isolamento: Ogni transazione deve essere eseguita in modo indipendente dalle altre transazioni, l'eventuale fallimento di una transazione non deve influire sull'esecuzione delle altre.
- Durabilità: Una volta espletata una transazione la cui query associata implica l'alterazione della base di dati, allora le modifiche effettuate devono essere permanenti.

Considerata la bassa dipendenza che vi è tra le proprietà ACID, è possibile scomporre l'architettura di un DBMS in moduli in grado di soddisfare singolarmente ciascuna delle proprietà, ne viene riportato di seguito un esempio tipico:



I moduli di maggior interesse al fine di garantire il soddisfacimento delle proprietà ACID sono:

- Gestore delle transazioni: Riceve le transazioni e ne gestisce la loro esecuzione comunicando con gli altri moduli appartenenti alla categoria di Storage Manager.
- Serializzatore: Si occupa di garantire il soddisfacimento della proprietà di isolamento.
- Gestore del ripristino: Si occupa di garantire le proprietà di atomicità e di integrità (in caso d'errore cercando di recuperare i dati corretti oppure forzando l'abort della transazione in corso).
- Gestore del buffer: Si occupa di garantire la proprietà di durabilità.

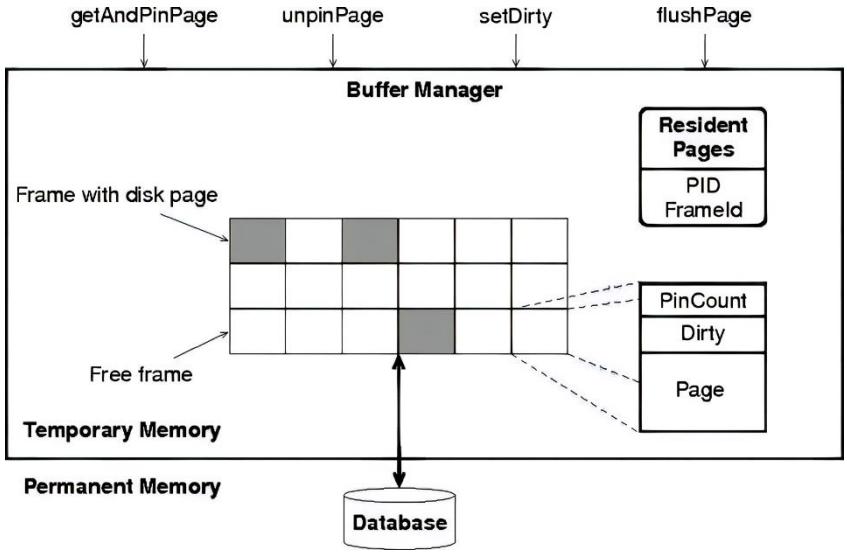
Sebbene il modello relazionale venga utilizzato per formulare le interazioni ad alto livello tra la base di dati e l'applicativo richiedente, un RDBMS necessariamente deve organizzare le rappresentazioni di tuple e tabelle in modo tale che possano essere contenute in un file system.
In genere tutti i record necessari al funzionamento di un RDBMS vengono salvati in pagine, le cui dimensioni dipendono dal file system, solitamente la loro capienza si attesta tra gli 8 e i 64 KB.

Affinché specifiche pagine possano essere caricate e scaricate dal file system è necessario che venga assegnato loro un identificativo univoco, in letteratura chiamato Page Identifier (abbreviato PID).

Le componenti più importanti appartenenti a ciascuna pagina sono:

- Serie di record: Contenente i dati associati alla pagina, solitamente rappresentata per mezzo di una pila
- Array degli offset: Contenente un puntatore all'indirizzo in memoria per ogni elemento appartenente alla serie di record
- Bit set dirty: viene impostato a true se la pagina è stata modificata, ogni volta che la pagina viene scaricata dalla memoria centrale a quella di massa il bit viene impostato a false.

La pagina è così strutturata per garantire l'accesso a un qualsiasi record per mezzo della specifica di una coppia RID: <PID, offset: i>.



Tenendo presente che per accedere a una qualsiasi tupla il DBMS è tenuto a caricare l'intera pagina contenente la stessa (dalla memoria permanente a quella centrale per mezzo del gestore de buffer), per minimizzare il tempo d'esecuzione di una query è necessario spostare il minor numero di pagine possibile.
I tempi necessari per il trasferimento di una pagina dalla memoria permanente a quella centrale si attestano, ad oggi, nell'ordine dei 10^{-3} secondi; quelli per accedere al contenuto di una pagina già presente all'interno della memoria centrale è nell'ordine dei 10^{-9} secondi.
Il costo temporale d'esecuzione di una query è quasi totalmente imputabile ai trasferimenti tra memoria di massa e centrale, il rapporto tra i fattori è nell'ordine di 10^{6}.

Un esempio molto comune di query è quello in cui, selezionata una certa relazione, si desidera ricavare una nuova relazione contenente un sottoinsieme di tuple presenti in quella originale:

select * from relazione where attr = n;

Essendo le tuple di una relazione contenute all'interno di una serie di m pagine, e ponendo la probabilità di trovare una determinata tupla all'interno di una pagina equiprobabile a quella di trovarla in tutte le altre pagine:

- Se attr non è chiave la ricerca impone il consulto di m pagine.
- Se attr è chiave:
    - La ricerca con successo, nel caso medio, impone il consulto di $\frac{1}{m}\sum_{i=1}^{m} i \;=\; \frac{1}{m} * \frac{m(m+1)}{2} = \frac{m+1}{2}$ pagine.
    - La ricerca con fallimento impone il consulto di m pagine.

Ordinando i record all'interno delle pagine la situazione migliora solo marginalmente, riducendo a $\frac{m+1}{2}$ le pagine consultate in ogni scenario considerato precedentemente, gli accessi alle pagine restano comunque nell'ordine di $O(m)$.

