University of Turin
Department of Computer Science

Bachelor of Science in Computer Science

Design and implementation of a relational DBMS

Speaker:
Prof. Roger G. Pensa

Candidate:
Leonardo Magliolo

Academic Year 2020/2021

I declare that I am responsible for the content of the paper I am submitting for the purpose of obtaining the degree, that I have not plagiarized all or part of the work produced by others, and that I have cited the original sources in a manner congruent with current plagiarism and copyright regulations. I am also aware that should my statement be found to be false, I may incur the penalties provided by law and my admission to the final examination may be denied.

Acknowledgements:

I would like to thank my parents for the crucial moral and financial support offered to me throughout my graduation.
Special thanks go to my mother who, despite numerous health problems, always showed herself available so that I could pursue my studies in peace, as far as possible.

I would also like to thank my friends and classmates for their patience and support during writing of this thesis.
The contribution made by Athena, Cristian, Fabio, Greta and Marco to the achievement of the title was particularly valuable.

Finally, I would like to thank Professor Pensa for allowing me the opportunity to work on the proposed graduation project and for the helpfulness and professionalism with which he conducted the role of lecturer.

Index

Chapter 1: Introduction

The objective of this thesis is to create a relational, portable, small-scale Database Management System, the operation of which can be examined for educational purposes, enhancing the understanding of how modern RDBMSs are designed in practice.

In addition, the implementation of the project aims to consolidate and expand the knowledge learned during the undergraduate course of formal languages, translators and databases.

In the chapter on formal languages and translators, mechanisms capable of specifying and interpreting SQL language syntax will be explored.

In the chapter on databases, a brief introduction to the relational model, the specification of the SQL language and the methodologies that can be implemented to data representation and query execution by a DBMS will be carried out.

The project chapter will provide additional details with respect to the implementation choices , taking advantage of the terminology and observations proposed in the previous chapters.

Chapter 2: Formal languages and translators

2.1 Languages formal:

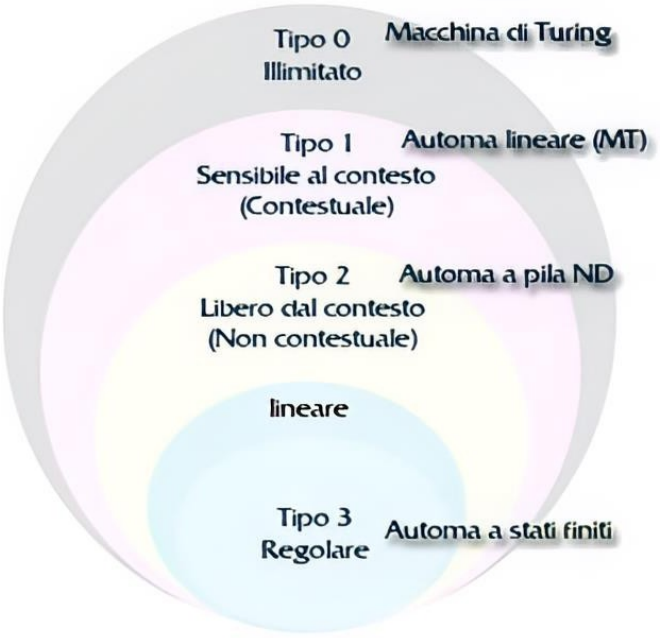
In the study of computer science and linguistics, a formal language [1] consists of a set of words defined on an alphabet that can comply with a specific set of rules.

Specifically, an alphabet (in jargon recognized asΣ) consists of a set of symbols, letters or tokens.
A peculiar set within the study of formal languages is the set of all possible strings that can be generated by means of the concatenation operation within an alphabet (in the literature denoted by).Σ^*

Considering the previous definitions, it is possible to observe that, in fact, for each Σ and for every formal language L defined onΣ the relation is always true: $L \subseteq \Sigma^*$.

With a view to being able to determine what characteristics a computational model should have in order to recognize a given subset of formal languages, Chomsky's hierarchy [2] was introduced into the field of formal languages in 1956.
The computational models considered by the hierarchy are respectively:

- Finite state automaton
- Non-deterministic stack automaton
- Linear automation
- Turing machine



As for the drafting of the project, regular languages (type 3) were used to correctly identify the tokens imposed by the SQL grammar by exploiting a lexical analyzer; instead, a parser capable of recognizing the SQL syntactic structure was implemented, exploiting the category of formal type 2 languages.

For these reasons, the following paragraphs will focus on a more precise definition of regular language and context-free language.

2.1.1 Languages regular:

A regular language **R** is a formal language for which a finite-state automaton *A* can be defined such that the set **L(**A) of strings accepted by *A* meets the following condition: ( )$\mathbf{L}A \subseteq \mathbf{R} \wedge \mathbf{R} \subseteq \mathbf{L}$ (*A*).

A deterministic finite-state automaton (DFA) can be defined by the quintuple:< **Q**, **Σ**, δ,q_0,**F** > whose elements are:

- A nonempty set of states **Q.**
- A finite set of symbols .**Σ**
- A transition function δ: **Σ**x**Q** →**Q** .
- An initial state$q_0 \in$ **Q.**
- A set of final states **F**⊆**Q** .

It is possible to see that the transition functionδ is sufficient to establish for each $s \in \mathbf{\Sigma}$ if it is true that $s \in$**L (DFA**).

To effectively extend transition function so that it can consider each $w \in \mathbf{\Sigma}^*$, it is necessary to define an extended transition function $\hat{\delta}$:**Σ*** x**Q** → **Q** such that it recursively exploits the function .δ

This can be done bearing in mind that the following condition is always true for every $w \in \mathbf{\Sigma}^*$: $w=w_1w_2..w_n$, with $w_i \in \mathbf{\Sigma}$for each i (the juxtaposition in the case of the sequence of w_i identifies the concatenation operation). Therefore,

the function$\hat{\delta}$ can be defined as follows:

- Base step:$\hat{\delta}(q,)\varepsilon=q$($\varepsilon$ in the literature refers to the empty string).

- Inductive step: $\hat{\delta}(q,)w=\hat{\delta}(q,w_1w_2...w_i)= \delta(\hat{\delta}(q,w_1w_2...w_{(i-1)}),w_i)$.

The definition of $\hat{\delta}$ allows a rigorous description of the language accepted by the former DFA: $(\mathbf{LDFA}) = \{ w \in \Sigma^* \mid \hat{\delta}(q_0,) \} w \in \mathbf{F}$

Although regular languages are recognizable by DFA, it is possible to formulate a structural [3] demonstration on the generic DFA pair *A* and *B* that proves that for regular languages **L(*A*)**, **L(*B*)**:

- **L(*A*)** U **L(*B*)** is regular.
- **L(*A*)L(*B*)** is regular (the juxtaposition between L*(A)* and L(*B*) identifies the concatenation operation).
- **L(*A*)*** and **L(*B*)*** are regular languages (* refers to Kleene's closure).

Thus, it becomes possible define regular languages from elements belonging to Σ by means of concise expressions exploiting the set operators listed above; thus avoiding the much more arduous task of having to explicitly provide
the transition function, but retaining the ability to be able to compute the language accepted by the expression by algorithmically generating a DFA that accepts a language equivalent to then calculate the results its extended transition function $\hat{\delta}$.
The expressions mentioned above are named regular expressions [4], which are used within this project for the definition of SQL tokens.

2.1.2 Languages free from context:

A context-free language L is a formal language for which a context-free grammar (CFG*) G* can be defined such that the set **L(*G*)** of strings accepted by *G* meets the following condition: $( )\mathbf{L}G \subseteq \mathbf{L} \wedge \mathbf{L} \subseteq \mathbf{L}(G)$.

A CFG *G* can be defined as a quadruple: **< V, T, P,** *S*> whose elements are:

- A set of variables .**V**
- A set of terminals .**T**
- A set of productions **P** , a production is defined as: $v \rightarrow c$ with $v \in \mathbf{V}$ and $c \in (\mathbf{V} \cup \mathbf{T})^*$.

In particular,v is often referred to as the "head" of production and$c$ as the "body."

- An initial symbol $S \in \mathbf{V}$.