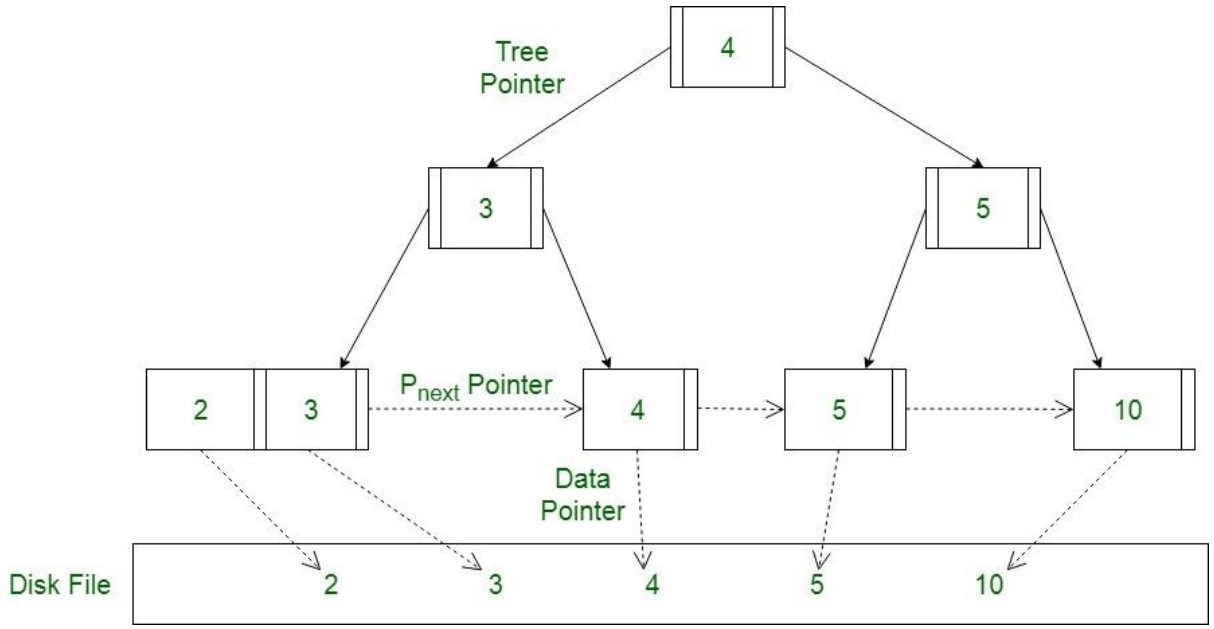
Considerato che in un sistema sufficientemente complesso il numero di pagine associate ad una relazione può superare anche 10^6, è necessario stabilire una differente strategia definendo una struttura dati secondo cui rappresentare le pagine in funzione dell'ordine stabilito su specifici attributi appartenenti alle tuple della relazione, il cui nome viene definito come indice.

Strutture dati largamente utilizzate per la realizzazione di indici sono gli alberi [20], nei successivi paragrafi tratteremo in particolare la definizione degli stessi per mezzo dei B+Tree.

Un B+Tree con branching factor equivalente ad m è un albero $(m-1)$-ario che presenta le seguenti caratteristiche:

- Massimo numero di chiavi per nodo: $m-1$.
- Minimo numero di chiavi per nodo interno: $\left\lceil \frac{m}{2} - 1 \right\rceil$.
- Numero di chiavi all'interno del nodo radice: compresi tra $[1, m-1]$.
- Se un nodo non foglia contiene n chiavi allora il nodo dovrà avere n+1 figli.
- L'albero dev'essere bilanciato.

- Le chiavi associate ad un nodo vengono rappresentate in modo ordinato.
- Ogni foglia, avente valore massimo x, punta al più a un'altra foglia, avente valore minimo $y > x$, tale che tra tutte le foglie vale: $min(y - x)$.



Per mezzo del master theorem [21] è possibile dimostrare che l'algoritmo in grado di trovare una specifica chiave all'interno del B+Albero sopra descritto è eseguibile, nel caso peggiore, in $O(\log_m n)$ operazioni (con n equivalente al numero di chiavi inserite nell'albero).

In particolare, è possibile definire un indice su una relazione **R** per mezzo di un attributo *A* immettendo in un B+Tree entry della forma**:**$< k \in dom(A), Lista\_di\_RID >$**.**

A seconda della tipologia di attributi su cui l'indice è definito è possibile classificarlo in principale, se definito sulla chiave primaria, o secondario altrimenti.
Per una relazione è possibile definire al più un indice principale e zero o più indici secondari.

Sebbene in determinate circostanze l'introduzione di un indice risulti particolarmente vantaggioso, definire indici su qualsiasi attributo non risulta essere una scelta efficace.
Il costo di un inserimento o di una modifica all'interno di un indice definito per mezzo di un B+Tree resta $O(\log_m n)$ nel caso peggiore.
Esistono anche particolari casi in cui il DBMS, per mezzo di alcune euristiche, sceglie di consultare un indice piuttosto che un altro a seconda del livello di selettività degli attributi su cui sono definiti, e per tabelle sufficientemente piccole è possibile caricare in memoria centrate tutte le pagine all'interno del buffer.

Pertanto vengono elencate alcune direttive generiche per la definizione di indici su relazioni:

- Non definirli per tabelle aventi poche pagine.
- Non definirli per attributi il cui valore viene aggiornato soventemente.
- Non definirli su attributi i cui valori sono poco selettivi.
- Non definirli su attributi i cui valori all'interno della relazione non sono uniformemente distribuiti.
- Definirli su chiavi primarie ed esterne.
- Definirli su tabelle le cui query necessitano di mostrare tabelle ordinate rispetto a determinati attributi.

Nel caso il DBMS si trovasse nella condizione di valutare la query (con $b_1 \wedge \ldots \wedge b_n$ predicati logici soddisfacibili con l'ausilio di indici e β una serie di predicati per cui un indice non è stato definito):

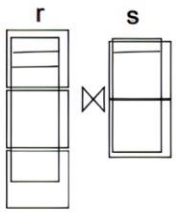
select * from relazione where $b_1 \wedge \ldots \wedge b_n \wedge \beta$;

le principali strategie adottabili sarebbero generalmente due:

- Calcolo di $\bigcap_{i=1}^{n}\{RID\}_{b_i}$ per mezzo dei vari indici, per poi valutare in memoria centrale β: presenta l'inconveniente di calcolare l'intersezione tra gli insiemi, un'operazione computazionalmente onerosa per cardinalità di $\{RID\}_{b_i}$ sufficientemente grandi.
- Considerato il predicato b_j risolvibile per mezzo dell'indice I_w più selettivo disponibile, viene calcolato $\{RID\}_{b_j}$ tramite I_w, per poi calcolare in memoria centrale $b_1 \wedge \ldots \wedge b_{j-1} \wedge \ldots \wedge b_{j+1} \wedge \ldots \wedge b_n \wedge \beta$: nella maggioranza dei casi smuove più pagine del metodo elencato precedentemente ma presenta il vantaggio di non dover calcolare l'insieme intersezione.

Viene riportato in fine uno dei più diffusi algoritmi per il calcolo di theta-join, il Block Nested-Loop Join:

```
for each block B_r of r do begin
    for each block B_s of s do begin
        for each tuple t_r in B_r do begin
            for each tuple t_s in B_s do begin
                Check if (t_r,t_s) satisfy the join condition
                if they do, add t_r • t_s to the result.
            end
        end
    end
end
```



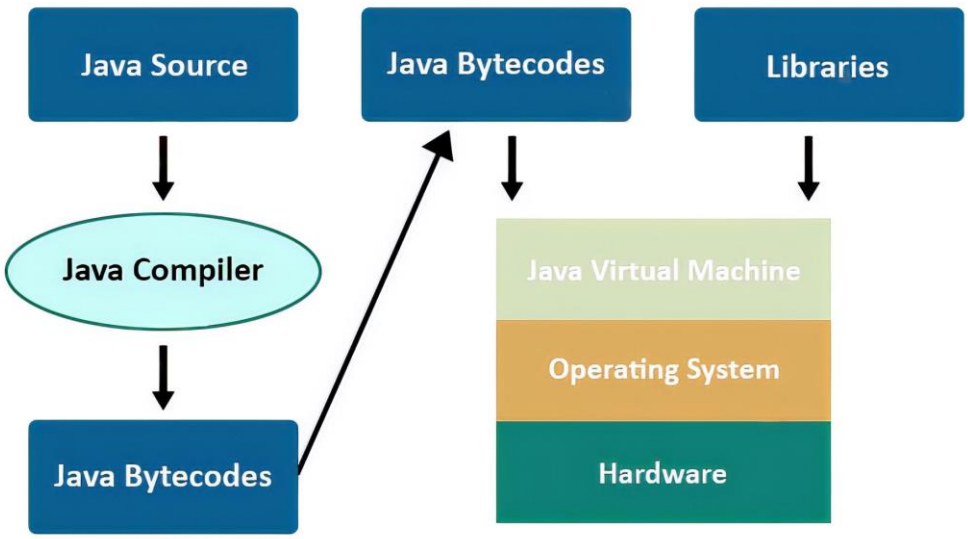
Il cui numero di blocchi trasferiti è compreso tra $b_r + b_s$ e $b_r * b_s + b_r$, a seconda della capienza del buffer in rapporto alla dimensione del singolo blocco.

Capitolo 4: Progetto

4.1 Perché l'utilizzo del linguaggio Java?:

L'utilizzo di Java come linguaggio implementativo per la realizzazione di un DBMS potrebbe destare numerosi dubbi riguardo le relative conseguenze sulle prestazioni del software prodotto, perché non utilizzare linguaggi non interpretati come C++?

Sebbene vi sia un'inevitabile degradazione delle prestazioni, l'utilizzo di un interprete spesso fornisce un supporto più robusto per quel che riguarda le funzionalità di sicurezza, inoltre permette un approccio multi-piattaforma riducendo peraltro la dimensione dell'eseguibile prodotto.
Java, in particolare, viene compilato per mezzo del compilatore JavaC, incaricato di generare il bytecode risultante dal codice sorgente, e successivamente eseguito per mezzo della Java Virtual Machine (abbreviata JVM).



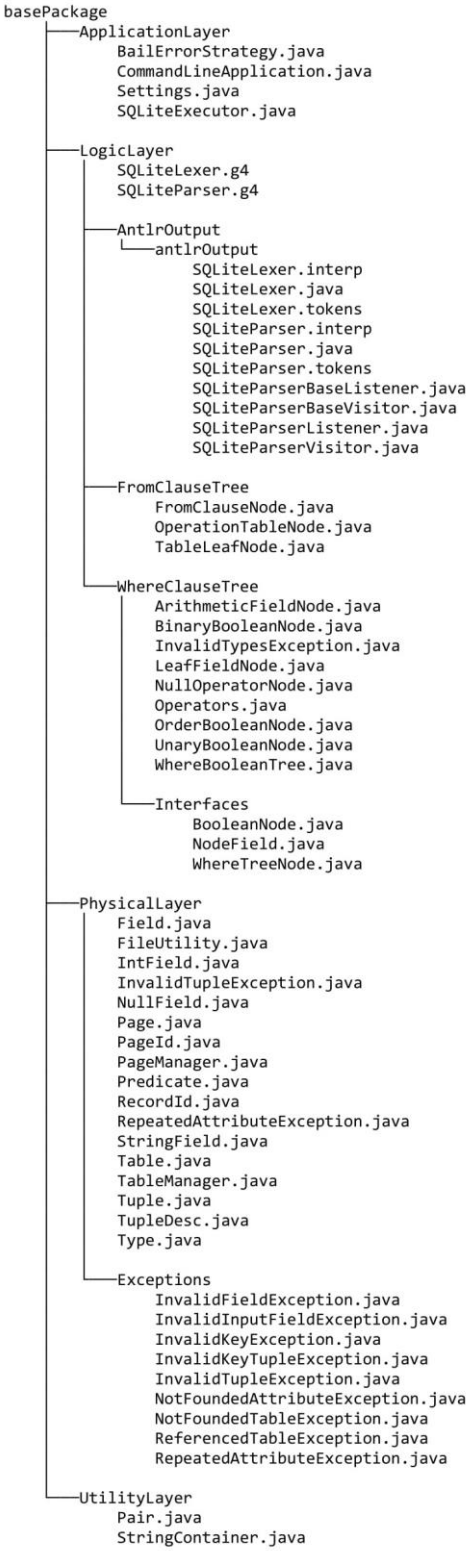
Considerate funzionalità come:

- Gestione automatica della memoria (per mezzo del Garbage Collector)
- Nessun riferimento a puntatori espliciti
- Gestione delle eccezioni
- Compilazione intermedia JIT

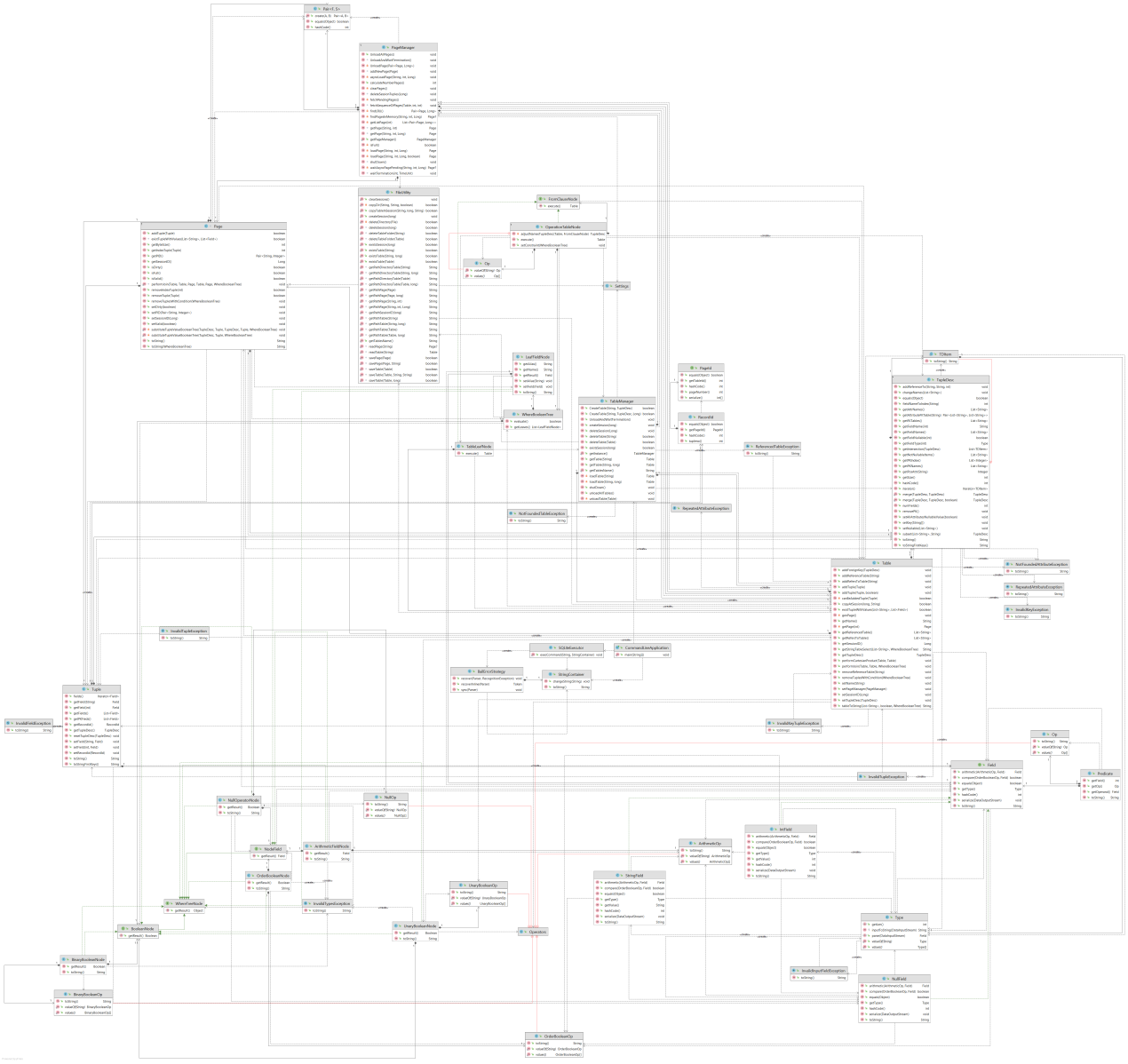
la scelta di Java permette di risolvere efficacemente il trade-off imposto tra prestazioni del software prodotto e tempistiche dedicate allo sviluppo e ai test.