For the purpose of formalizing which language is accepted by a CFG, it is useful to introduce the derivation operator $:\Rightarrow_G \alpha A \beta \Rightarrow_G \alpha \gamma \beta \leftrightarrow (A \rightarrow \gamma) \in \mathbf{P}$ with $A \in \mathbf{V}$ e $\alpha, \beta, \gamma \in (\mathbf{V} \cup)\mathbf{T}^*$

Similarly, because of the way the transition functionδ has been extended toδ^*, we want to define a new operator$\Rightarrow^*_G$ on recursive bases that identifies multiple derivation sequences:

- Basic step:$\forall_{\alpha \in (\mathbf{V} (\cup) (\mathbf{T})) :^*} \alpha \Rightarrow^*_G \alpha$.
- $\beta \Rightarrow_G$Inductive step:$\forall_{(\alpha,\beta,\gamma \in (\mathbf{V}) (\cup) (\mathbf{T})) :^*} \alpha \Rightarrow^*_G \wedge \beta \gamma allora \alpha \Rightarrow^*_G \gamma$.

At this point it is possible to specify precisely what language is accepted by the grammar:

$$\mathbf{L}( )G = \{ w \in \mathbf{T}^* | S \Rightarrow^*_G w \}.$$

It can be seen that one can recursively construct a tree whose nodes belong $(\mathbf{V} \cup)\mathbf{T}^*$ such that it identifies a sequence of productions applied on the grammar G:

- Basic step:

  $\forall_{\gamma \in \mathbf{T}^*} \gamma \Rightarrow^*_G \gamma$ is associated with a tree composed of a single leaf containing γ
- Inductive step:

  $\forall_{(\alpha,\beta)_1 (,\ldots,\beta)_n (\in (\mathbf{V}) (\cup) \mathbf{T}) ^*} \alpha \Rightarrow^*_G \beta_1 \ldots \beta_n$is associated with a tree whose root represents a node containingα having exactly n children.

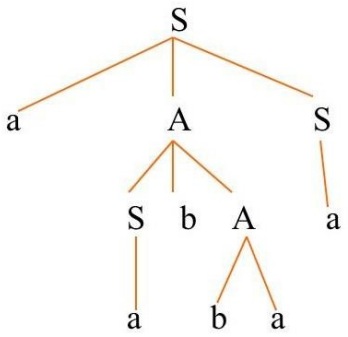
  Each i-th child is also a tree made from $\beta_i \Rightarrow^*_G \omega_1 \ldots \omega_m$.

In particular, if $w \in \mathbf{L}(G)$ it is possible to generate a syntactic tree by means of the above procedure such that the concatenation from left to right of all its n leaves $w_i \in \mathbf{T}^*$is equivalent to$w$, or in other words: $w = w_1 \ldots w_n$

The tree thus generated is called a "syntactic tree" in the literature, the construction of which can be exploited by a program (parser) to determine whether $S \Rightarrow^*_G w$ and thus determine whether
$w \in \mathbf{L}(G)$.

An example a syntactic tree is given below:

G = ({S,A} ,{a, b}, P, S)
Con P:
S → a A S
A → S b A
A → S S
S → a
A → b a



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

S → aAS → aSbAS → aabAS → aabbaS → aabbaa

For some grammars it is possible to prove the existence of at least one string w in the accepted language for which there are different syntactic trees whose leaves concatenated from left to right are still equivalent to w.

Grammars possessing the property listed above are classified as "ambiguous" [5] and in many cases represent an obstacle in the definition of a parser if the syntactic tree is to be constructed observing a certain order of precedence with respect to the productions to be applied.

2.2 Translators:

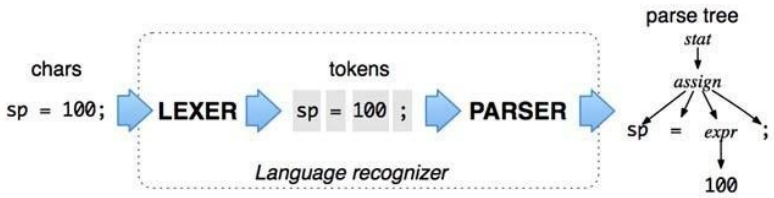
Translator, in computing, is a term referring to programs that can convert code written in a specific programming language into another language [6].

Since programming languages are also formal languages, it is possible in most cases to apply the methodologies outlined in the previous paragraphs for their lexical and syntactical definition.

It is usually possible to decompose a translator into subprograms that can talk to other, so that the complexity of the code can be effectively managed and its tasks can be broken down for greater understanding of the logic of operation.

The main subprograms implemented during the writing of a translator are a lexical analyzer (in jargon called a lexer) and a syntactic analyzer (called a parser).

An intuitive example illustrating the division of tasks between lexers and parsers is shown below (taken from the source [7]):



2.2.1 Analysis lexical:

Lexical analysis is the process that can convert a sequence of characters into a sequence of tokens; program that can perform lexical analysis is called a lexer or tokenizer.

A token is a string for which a role is assigned within the formal language; More precisely, it can be defined as a pair of strings: <Token Name, Token Value>.

The most common types tokens within programming languages are:

- Identifier: Name associated with a variable chosen by the programmer.
- Keywords: Name referring to typical language constructs.
- Operator: Symbol capable of taking certain types of data as input and returning a result.
- Literal: Represents constants for a particular data type.
- Comment: Arbitrary character sequence not interpreted by the translator.

Characters identified as separators (usually space and carriage return) are considered of particular importance, as they are used by the lexical analyzer to determine the end of the sequence of characters with which to associate a token.

Most programming languages have tokens capable of being associated with character sequences belonging to the regular language family, so token recognition is often delegated to the use of regular expressions.

2.2.2 Analysis syntactic:

Syntactic analysis is the process that can determine how a terminal string can be generated from a grammar; program capable of performing syntactic analysis is called a parser. Typically, the syntactic analysis performed by a parser takes advantage of the generation of a syntactic tree.

Depending on the type parsing used to construct the syntactic tree mentioned above, parsers can be classified as LR or LL:

- Top-down analysis: The construction of the syntactic tree begins by placing the initial symbol of the formal grammar under consideration as the root, progressively determining smaller and smaller portions of the tree, associated with the descendants of the root node.
  LL parsers perform top-down parsing.
- Bottom-up analysis: The construction of the syntactic tree begins with the generation of leaves associated with the tokens of the considered string, progressively determining larger and larger portions of the tree, associated with the ancestors of the leaf nodes.
  LR parsers perform bottom-up analysis.

The parser used for SQL syntax recognition implemented for this project is of the LL type. Therefore, the remainder of this section will focus in more detail on the methodologies that can be adopted for the construction of the syntactic tree using top-down parsing.

An LL-type parser is able to generate the syntactic tree for a particular sequence tokens by exploiting a methodology similar to that used by the Depth-First Search algorithm [8], which uses a stack as a queue to explore the nodes in the tree, depending on their depth.

Since today's programming languages have function calls, implemented by exploiting a stack as a data structure, it is possible to make an LL-type parser by associating a function with each variable belonging to the grammar.
Each function will make specific calls to other functions in relation to the output to be applied, having symbol in the head equal to that of the variable associated with function.

It is of considerable importance in writing a parser to be able to determine, having chosen a given grammar variable, which production to apply by means of the next derivation. Grammars that allow predictive descending parsing are called LL(k), where k is the number of symbols needed to identify the next production to be applied unambiguously.

The set **LL(k)** contains all and only formal languages capable of being defined by means of a grammar LL(k), for a value of k >= 1.

A parser is classified as LL(*) if it can parse a generic LL(K) grammar, for any $k \in \mathbb{N}$.

For simplicity, we proceed to illustrate the criterion by which an LL parser is able to determine which production to apply unambiguously given an LL(1) grammar.