4.2 Struttura del progetto:

Viene riportato di seguito un albero rappresentate la suddivisione delle classi progettuali in package:



Viene inoltre riportato il diagramma delle classi UML associato al progetto:



Layer fisico:

Le classi contenute nel package supportano la rappresentazione dei concetti di tipo, tupla, pagina e relazione, descritti all'interno del secondo capitolo del documento (è stata usata la fonte [22] come spunto per la progettazione di alcune funzionalità delle classi sopracitate).

É stata fissata una dimensione delle pagine pari a 16Kbyte ed è stata gestita la persistenza dei dati mediante i meccanismi di serializzazione di oggetti messi a disposizione nativamente da Java.

La classe PageManager ricopre un ruolo di particolare importanza all'interno del layer fisico, in quanto permette di garantire le proprietà di atomicità e durabilità.
Tenendo presente l'odierno divario di velocità in scrittura e lettura tra memoria secondaria e memoria centrale (descritto in maniera più approfondita nel paragrafo dedicato ai DBMS), è stato scelto di mediare il passaggio di dati tra memoria centrale e secondaria (e viceversa) per mezzo di tecniche multithreading.
Mediante una threadpool, i cui parametri sono modificabili per mezzo della classe settings, è possibile richiedere al PageManager il fetch asincrono di sequenze di multiple di pagine, la stessa threadpool è incaricata di salvare le pagine rimosse dal buffer all'interno degli appositi file dedicati alla rappresentazione delle pagine nel filesystem.

Le classi Table e TableManager permettono di garantire la proprietà di consistenza.

Layer logico:

Le classi contenute nel layer logico permettono l'intepretazione delle query SQL mediante l'utilizzo dei componenti specificati all'interno del layer fisico.
Viene descritta all'interno del package la struttura del traduttore in linguaggio ANTLR, contenuta nel file SQLiteParser.g4, e vengono riportate le classi generate successivamente da ANTLR all'interno del package antlrOuput.

Layer applicativo:

Per limitare la complessità del DBMS, è stato scelto di posticipare a futuri sviluppi l'implementazione di un driver vero e proprio dedicato alla comunicazione diretta tra il DBMS e gli applicativi.
All'interno del package sono state specificate alcune classi che permettono di inviare, in maniera diretta, comandi SQL al DBMS per mezzo di una shell.
Affinché il programma possa rappresentare il risultato delle query in maniera corretta, è

necessario che il buffer delle righe della shell ,su cui la classe CommandLineApplication verrà eseguita, sia sufficientemente grande.

4.3 Costrutti SQL implementati:

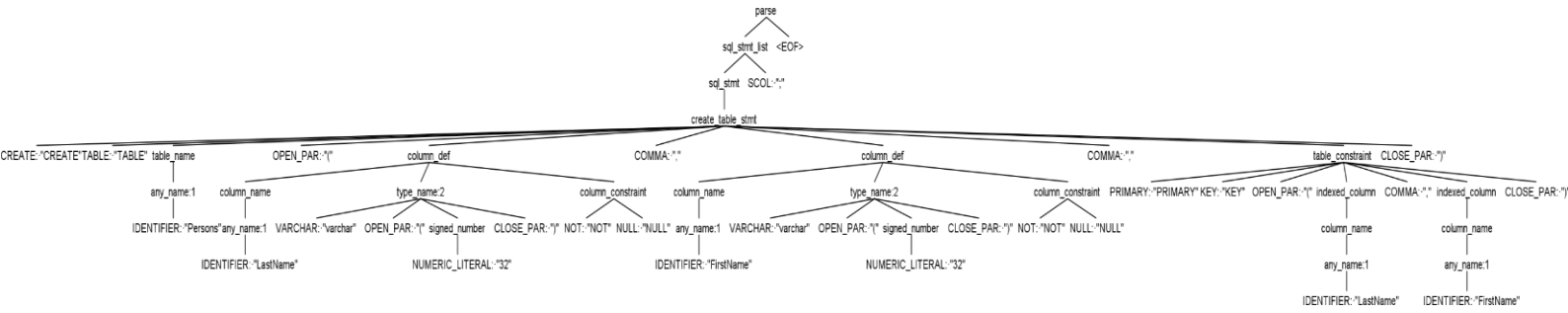
Nel corso del paragrafo seguente verranno specificate le grammatiche associate alle porzioni di traduttore in grado di interpretare query DDL, DML e DQL, verranno anche illustrati esempi di alberi di parsificazione generati per mezzo della grammatica per alcune query tipiche del linguaggio.
Quando possibile si preferirà esplicitare la struttura della grammatica per mezzo di un grafo sintattico [23], piuttosto che utilizzare direttamente la meta-sintassi EBNF.

Parsificazione DDL:

La porzione di grammatica trattata supporta tutti i tipi di costrutti introdotti all'interno del paragrafo dedicato al linguaggio SQL, in particolare è possibile dichiarare FOREIGN KEY e PRIMARY KEY, nel caso una primary key non venga definita il DBMS porrà chiave primaria l'insieme di attributi su cui la relazione è definita.
Per semplicità, il DBMS supporta come tipi di dato: VARCHAR e INT, l'approccio con cui è stato scritto garantisce una futura espansione dei tipi accettati dal linguaggio senza dover riformulare la struttura del traduttore o delle classi utili all'interpretazione DDL.

Albero di parsificazione costruito sulla query:

```
CREATE TABLE Persons (
    LastName varchar(32) NOT NULL,
    FirstName varchar(32) NOT NULL,
    PRIMARY KEY (LastName,  FirstName)
);
```



Parsificazione DML:

Così come nel caso precedente, è stata definita una grammatica in grado di accettare tutti i costrutti DML trattati del paragrafo dedicato alla sintassi SQL.
É possibile, tra l'altro, esplicitare solo alcuni attributi di una relazione purché gli attributi omessi siano nullable, variando anche l'ordine con cui sono stati definiti all'interno della query DDL di creazione della relazione.

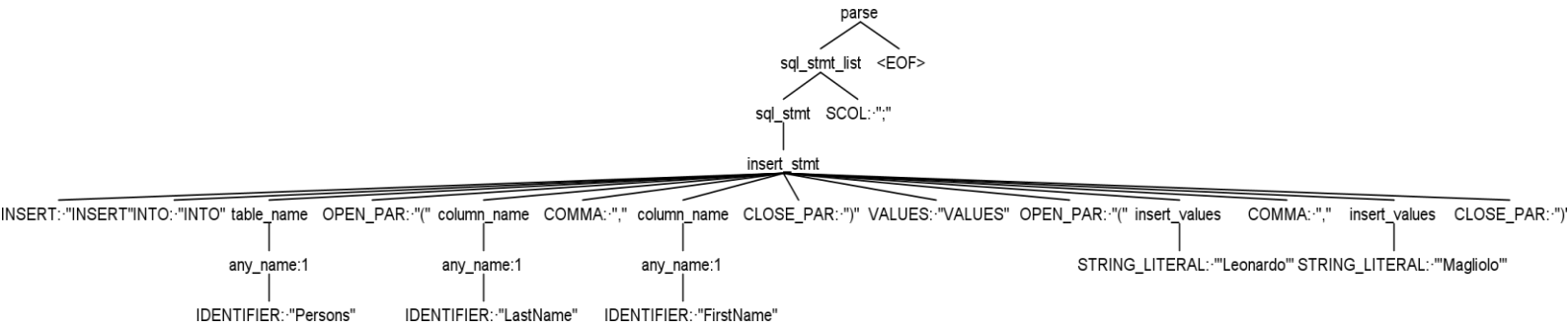
Vista la struttura più complessa della grammatica considerata, tenendo presente che i grafi sintattici talvolta presentano l'inconveniente di estendersi eccessivamente verso destra, si è deciso di rappresentare la grammatica per mezzo della sintassi EBNF:

```
insert_stmt:
    (
        INSERT
    ) INTO  table_name(
        '(' column_name
         ( ',' column_name)*')'
    ) (
        (
            (
                VALUES '(' insert_values
                           (',' insert_values )*
                       ')'
                (
                    ',' '(' insert_values) ')'
                    ( ',' insert_values)*
                )
            )
        )
    )
);

insert_values:
    NUMERIC_LITERAL
    | STRING_LITERAL;

column_name: any_name;

table_name: any_name;
```

Albero di parsificazione costruito sulla query:

INSERT INTO Persons (LastName , FirstName)

VALUES ('Leonardo', 'Magliolo');

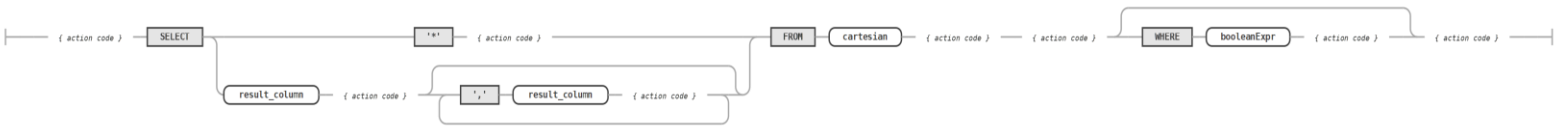


Parsificazione DQL:

Sebbene i costrutti fondamentali delle interrogazioni DQL siano stati ultimati, alcuni aspetti marginali necessiteranno ulteriori sviluppi per essere implementati.

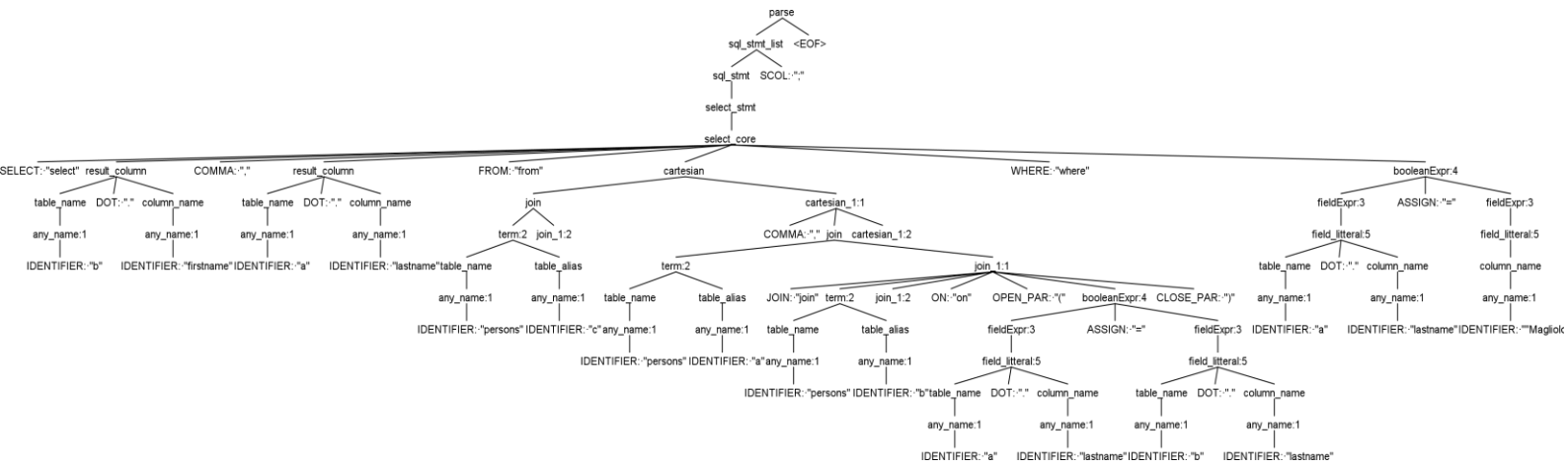
Alla sua attuale versione, il DBMS non dispone della possibilità di poter usufruire delle funzioni, di poter ordinare i record o di poter eseguire subquery.

Resta comunque possibile la definizione di alias per le relazioni, la possibilità di selezionare le tuple per mezzo delle clausole WHERE e JOIN ON utilizzando predicati booleani, e resta infine possibile utilizzare la wildcard * per mantenere tutti gli attributi o utilizzare il contenuto della lista di attributi dopo il comando SELECT per selezionare solo alcuni attributi della relazione risultante, anche in ordine sparso.