Given a CFG $G <$ **V, T, P,** $S >$, the set **FIRST(α)** containing all $a \in$ **T** such that $\alpha \Rightarrow^*_G a$ is recursively computable as follows:

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

The set **FOLLOW(A)** with A ∈**V**, containing all $a \in$ **T**with which strings begin following A in the derivations of the grammar *G* (defining$ as the final character of the string), can be calculated as follows:

- $V = S \rightarrow$ **FOLLOWSTART(**V**)=** $\{\$\}$
- $V \neq S \rightarrow$ **FOLLOWSTART(**V**)** $=\varnothing$

- **FOLLOW(** $)A =$ $\{\bigcup_{B \rightarrow \alpha A \beta \in P}$ **(** $)$**FIRST**$\beta - \{\varepsilon\}\} \cup \{\bigcup_{\substack{B \rightarrow \alpha A \beta \in P \\ \wedge \beta \Rightarrow^*_G S \\ \wedge B \neq A}}$ **FOLLOW (**B **)**$\}$

$$\cup$$ **(** $)$**FOLLOWSTART**A

Finally, considering G, the set **PREDICT($A \rightarrow \alpha$)** containing all $a \in \mathbf{T}$ with which the strings that can be generated by $A \rightarrow \alpha$ begin is computable as follows:

- $\neg(\alpha \Rightarrow^*_G \varepsilon) \rightarrow$ **PREDICT($A \rightarrow \alpha$) = FIRST(α)**
- $\alpha \Rightarrow^*_G \varepsilon \rightarrow$ **PREDICT($A \rightarrow \alpha$) = (FIRST(α) − {ε}) ∪ FOLLOW(A)**

If $\mathbf{L(G) \in LL(1)}$ then for each $A \in \mathbf{V}$: $(\bigcap_{A \rightarrow \alpha \in P} \mathbf{PREDICT}A \rightarrow \alpha) = \emptyset$

Hence, it is sufficient for an LL(1) language recognizer parser to determine the next token to be consumed 'next_token' to calculate, depending on the variable A considered, what the next production to be applied with A in the lead: $A \rightarrow \alpha$ is applicable if and only if next_token $\in$ **PREDICT($A \rightarrow \alpha$)**.

2.2.3 Analysis semantics:

The semantic parser uses the syntactic tree, constructed by the parser, to ensure that the source program respects the semantic constraints defined within the language under consideration. In addition, it gathers information about the data types of variables by storing them, typically, within a symbol table, inspected later for generating intermediate code , or, to implement generic semantic actions at subsets of the language syntax.
A particularly important example of semantic analysis 'type checking' [9], analysis that can determine for each operator whether it has been given operands having correct data type.

The most common approach to designing a semantic analyzer is to define:

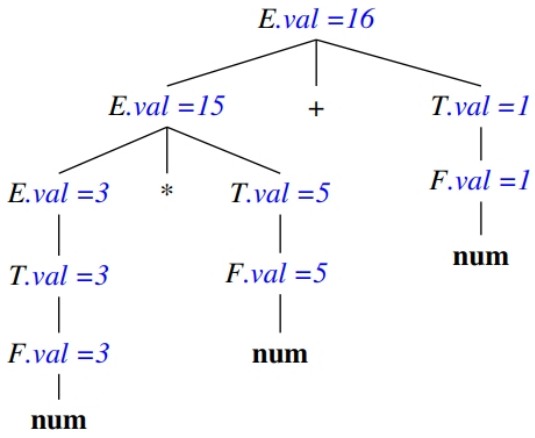
- A set of attributes (definable as a triple <Name, Data Type, Value>) associated with variables belonging to the formal grammar under consideration.
- A Syntax-Directed Definition (SDD for short), used to make explicit the evaluation rules to be performed on the attributes of the grammar in relation to the applied output.

Translation is the result of the evaluation of semantic rules established means of an SDD scheme.

In the literature, a syntactic tree that reports the values of attributes associated with nodes is called an annotated syntactic tree.

An example of SDD with attached annotated syntactic tree is given below (taken from the source [10]):

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



(Left: Example of SDD associated with the productions of a grammar)

(Right: Example of annotated syntactic tree, generated from the grammar given in the example on the left, for the input: 3 * 5 + 1)

To access the value of an attribute 'a' associated with a variable of the 'V' grammar, the notation: 'V.a' was used.
Note, too, how attributes associated with terminal variables have a value provided directly by the lexical analyzer; in SDD schemes it is not possible to make explicit semantic rules for calculating the values of attributes associated with terminal variables.

Depending on how the attribute 'a' belonging to the variable 'V' is computable from a given node of the syntactic tree n, it is classified as:

- Synthesized: If all the attributes needed to calculate 'a' are directly associated with node n or its child nodes.
  In this case, the variable 'V' is present in the head of the considered production.

- Inherited: If all the attributes needed to calculate 'a' are directly associated node n, its parent node or its sibling nodes.
  In this case, the variable 'V' is present in the body of the considered production.

Depending on the types of attributes declared within an SSD schema, the schema can be classified as:

- S-attributed: If it uses only synthesized attributes
- L-attribute, if for each attribute 'a' belonging to the variable 'V' one of the following occurs. conditions:
    - 'a' is summarized
    - If V.a is an inherited attribute computed by a semantic rule associated with production $A \rightarrow \omega_1 \ldots \omega_n V \omega_{n+2} \ldots \omega_m$, then the rule can only use :
        - inherited attributes of A.
        - attributes of variables $\omega_1 \ldots \omega_n$.
        - attributes of V as long as the introduction of them still makes it possible for a finite evaluation sequence to exist.
        - attributes of ω_i , with i between [n+2, m], provided that ω_i is terminal.

Although SDD schemes specify which semantic actions to perform in relation to the applied output, they do not necessarily specify an allowable order with which to calculate the value of attributes.
For this reason, Syntax-Directed Translation schemes (SDT for short) are introduced, which can specify translation operations to be performed during parsing.

A translation scheme is an SDD, in which semantic actions, enclosed in curly brackets, are placed in the body of the productions, in such a position that the values of the attributes indicated within the semantic actions have already been calculated.

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

(Example SDT scheme, formulated on the grammar of SDD illustrated above)

Depending on the type of SDD, it is possible to algorithmically generate an SDT scheme that can satisfy the availability constraints imposed on the attributes within the semantic actions:

- Generation an SDT scheme from an S-attributed SDD:

  Actions, which calculate the synthesized attributes of the variables at the head of the productions, are entered at the end of the productions.
- Generation of an SDT scheme from an L-attributed SDD:
  - Actions that calculate the synthesized attributes of the variables at the head of the productions are entered at the end of the productions.
  - Actions that compute inherited attribute values for a nonterminal V immediately prior to its occurrence inserted into the body of affected productions.
  - If several attributes inherited from V depended on each other, evaluations of the affected attributes would be ordered so that those first needed would be calculated.

For further discussion regarding SDDs and SDT schemes, the source [11] is recommended.

2.3 ANTLR:

ANTLR (ANother Tool for Language Recognition), is an automatic LL(*) parser generator for arbitrary grammars of type LL(k).
Typically used for building interpreters and translators for domain-specific languages (DSLs), in its version 4 it can support numerous languages for writing lexers and parsers (including: Java, C#, C++, and Python).

By means of the EBNF (extended Backus-Naur form) meta-syntax [12] it is possible to declare, in a straightforward manner, translation schemes (SDTs) that are later used by ANTLR to automatically generate the corresponding translators.

2.3.1 Why ANTLR?:

There are considerable benefits to using an automatic parser generator if you intend to implement a translator:

- Greater decoupling between production rule definitions and related semantic actions; by means of the meta-syntax chosen by the generator, used as an intermediate representation language, recognizers written in different source languages can be generated without having to manually code the same parser multiple times in different programming languages.
- Significantly reduced development timelines.
- Automatic reporting of errors due to lexical analysis and syntactic analysis processes.
- IDE support plugins that can facilitate debugging.

Although LR parsers are able to recognize a wider family of formal languages not recognizable by LL parsers, the technology on which LR parsers are based is less intuitive than that of LL recursive descendant parsers, making some of the operations required for debugging difficult.

Finally, ANTLR enjoys comprehensive documentation [13] and a particularly active developer .
Source files made available from the official Apache and ANTLR repositories were helpful in the development of the project:

- Apache Cassandra (DBMS written in java with ANTLR support) [14]
- An example SQL grammar written in ANTLR [15].