Albero di parsificazione costruito sulla query:

select b.firstname, a.lastname from persons c, persons a join persons b on (a.lastname = b.lastname) where a.lastname = "Magliolo";



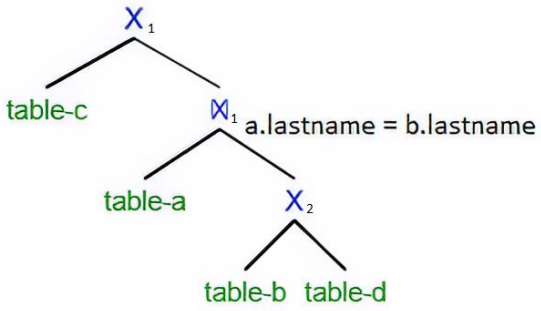
Sono state effettuate ottimizzazioni riguardanti la porzione di traduttore che si occupa di costruire l'albero sintattico dell'espressione relazionale associata alla query SQL.
In particolare, è stata riformulata la grammatica del traduttore in modo tale prioritizzare l'esecuzione dei theta-join piuttosto che dei prodotti cartesiani, spostando verso la frontiera i nodi join quando possibile.

```
1  cartesian returns [FromClauseNode nodeToReturn]:
2      n1=join  n2=cartesian_1[$n1.nodeToReturn] {$nodeToReturn = $n2.nodeToReturn;};
3
4  cartesian_1[FromClauseNode n] returns [FromClauseNode nodeToReturn]:
5      ',' n1=join n2=cartesian_1[new OperationTableNode(n, $n1.nodeToReturn, OperationTableNode.Op.CARTESIAN_PRODUCT , sessionID)] {$nodeToReturn =
   $n2.nodeToReturn;}|{$nodeToReturn = n;};
6
7  join returns [FromClauseNode nodeToReturn]:
8      n1=term n2=join_1[$n1.nodeToReturn]{$nodeToReturn = $n2.nodeToReturn;};
9
10 join_1[FromClauseNode n] returns [FromClauseNode nodeToReturn]:
11     JOIN n1=term {OperationTableNode joinNode = new OperationTableNode(n, $n1.nodeToReturn, OperationTableNode.Op.JOIN , sessionID);
12                   leaves = new ArrayList<>();} n2=join_1[joinNode] ON '(' constraint = booleanExpr ')'
13     {
14         WhereBooleanTree booleanTree = new WhereBooleanTree($constraint.node, leaves);
15         joinNode.setConstraint(booleanTree);
16         $nodeToReturn = $n2.nodeToReturn;
17     }
18     |{$nodeToReturn = n;};
19
```

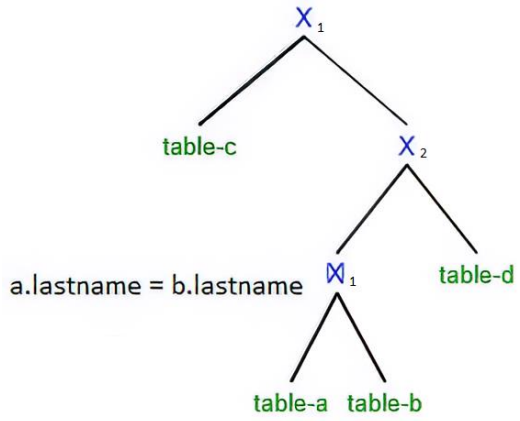
Prendendo in considerazione la porzione di query d'esempio:

persons c, persons a join persons b on (a.lastname = b.lastname), persons d

vengono riportati due possibili alberi sintattici differenti restituenti la medesima relazione:



(Albero non ottimizzato, generato considerando soltanto l'ordine di precedenza da sinistra verso destra)



(Albero ottimizzato, generato considerando l'operatore di join con precedenza maggiore rispetto al prodotto cartesiano)

Essendo l'ordine di valutazione bottom-up, i nodi più vicini alla frontiera dell'albero verranno eseguiti prima.
Per dimostrare che l'albero ottimizzato sia effettivamente in grado di smuovere meno tuple di quello non ottimizzato, si procede al calcolo delle tuple mosse:

Per l'albero non ottimizzato:

- Costo del prodotto cartesiano X_2: $|\mathbf{b}| * |\mathbf{d}|$.
- Costo del theta-join $\bowtie_{1\ \theta:\ a.lastname\ =\ b.lastname}$: $(|\mathbf{b}| * |\mathbf{d}|) * |\boldsymbol{a}|$.
- Costo del prodotto cartesiano X_1: $\alpha * |\boldsymbol{c}|, con\ 0 \leq \alpha \leq (|\mathbf{b}| * |\mathbf{d}|) * |\boldsymbol{a}|$.
- Somma dei costi: $\Phi = |\mathbf{b}| * |\mathbf{d}| + (|\mathbf{b}| * |\mathbf{d}|) * |\boldsymbol{a}| + \alpha * |\mathbf{c}|$.

Per l'albero ottimizzato:

- Costo del theta-join $\bowtie_{1\ \theta:\ a.lastname\ =\ b.lastname}$: $|\mathbf{a}| * |\mathbf{b}|$.
- Costo del prodotto cartesiano X_2: $\beta * |\mathbf{d}|, con\ 0 \leq \beta \leq |\mathbf{a}| * |\mathbf{b}|$.
- Costo del prodotto cartesiano X_1: $(\beta * |\mathbf{d}|) * |\mathbf{c}|$.
- Somma dei costi: $\varphi = |\mathbf{a}| * |\mathbf{b}| + \beta * |\mathbf{d}| + (\beta * |\mathbf{d}|) * |\boldsymbol{c}| = |\mathbf{a}| * |\mathbf{b}| + \beta * |\mathbf{d}| * (1 + |\boldsymbol{c}|)$.

Per evitare di considerare ulteriori ottimizzazioni, si supponga che $|\mathbf{a}|, |\mathbf{b}|\ e\ |\mathbf{c}|$ appartengano allo stesso ordine di grandezza.

In questo caso, siccome $|\boldsymbol{a}| \approx |\mathbf{b}| \approx |\mathbf{c}|$, nella maggioranza dei casi $\varphi < \Phi$ siccome $\beta < \alpha$ e $\beta * |\mathbf{d}| < |\mathbf{b}| * |\mathbf{d}| * |\boldsymbol{a}|$.

Nello specifico, per determinare con una certa probabilità il valore di β è necessario poter calcolare la selettività del predicato θ per mezzo delle statistiche raccolte dal DBMS sulle tabelle interessate dal theta-join.

4.4 Principali strutture dati e algoritmi utilizzati:

4.4.1 Risolutore espressioni clausole condizionali:

Le clausole condizionali dichiarabili in SQL sfruttano operatori classificabili principalmente in: connettivi logici, operatori d'ordine, operatori aritmetici, operatori speciali che sfruttano la logica a valori nulli e funzioni; sono inoltre suddivisibili in unari oppure binari.