2.3.2 Definition of a lexer:

A code fragment representing an example of a Lexer defined in ANTLR is given below (taken from the source [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                : ~[\])]+ ;
```

The example shows how character sequences can be defined for token definition, using an intuitive syntax that combines elements of EBNF notation along with regular expression definition.

The keyword 'fragment' identifies reusable rules within the token definition, used in this context to make declared keywords case-insensitive.

2.3.3 Definition of a translator:

The following is a code snippet containing a representative portion of the translator implemented for this project; the example was chosen so that it shows almost all of the constructs made available by ANTLR for defining SDT schemes:

```
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);
    }
    n2 = join_1[$n2.nodeToReturn]{$nodeToReturn = $n2.nodeToReturn;}
    |{$nodeToReturn = n;};
```

In addition to the EBNF notation, it is possible to state:

- Semantic actions, enclosed in curly brackets.
- Return values associated with grammar variables, by means of the syntax 'nameVariable returns [typeDiDate nameValue]'.
- Parameters associated with grammar variables, by means of the syntax 'nameVariable [typeOfDataNameParameter]'.
  Parameter declaration allows to implement inherited attributes, and thus to declare SDT patterns derived from L-Attributed SDDs.
- Aliases associated with grammar variables called in productions.

Each variable invoked within a production possesses certain attributes, some declared within the grammar (this is the case with parameters or return values) and some declared implicitly for each variable by ANTLR itself (a particularly important attribute is 'text', the contents of which represent the sequence of characters associated with the production applied on the called variable).
To access the attributes of a variable within a semantic action it is necessary to use the syntax: '$AliasVariable.nameAttribute'.

For further discussion regarding the definition of translators through ANTLR, the source [13] is recommended.

Chapter 3: Bases of data

3.1 Model relational:

The relational model was introduced as a logical model for data representation and structuring by Edgar Codd in 1970, allowing greater independence data and simplification of queries to databases.
The relational model primarily exploits the concepts of set relation and table.

In particular, for a relationship **R** between n sets (in jargon called n-air): $\mathbf{I_1}, \mathbf{I_2}, \ldots \mathbf{I_n}$ it is always true that $\mathbf{R} \subseteq \mathbf{I_1} \times \mathbf{I_2} \times \ldots \times \mathbf{I_n}$, therefore **R** is a set whose elements are definable as ordered collections $(i_1, i_2, \ldots, i_n)$ where $\forall_{j \in [1,n]} (i_{(j)}) \in \mathbf{I_j}$
The elements contained in **R** are named tuples.

Although the concept of relationship is rather simple to define, in the context of databases it presents some problems.
As noted earlier, the tuples contained within a relation are defined by means of positional rather than symbolic notation, which is particularly inconvenient to use by means of queries. Therefore, it would be preferable to consider tuples whose values are identified by attributes.

We then proceed to provide an alternative formal definition that can solve the problem:

- A set of **D** domains is defined.
- A set attributes **A** is defined.
- A new *dom* function is defined, such that: $\mathbf{A} \rightarrow \mathbf{D}$.
- The concept of a tuple is reformulated by means of a function *t*, such that:
  $\mathbf{A} \rightarrow dom(A)$.
- The concept of relationship is reformulated so that a relationship **R** defined on **I** is definable as a collection of tuples: $\{t \mid t(\mathbf{A})\}$.

To access the values of attributes **A** belonging to a tuple *t* the following notation will be used: $t[\mathbf{A}]$.

Given the new relationship reformulation, it can be conveniently represented by means of a table, having indexes for the columns belonging to the set of

relationship attributes and rows containing tuples, represented with the attributes in order with respect to the order chosen for column representation.

The tabular representation of a relationship, while convenient, necessarily forces one to choose an order of presentation of attributes, not proper to the relationship as defined above. Therefore, it is necessary to keep in mind that for a given relationship there can be n! different tables representing the same one, with n being equivalent to the cardinality of its set of attributes.

3.1.1 Constraints of integrity:

The integrity constraint is a property capable being satisfied or not, applicable to instances of the database to examine its consistency.
Based on the type of instances involved, integrity constraints can be classified into two categories:

- Intra-relational constraints: Having chosen a relation, it is possible to determine whether its constituent elements meet relational constraints by examining tuples as instances (in the case of tuple constraints) or by examining the values of attributes contained in some tuples (domain constraints).
- Inter-relational constraints: Conditions are checked by considering tuples of different relationships (referential integrity constraints).

For simplicity, only the integrity constraints, knowledge of which is critical to understanding the implemented DBMS, will be examined in detail.

In order for the relational model to be used in practice, it is necessary to be able to uniquely identify a particular instance belonging to a relationship.
Recalling that a relation **R** is nothing but a set of tuples *t*, defined over a set attributes **A**, since the contents of a set do not admit repetitions of the same elements, there cannot exist a pair of tuples $t_1, t_2 \in$ R such that $t_1[\mathbf{A}] = t_2[\mathbf{A}]$ with $t_1 \neq t_2$.

A superkey **R** is defined as a set of **SK** attributes such that:

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

Intuitively, a superkey is sufficient to identify any instance of the relation on which it is defined.
Note how, given the above considerations, for every **R**: A is superkey of **R.**

An intra-relational key constraint dictates that chosen a relation **R** and chosen a set of attributes**I** ⊆ **A** then:**I** is superkey of **R.**
Should a given tuple make key constraint false, it could not be inserted
Within the **R** report.

Another noteworthy constraint is called referential integrity; it is used to reference the tuple of a particular relation within a tuple of a different relation.
The referential integrity constraint defined on the tuples of a relation$\mathbf{R}_1$ toward the tuples of a relation$\mathbf{R}_2$ dictates that chosen an **I**, given a key constraint defined on$\mathbf{R}_2$ having superkey :**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 Advantages and disadvantages:

Compared with the main alternative logical models of data representation (hierarchical, lattice, and object-oriented), the relational model offers a sharper separation between the data presentation (logical) layer and the data representation (physical) layer, which is often defined by means of pointers.
This allows, from the user's point of view, a greater focus on the nature of the data regardless of the underlying physical architecture, greatly simplifying the definition and querying of databases.

In any , the use of the relational model is not always justified regardless of the volume and characteristics that the database will be able to possess.
As the number of tables increases, in fact, many queries exhibit degradations in response time.
In addition, the requirement to have to define the database by means of relationships is not particularly efficient in terms of performance for some applications (the definition of hierarchies for OLAP aggregations is an example).

3.1.3 Algebra relational:

In order for a database to be queried, a language must be defined such that it considers a given subset of relationships contained in the database, and returns a relationship containing the tuples that meet the query criteria.

Relational algebra turns out to be particularly important as far as query planning by the DBMS is concerned, given its characteristic of being a procedural language [17].

A list of operators belonging relational algebra is defined below, so that in the following paragraphs we can refer to it to explain some parts of the operation of the DBMS:

- Cartesian product (or cross join): It is possible to extend the Cartesian product to relations by means of the juxtaposition operation applicable to tuples:

  $$\mathbf{R}_1 \mathbf{x} \mathbf{R}_2 = \{t_1 t_2 \mid t_1 \in \mathbf{R}_1 \wedge t_2 \in \mathbf{R}_2\}.$$

  By doing so, the new set of attributes on which the tuples are defined becomes $\mathbf{A}_1 \cup \mathbf{A}_2$
- Union, intersection and difference: It is possible to extend these typical set operators to relations such as: $\mathbf{R}_1, \mathbf{R}_2 \rightarrow \mathbf{R}$, as long as $\mathbf{A}_1 = \mathbf{A}_2$.
- Renaming: Because of the limitations imposed in the previous point, it is sometimes necessary to change the names of the attributes of a relationship; the renaming operator can be used to do this, intuitively:

  $$\rho_{newattr1, newattr2, \ldots, newattrn \Leftarrow attr1, attr2, \ldots, attrn}(\mathbf{R}).$$
- Selection: Selection is used to consider only a subset of tuples belonging to a relation respecting a Boolean condition, generically composed of order operators bound by means of logical connectives:

  $$\sigma_{booleanExp}(\mathbf{R}).$$

  Note how $|\sigma_{booleanExp}(\mathbf{R})| \leq |\mathbf{R}|$ *siccome* $\sigma_{booleanExp}(\mathbf{R}) \subseteq \mathbf{R}$.
- Projection: Projection is used to consider only a subset attributes belonging to a given relationship:

  $$\pi_{attr1, attr2, \ldots, attrn}(\mathbf{R}), \text{ with } \{attr1, attr2, \ldots, attrn\} \subseteq \mathbf{A}.$$
- Natural Join: It is used to correlate the contents of multiple relationships by considering the Cartesian product of tuples having equal value for attributes in common:

  $$\mathbf{R}_1 \bowtie \mathbf{R}_2 = \{t_1[\mathbf{A}_1] t_2[\mathbf{A}_2 - \mathbf{A}_1] \mid t_1 \in \mathbf{R}_1 \wedge t_2 \in \mathbf{R}_2 \wedge t_1[\mathbf{A}_1 \cap \mathbf{A}_2] = t_2[\mathbf{A}_1 \cap \mathbf{A}_2]\}.$$

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

Since relational algebra belongs to the family of context-free languages, it follows that it is possible to represent a given expression by means of its syntactic tree.

By means of certain properties associated with the consecutive application of operators, it is possible to generate syntactic trees representing expressions that are syntactically different but equivalent in terms of the relation returned.

Some basic properties used by relational DBMSs to generate alternative syntactic trees for optimization purposes are listed:

- $\pi_{attr1,attr2,...,attrn}(\mathbf{R}) = \pi_{attr1,attr2,...,attrn}(\pi_{atr1,attr2,...,attrn,...,atrm}(\mathbf{R}))$.
- $\sigma_{booleanExp1 \wedge booleanExp2}(\mathbf{R}) = \sigma_{booleanExp1}(\sigma_{booleanExp2}(\mathbf{R}))$.
- *proiezioni e selezioni godono in fine della proprietà distributiva*.

For more information logical optimization applicable to queries we recommend consulting the source [19].

3.2 Language SQL:

Structured Query Language is a standardized language for the interaction of relational databases, designed to be able to allow the following operations:

- Create and modify database schemas, by means of operations classified as of Data Definition Language (abbreviated DDL).
- Entering and modifying data entered within schemas, by means operations classified as of Data Manipulation Language (Abbreviated DML).
- Query the database to extract information, by means of operations classified as of Data Query Language (abbreviated DQL).
- Create and manage data control and access tools, by means of operations classified as of Data Control Language (abbreviated DCL).

Since SQL language has characteristics typical of both declarative and procedural languages, it proves to be particularly versatile according to usage requirements, so much so that it

have become the de facto standard language for querying any relational database.

Since 1974, SQL has been upgraded to different versions, gradually expanding the acceptable syntax to perform increasingly complex operations.
For the sake of convenience, this thesis will discuss in detail only a subset of constructs implemented in SQL-92, which brings from the first version (SQL-86) the introduction of referential integrity constraints and the explicit join statement.

For more information regarding SQL, the source [18] is recommended.

3.2.1 DDL syntax main:

The main SQL commands of DDL type are 'Create' and 'Drop', can respectively create and drop a relation belonging to the chosen database.

Each relationship within the database is identified by its name, so it must be mentioned in both 'Create' and 'Drop'.
Furthermore, as defined of the previous paragraphs, a relation is a collection of tuples defined over a set of attributes having an associated name and domain.

An intuitive example of syntax for the 'Create' command is given below:

```
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),
....
);
```

Note how FOREIGN KEY allows the definition of inter-relational integrity constraints such as that of referential integrity.

The drop command, when used to drop a relationship within the database, takes on the following syntax: DROP TABLE name_TABLE.

3.2.2 DML syntax main:

The main SQL commands of type DML are 'Insert' and 'Delete', can add and remove tuples within a specific relation, respectively.

By means of the following syntax it is possible to specify a set of tuples to be added to a relation:

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

Instead, it is possible to delete previously inserted tuples within a relation as follows: DELETE FROM table name WHERE booleanExp.

3.2.3 DQL syntax main:

The main SQL command of DQL type is 'Select', which can query the database:

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

Only the Select and From clauses are needed at the query statement.

In relation to the above code fragment, the operation of the 'select' command is described as follows:

- Attribute list: It is used to declare the attributes of the tuples belonging to the resulting relation and, simultaneously, establish an order of the attributes for generating the table associated with the relation.
  You can use the '*' character to preserve all attributes belonging the tuples in the relation.

- List of functions: It is used to explicate functions (sometimes aggregation functions) applied to the resulting relation generated by means of the query.
  The most commonly used functions are: MIN, MAX, AVG, SUM and COUNT.
- List of tables: This is used to declare sequences of joins of various types. In SQL92 the most commonly used join types are cross joins, declared by means of the syntax: table_1, table_2, ..., table_n.
  You can also declare different types of joins by means of the syntax: table_1 JOIN table_2 ON (booleanExp_1) ... JOIN ON (booleanExp_n).
  Doing so will execute sequences of Cartesian products whose resulting tuples will satisfy the expression: booleanExp_1 ... ∧ ∧ booleanExp_n. In addition, SQL allows the possibility of pointing to tables generated by means of other queries, thus implementing a subqueries mechanism.
- Condition: A condition is expressed such that all tuples given in the relation returned by the query must satisfy, similar to how carried out by the selection operator in relational algebra.
  Similar to as described for the FROM clause, there are subqueries features that can retrieve values from different tables than those mentioned in the FROM clause.
- Grouping Attributes: In case you intend to use aggregation functions (such as AVG, SUM and COUNT), the contents of the GROUP BY clause make explicit the attributes such that if a pair of tuples possesses those equal attributes, then they fall into the same aggregation category.
- Grouping conditions: Conditions expressed within the HAVING clause are evaluated after the aggregate attributes are calculated, allowing a wider range of selection not possible within the WHERE clause.
- Sorting Attributes: It is possible to declare an order among the attributes of the relation (of type ASC ascending or DESC descending) such that the tuples of the relation will be shown within the table.

3.3 RDBMS:

A Database Management System (DBMS) is a software system aimed at managing databases, typically providing the ability to perform DDL, DML, DQL and DCL operations on them, by means of a specific language.
In the case where the DBMS is concerned with managing relational databases, then we refer it as Relational DBMS (abbreviated RDBMS).

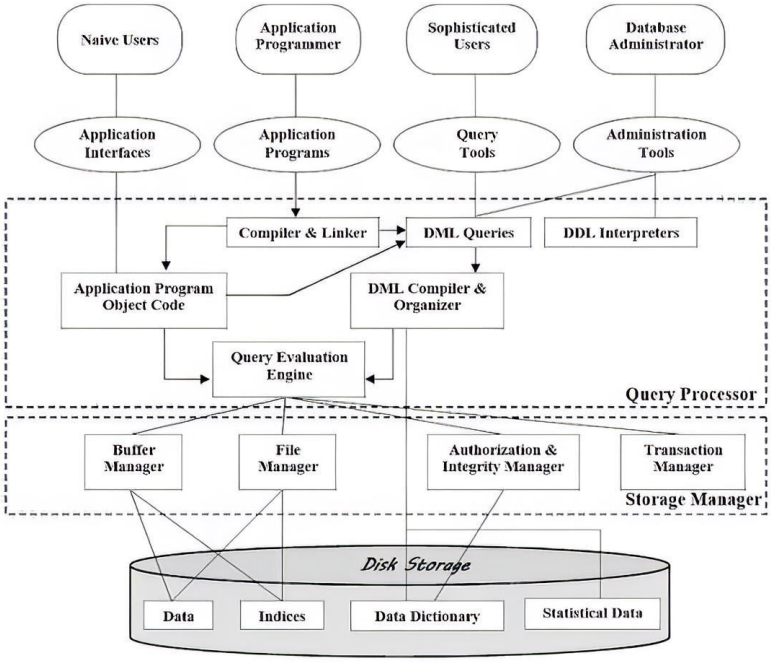
In order for a DBMS to be put to practical use, it must have a mechanism such that it allows the database to be queried by several programs simultaneously; the portion of the software charged with performing this task is called a driver (or connector).

One of the pivotal concepts to understanding how a DBMS works is the transaction.
A transaction identifies a request for interaction with the database by an application, which can be represented as a pairT_i : <ID_Identifier:i , query>.

A DBMS is concerned with managing transactions by ensuring, for each of them, the satisfaction of properties that in jargon are called ACIDs:

- Atomicity: For a DBMS, the indivisible unit of the processes it operates must be the transaction; therefore, the instructions associated with the transaction query must take effect on the database only if they are all executed successfully.
- Consistency: The database must respect integrity constraints defined on it, before and after any transaction is handled.
- Isolation: Each transaction must be executed independently other transactions; any failure of one transaction must not affect the execution of the others.
- Durability: Once a transaction has been performed whose associated query involves altering the database, then the changes made must be permanent.

Given the low dependence that exists among the ACID properties, it is possible to decompose the architecture of a DBMS into modules that can individually satisfy each of the properties; a typical example is given below:



The modules of greatest interest in order to ensure the fulfillment of ACID properties are:

- Transaction Manager: Receives transactions and manages their execution by communicating with other modules belonging to the Storage Manager category.
- Serializer: It is responsible for ensuring that the isolation property is satisfied.
- Recovery Manager: Takes care of ensuring atomicity and integrity properties (in case of an error by trying to recover the correct data or by forcing abort of the current transaction).
- Buffer manager: Responsible for ensuring the durability property.

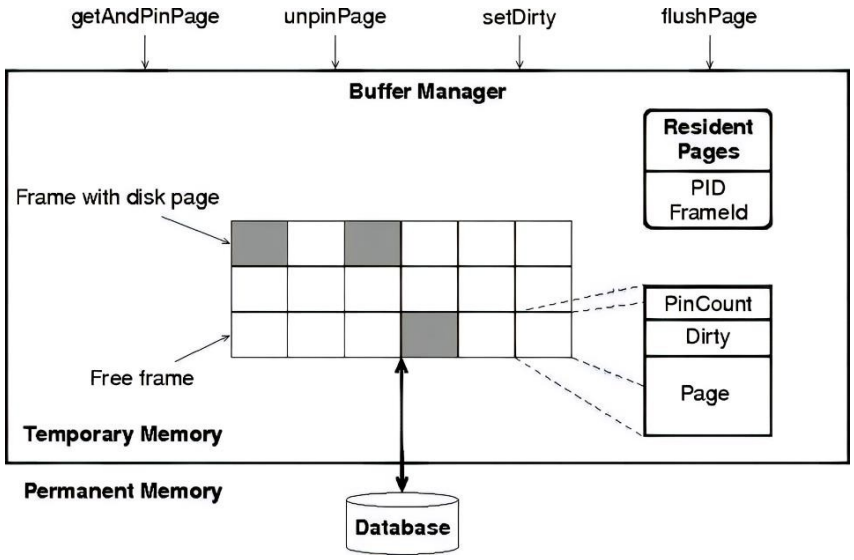
Although the relational model is used to formulate the high-level interactions between the database and the requesting application, an RDBMS necessarily must organize the representations of tuples and tables in such a way that they can be contained in a file system. Typically, all records necessary for the operation of an RDBMS are saved in pages, the size of which depends on the file system; usually their capacity between 8 and 64 KB.

In order for specific pages to be uploaded and downloaded from the file system, they must be assigned a unique identifier, in the literature called a Page Identifier (abbreviated PID).