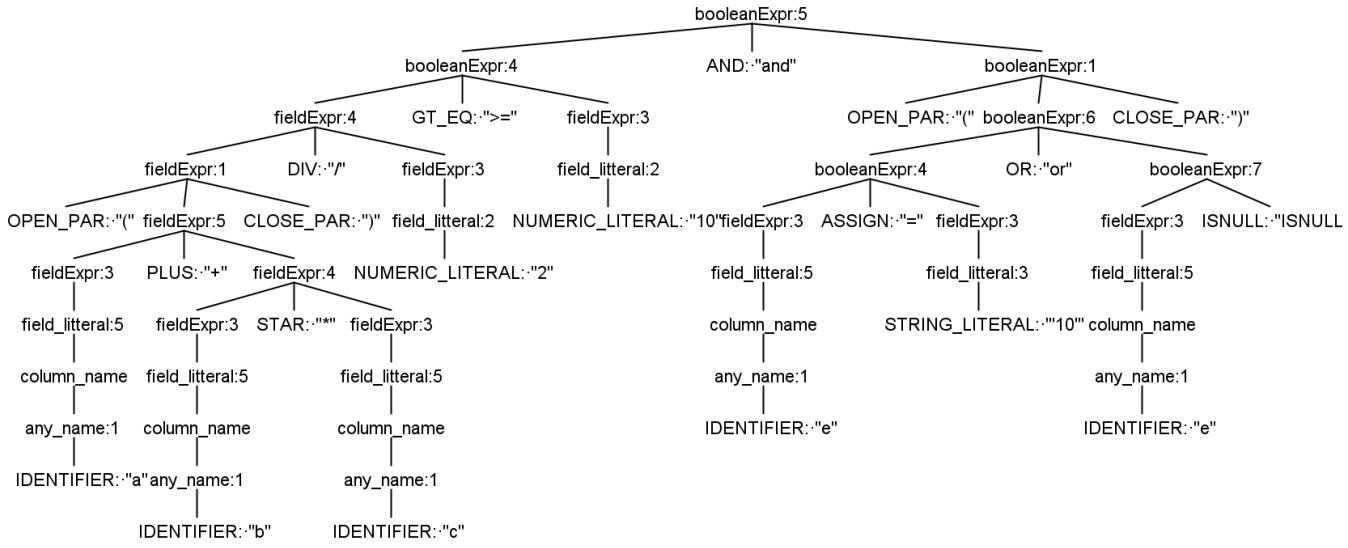
La versione di SQL implementata supporta:

- Connettivi logici: Per mezzo della classe BynaryBooleanNode è possibile definire operatori AND, OR, per mezzo della classe UnaryBooleanNode è possibile definire l'operatore NOT.
- Operatori d'ordine: Definibili come una funzione **Field** x **Field** → {**True**, **False**} implementata per mezzo della classe OrderBooleanNode.
  Gli operatori d'ordine disponibili includono: $=,\ >, \geq, <, \leq\ e\ <>$.
  Si noti come la seguente definizione non permetta sintassi del tipo $m < a < n$.
  È comunque possibile esprimere una condizione equivalente per mezzo dei connettivi logici: $m < a\ AND\ a < n$.
- Operatori aritmetici: Definibili come una funzione **Field** x **Field** → **Field** implementata per mezzo della classe ArithmeticFieldNode.
  Gli operatori aritmetici disponibili includono: +, *, -, /.
- Operatori speciali basati su logica a valori nulli: Per mezzo di NullOperatorNode è possibile definire l'operatore ISNULL.
  Abbinato ai connettivi logici è possibile esprimere la condizione a NOT NULL mediante la seguente espressione: NOT (a ISNULL).

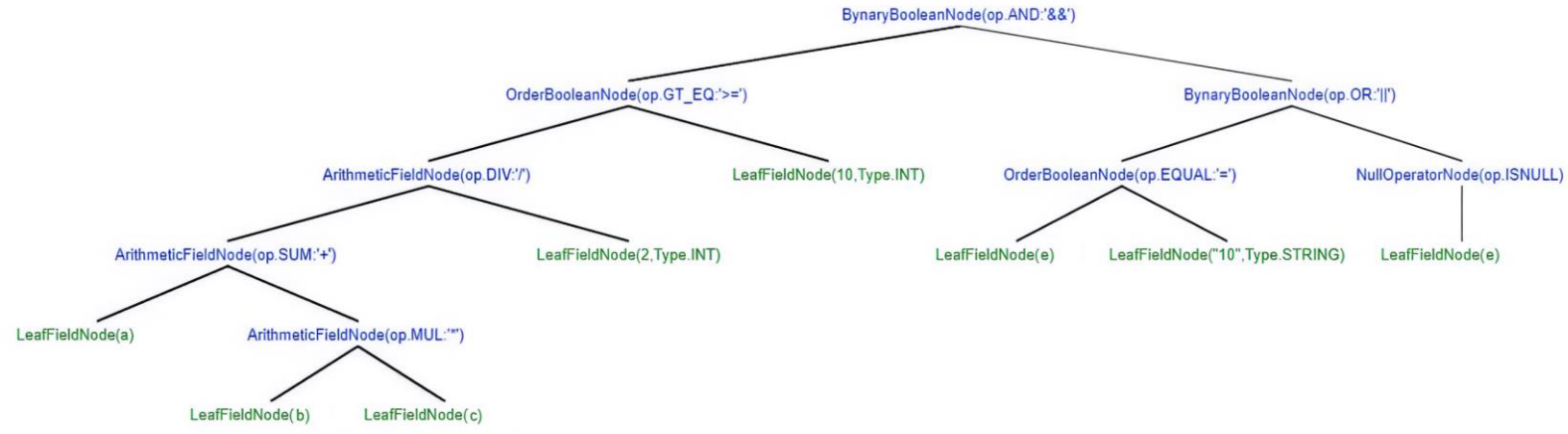
Gli operatori della grammatica sono valutati in ordine di priorità da sinistra verso destra: aritmetici, d'ordine e logici.
Come discusso nei paragrafi precedenti, essendo l'ordine di valutazione bottom-up, è necessario che la grammatica parsifichi prima gli elementi del linguaggio avente priorità minore.

Viene riportato l'albero di parsificazione, generato dalla porzione di grammatica SQL, costruito sulla condizione d'esempio:

(a+b*c)/2 >= 10 and (e = '10' or e ISNULL).



Essendo la condizione da applicare a tutte le tuple appartenenti ad una certa relazione, è stato scelto di generare, per mezzo del traduttore, un albero equivalente le cui foglie rappresentano costanti (Field) oppure attributi associati alla relazione:



L’albero verrà utilizzato successivamente da un algoritmo in grado di sostituire i valori all’interno delle sue foglie con i campi fields associati a ciascuna delle tuple interessate dalla relazione:

```java
private static void substituteTupleValueBooleanTree(TupleDesc tdescLeft, Tuple tupleLeft, TupleDesc tdescRight,
                            Tuple tupleRight, WhereBooleanTree tree) throws NotFoundedAttributeException {
        if (tree != null && tree.getLeaves().size() > 0) {
            List<String> namesLeft = tdescLeft.getAttrNames();
            List<String> namesRight = null;
            if(tdescRight != null){
                namesRight = tdescRight.getAttrNames();
            }
            if (tree != null) {
                for (LeafFieldNode leaf : tree.getLeaves()) {
                    boolean founded = false;
                    String nameLeaf = leaf.getName();
                    if (!leaf.getAlias().equals("")) {
                        nameLeaf = leaf.getAlias() + "." + nameLeaf;
                    }

                    for (int i = 0; i < namesLeft.size(); i++) {
                        if (namesLeft.get(i).equals(nameLeaf)) {
                            leaf.setField(tupleLeft.getField(i));
                            founded = true;
                            break;
                        }
                    }

                    if (!founded && namesRight != null) {
                        for (int i = 0; i < namesRight.size(); i++) {
                            if (namesRight.get(i).equals(nameLeaf)) {
                                leaf.setField(tupleRight.getField(i));
                                founded = true;
                                break;
                            }
                        }
                    }

                    if (!founded) {
                        throw new NotFoundedAttributeException(nameLeaf);
                    }
                }
            }
        }
    }
```