The most important components belonging each page are:

- Record set: Containing the data associated with the page, usually represented by means of a stack
- Array of offsets: Containing a pointer to the address in memory for each element belonging to the record array
- Bit set dirty: is set to true if the page has been modified, whenever the page is downloaded from the central memory to the mass memory the bit is set to false.

The page is thus structured to grant access to any record by specification of a RID: <PID, offset: i> pair.



Bearing in mind that to access any tuple the DBMS is required to load the entire page containing it (from permanent memory to central memory by means of the de buffer manager), to minimize the execution time of a query it is necessary to move as few pages as possible.

The times required to transfer a page from permanent memory to central memory are, to date, in the range of 10^{-3} seconds; those to access the contents of

a page already within the central memory is in the range of $10^{(-9)}$) seconds.

The time cost of executing a query is almost entirely attributable to transfers between mass and central memory; the ratio of the factors is on the order of 10^{6}.

A very common example of a query is one in , having selected a certain relation, one wishes to derive a new relation containing a subset of tuples present in the original one:

select * from relationship where attr= n;

Since the tuples of a relation are contained within a set of m pages, and placing the probability of finding a given tuple within a page equiprobable to that finding it on all other pages:

- If attr is not key the search forces the consultation m pages.
- If attr is key:
  - The successful search, in the average case, requires the consultation of $\frac{1}{m}\sum_{i=1}^{m} i = \frac{1}{m} * \frac{m(m+1}{2} = \frac{m+1}{2}$ pages.
  - Searching with bankruptcy requires consulting m pages.

Sorting the records within pages improves the situation only marginally,
By reducing the pages consulted in each scenario considered above to $\frac{(m+1)}{2}$, the
page accesses still remain in the range of $O(m)$.

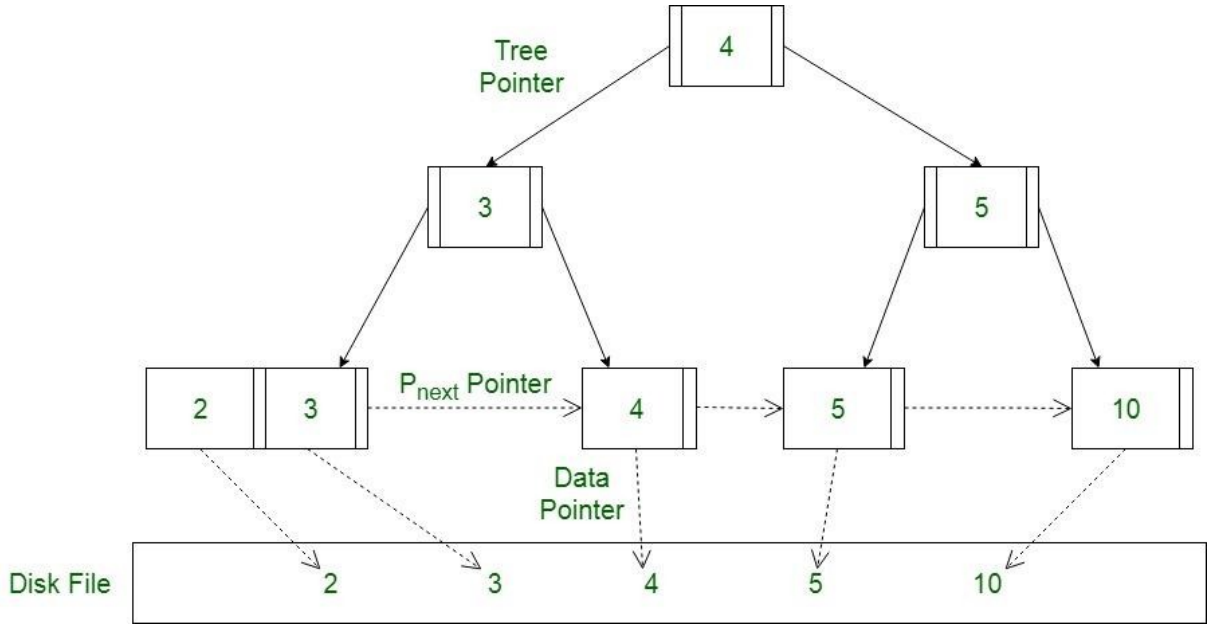
Given that in a sufficiently complex system the number of pages associated with a report may exceed even 10^6, it is necessary to establish a different strategy by defining a data structure according to which to represent the pages according to the order established on specific attributes belonging to the tuples in the relation, the name of which is defined as the index.

Widely used data structures for the realization of indexes are trees [20], in the following sections we will deal in particular with their definition by means of B+Trees.

A B+Tree with branching factor equivalent to m is a $(m-1)$-ary tree that has the following characteristics:

- Maximum number of keys per node: $m - 1$.
- Minimum number of keys per internal node: $\lfloor \frac{m}{2} - 1 \rceil$.
- Number of keys within the root node: between $[1, m-1]$.
- If a nonleaf node contains n keys then the node must have n+1 children.
- The tree must be balanced.

- The keys associated a node are represented in an orderly manner.
- Each leaf, having maximum value$x$, points at most to another leaf, having minimum value $y > x$, such that among all leaves it holds: $min(y - x)$.



By means of the master theorem [21], it can be shown that the algorithm capable of finding a specific key within the B+Tree described above is executable, in the worst case, in $O(\log_m n)$ operations (with n equivalent to the number of keys entered in the tree).

Specifically, it is possible to define an index on an **R** relation by means of an *A* attribute by entering into a B+Tree entry of the form**:** $<k \in dom(A)$, *Lista_di_RID* .>

Depending on the type of attributes on which the index is defined, it can be classified into primary, if defined on the primary key, or secondary otherwise.
For a relationship, at most one primary index and zero or more secondary indexes can be defined.

Although the introduction of an index is particularly advantageous in certain circumstances, defining indexes on any attribute does not prove to be an effective choice.
The cost of an insertion or change within an index defined by means of a B+Tree remains $O(\log_m n)$ in the worst case.
There are also special cases in which the DBMS, by means of certain heuristics, chooses to consult one index rather than another depending on the level of selectivity of the attributes on which they are defined, and for sufficiently small tables it is possible to load into memory centered all pages within the buffer.

Therefore, some generic guidelines for defining indexes on relationships are listed:

- Do not define them by tables having a few pages.
- Do not define them for attributes whose value is updated frequently.
- Do not define them on attributes whose values are unselective.
- Do not define them on attributes whose values within the relationship are not evenly distributed.
- Define them on primary and foreign keys.
- Define them on tables whose queries need to show tables sorted respect to certain attributes.

In case the DBMS is in the condition of evaluating the query (with $b_1 \wedge .. \wedge b_n$ logical predicates satisfiable with the help of indexes and β a set of predicates for which an index has not been defined):

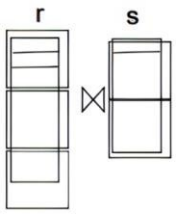
select * from relationship where $b_1 \wedge .\wedge b_n \ ; \wedge \beta$

the main strategies that can be adopted would generally be two:

- Calculation of $\{RID\}\bigcap_{i=1}^{n} b_i$ by means of the various indices, and then evaluate in memory central β : has the drawback of computing the intersection between sets, a computationally onerous operation for cardinality of $\{RID\}_{b_i}$ sufficiently large.
- Given the predicate b_j solvable by means of the most selective index I_w available, $\{RID\}_{b_j}$ is computed via I_w , and then computed in central memory $b_1 \wedge ... \wedge b_{j-1} \wedge . . . \wedge b_{(j+1)}) \wedge . . . \wedge b_n \wedge \beta$: in most cases.
  moves more pages than the method listed above but has the advantage of not having to calculate the intersection set.

One of the most popular algorithms for theta-join calculation, the Block Nested-Loop Join, is reported at the end:

```
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
```



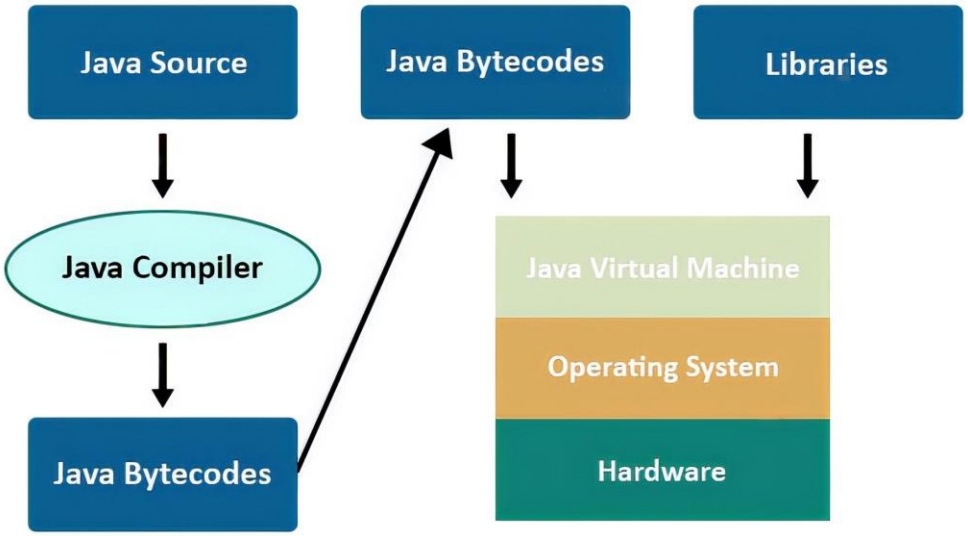
The number of blocks transferred is between $b_r + b_s$ and $b_r * b_s + b_r$, depending on the buffer capacity in relation to the size of the individual block.

Chapter 4: Project

4.1 Why the use of the language Java?:

The use of Java as an implementation language for building a DBMS could raise many questions about the relative consequences on the performance of the software produced, so why not use uninterpreted languages such as C++?

Although there is an inevitable degradation in performance, the use of an interpreter often provides more robust support with regard to security features, plus it allows a cross-platform approach while also reducing the size of the executable produced.
Java, in particular, is compiled by means of the JavaC compiler, which is responsible for generating the resulting bytecode from the source code, and then executed by means of the Java Virtual Machine (abbreviated JVM).



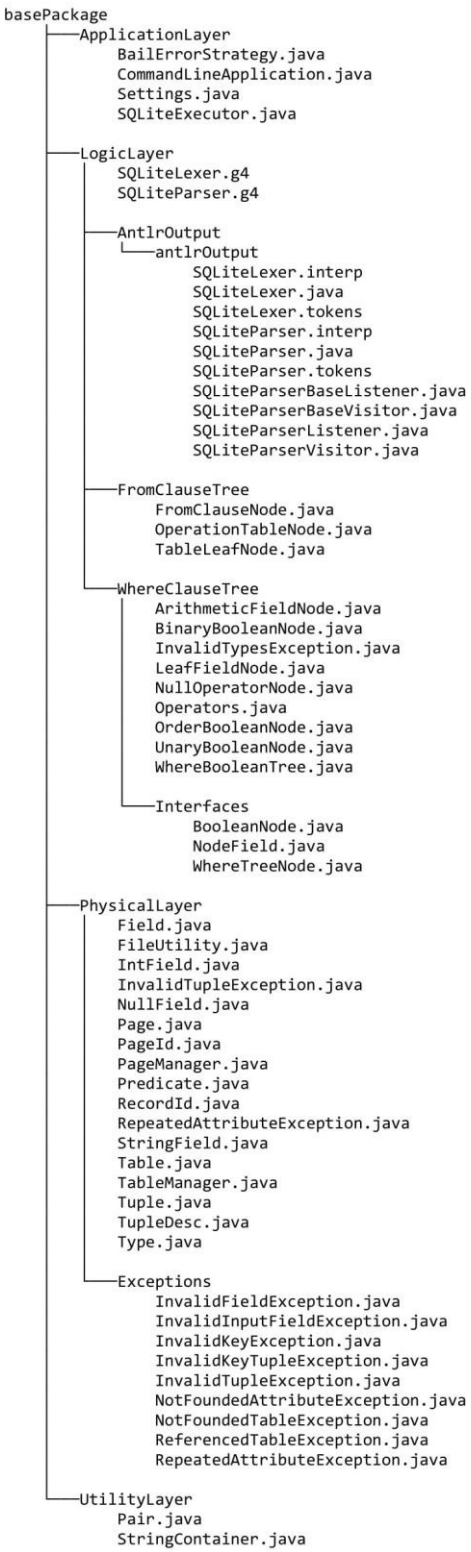
Consider features such as:

- Automatic memory management (by means of Garbage Collector)
- No reference to explicit pointers
- Exception handling
- JIT intermediate compilation

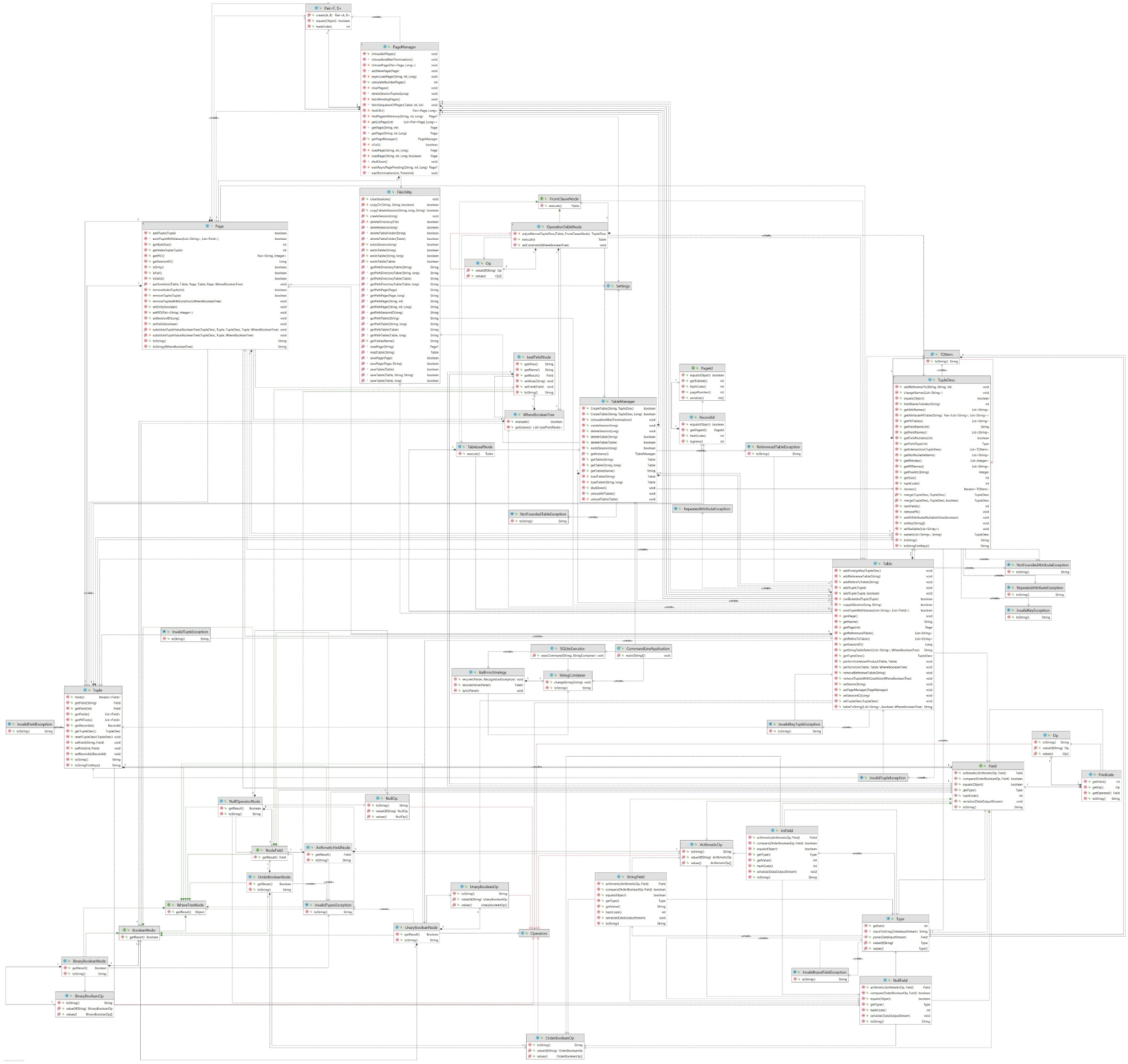
the choice of Java makes it possible to effectively resolve the trade-off imposed between the performance of the software produced and the time devoted to development and testing.

4.2 Structure of project:

A tree represented the division of design classes into packages is shown below:



The UML class diagram associated with the project is also shown:



Physical layer:

The classes contained in the package support the representation of the concepts of type, tuple, page and relation, described within the second chapter of the paper (the source [22] was used as a cue for the design of some of the functionality of the above-mentioned classes).

A page size of 16Kbytes was set, and data persistence was managed using the object serialization mechanisms provided natively by Java.

The PageManager class plays a particularly important role within the physical layer, as it allows the properties of atomicity and durability to be guaranteed.
Keeping in mind today's write and read speed gap between secondary and central memory (described in more detail in the section on DBMSs), the choice was made to mediate the passage of data between central and secondary memory (and vice versa) by means of multithreading techniques.
By means of a threadpool, whose parameters are modifiable by means of the settings class, it is possible to request from the PageManager the asynchronous fetch of sequences of multiple pages; the same threadpool is in charge of saving the pages removed from the buffer within the appropriate files dedicated to the representation of pages in the filesystem.

The Table and TableManager classes allow the consistency property to be guaranteed.

Logical layer:

The classes contained in the logical layer allow interpretation of SQL queries using the components specified within the physical layer.