4.4.2 Implementazione Block Nested Loop Join:

Al fine di mantenere un accoppiamento basso e al tempo stesso favorire la coesione delle funzioni implementate all'interno delle classi, è stato scelto di poter accedere direttamente ad una determinata tupla solo mediante l'istanza di pagina che la contiene.
Per questa ragione, l'implementazione dell'algoritmo Block Nested Loop Join ha richiesto la specifica dello stesso in due parti:

1. Codice relativo alla classe Table:

```
public void performJoin(Table leftTable, Table rightTable, WhereBooleanTree tree)
            throws RepeatedAttributeException,... NotFoundedAttributeException{
        ...
        if(leftTable.lastID.get() > 0 && rightTable.lastID.get() > 0){
            ...
            for(int i = 1; i<=leftTable.lastID.get(); i++){
                Page leftPage = leftTable.getPage(i);
                for(int j = 1; j<=rightTable.lastID.get(); j++){
                    Page.performJoin(this, leftTable, leftPage, rightTable, rightTable.getPage(j), tree);
                }
            }
        }
    }
```

2. Codice relativo alla classe Page:

```
static void performJoin(Table table, Table leftTable, Page leftPage, Table rightTable, Page rightPage, WhereBooleanTree tree)
        throws RepeatedAttributeException,... NotFoundedAttributeException{
        for(Tuple tupleLeft: leftPage.tuples){
            for(Tuple tupleRight: rightPage.tuples){
                substituteTupleValueBooleanTree(leftTable.getTupleDesc(), tupleLeft, rightTable.getTupleDesc(), tupleRight, tree);
                if(tree.evaluate()){ //perform the following actions only if the theta conditions is true
                    Tuple toAdd = new Tuple(table.getTupleDesc());
                    List<Field> fields = tupleLeft.getFields();
                    for(int i = 0; i < fields.size(); i++){ //fill the left side of the new tuple
                        toAdd.setField(i, fields.get(i));
                    }
                    int prevLen = fields.size();
                    fields = tupleRight.getFields();
                    for(int i = prevLen; i < (fields.size()+prevLen); i++){//fill the right side of the new tuple
                        toAdd.setField(i, fields.get(i-prevLen));
                    }

                    table.addTuple(toAdd, false); // false allow to not perform integrity checks
                }
            }
        }
    }
```

Si noti come nella seconda porzione di codice, alla quinta riga, venga chiamata la funzione *substituteTupleValueBooleanTree* per sostituire i valori delle foglie dell'albero condizionale in modo tale che possa essere valutata la condizione θ sulla tupla considerata.

Capitolo 5: Conclusioni

La progettazione e l'implementazione del software prodotto ha permesso di esaminare più a fondo le interazioni tra gli insegnamenti di linguaggi formali, traduttori e basi di dati, atte alla realizzazione e al funzionamento degli odierni RDBMS.

Sebbene il progetto sia in grado d'eseguire le principali funzioni DDL, DML e DQL, necessarie affinché un DBMS possa essere usato concretamente, la quantità limitata di tempo e la complessità del progetto trattato giustificano migliorie introducibili in futuri sviluppi:

- Sviluppo di un driver dedicato alla comunicazione tra applicativi esterni e DBMS
- Implementazione di più tipi accettati dalla sintassi DDL
- Estensione della sintassi DQL, applicabile all'interpretazione delle funzioni e delle sub-queries
- Implementazioni di indici mediante B+Tree
- Ottimizzazioni logiche ulteriori, introducibili manipolando l'albero di parsificazione delle query DQL interpretate

Fonti:

1. https://it.wikipedia.org/wiki/Linguaggio_formale

2. https://it.wikipedia.org/wiki/Gerarchia_di_Chomsky

3. Alfred Vaino Aho; Monica S. Lam; Ravi Sethi; Jeffrey D. Ullman (2007). "3.7.4 Construction of an NFA from a Regular Expression" (print). Compilers : Principles, Techniques, & Tools (2nd ed.). Boston, MA, USA: Pearson Addison-Wesley. p. 159–163. ISBN 9780321486813

4. Alfred Vaino Aho; Monica S. Lam; Ravi Sethi; Jeffrey D. Ullman (2007). "3.3.3 Regular Expression" (print). Compilers : Principles, Techniques, & Tools (2nd ed.). Boston, MA, USA: Pearson Addison-Wesley. p. 120–125. ISBN 9780321486813

5. Alfred Vaino Aho; Monica S. Lam; Ravi Sethi; Jeffrey D. Ullman (2007). "4.2.5 Ambiguity" (print). Compilers : Principles, Techniques, & Tools (2nd ed.). Boston, MA, USA: Pearson Addison-Wesley. p. 203 ISBN 9780321486813

6. https://en.wikipedia.org/wiki/Translator_(computing)

7. https://www.researchgate.net/publication/318477435

8. https://it.wikipedia.org/wiki/Ricerca_in_profondità

9. Alfred Vaino Aho; Monica S. Lam; Ravi Sethi; Jeffrey D. Ullman (2007). "6.5 Type Checking" (print). Compilers : Principles, Techniques, & Tools (2nd ed.). Boston, MA, USA: Pearson Addison-Wesley. p. 378-384 ISBN 9780321486813

10. https://computerscience.unicam.it/diberardini/didattica/lpc/2009-10/slides/SDD.pdf

11. Alfred Vaino Aho; Monica S. Lam; Ravi Sethi; Jeffrey D. Ullman (2007). "5 Syntax Directed Translation" (print). Compilers: Principles, Techniques, & Tools (2nd ed.). Boston, MA, USA: Pearson Addison-Wesley. p. 303-354 ISBN 9780321486813

12. https://en.wikipedia.org/wiki/Extended_Backus–Naur_form

13. Parr, Terence (May 17, 2007), The Definitive Antlr Reference: Building Domain-Specific Languages (1st ed.), Pragmatic Bookshelf ISBN 978-0-9787392-5-6

14. https://github.com/apache/cassandra/tree/trunk/src/antlr

15. https://github.com/antlr/grammars-v4/tree/master/sql/sqlite

16. https://github.com/gabriele-tomassetti/antlr-mega-tutorial

17. https://it.wikipedia.org/wiki/Programmazione_procedurale

18. C. J. Date with Hugh Darwen: A Guide to the SQL standard: a users guide to the standard database language SQL, 4th ed., Addison Wesley, USA 1997, ISBN 978-0-201-96426-4

19. R. Ramakrishnan; J. Gehrke: Database Management Systems, 3rd ed. (2003), Mc Graw Hill, " Overview of query evaluation", p. 393–417, ISBN 978-0-072-46563-1

20. R. Ramakrishnan; J. Gehrke: Database Management Systems, 3rd ed. (2003), Mc Graw Hill, " Tree-structured indexing", p. 338–364, ISBN 978-0-072-46563-1

21. https://en.wikipedia.org/wiki/Master_theorem_(analysis_of_algorithms)

22. https://github.com/gatesporter8/DBMS-SimpleDB-

23. https://en.wikipedia.org/wiki/Syntax_diagram