The structure of the ANTLR language translator, contained in the SQLiteParser.g4 file, is described within the package, and the classes subsequently generated by ANTLR within the antlrOuput package are reported.

Application layer:

To limit the complexity of the DBMS, it was decided to postpone to future developments the implementation of a real driver dedicated to direct communication between the DBMS and the applications.
A number of classes have been specified within the package that allow SQL commands to be sent, a direct manner, to the DBMS by means of a shell.

In order for the program to represent the result of the queries correctly, the shell row buffer, on which the CommandLineApplication class will be executed, must be large enough.

4.3 SQL constructs implemented:

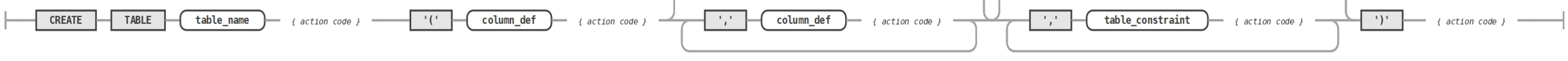
In the following section, the grammars associated with the translator portions capable of interpreting DDL, DML and DQL queries will be specified; examples of parse trees generated by means of the grammar for some typical language queries will also be illustrated. Whenever possible, it will be preferred to make the structure of the grammar explicit by means of a syntactic graph [23], rather than using the EBNF meta-syntax directly.

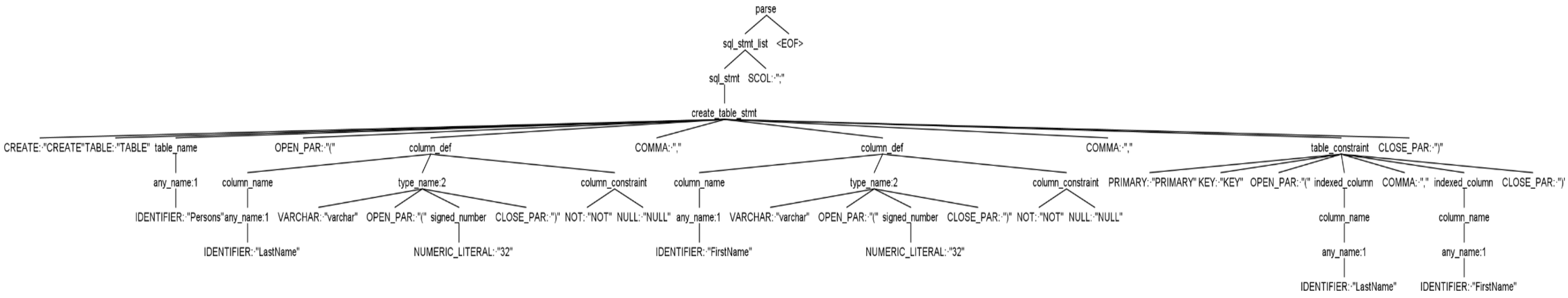
DDL Parsification:

The portion of the grammar covered supports all types of constructs introduced within the section on SQL language, in particular it is possible to declare FOREIGN KEY and PRIMARY KEY, in case a primary key is not defined the DBMS will place primary key the set of attributes on which the relationship is defined.
For simplicity, the DBMS supports as data types: VARCHAR and INT, the approach with which it was written guarantees future expansion of the types accepted by the language without having to reformulate the structure of the translator or classes useful for DDL interpretation.

Parsing tree built on the query: CREATE TABLE

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


CREATE
TABLE
table_name
{ action code }
'('
column_def
{ action code }
','
column_def
{ action code }
','
table_constraint
{ action code }
')'
{ action code }


parse
sql_stmt_list
<EOF>
sql_stmt
SCOL:";"
create_table_stmt
CREATE:"CREATE"
TABLE:"TABLE"
table_name
any_name:1
IDENTIFIER:"Persons"
OPEN_PAR:"("
column_def
column_name
any_name:1
IDENTIFIER:"LastName"
type_name:2
VARCHAR:"varchar"
OPEN_PAR:"("
signed_number
NUMERIC_LITERAL:"32"
CLOSE_PAR:")"
column_constraint
NOT:"NOT"
NULL:"NULL"
COMMA:","
column_def
column_name
any_name:1
IDENTIFIER:"FirstName"
type_name:2
VARCHAR:"varchar"
OPEN_PAR:"("
signed_number
NUMERIC_LITERAL:"32"
CLOSE_PAR:")"
column_constraint
NOT:"NOT"
NULL:"NULL"
COMMA:","
table_constraint
PRIMARY:"PRIMARY"
KEY:"KEY"
OPEN_PAR:"("
indexed_column
column_name
any_name:1
IDENTIFIER:"LastName"
COMMA:","
indexed_column
column_name
any_name:1
IDENTIFIER:"FirstName"
CLOSE_PAR:")"

DML parsing:

Just as in the previous case, a grammar has been defined that can accept all DML constructs covered by the section on SQL syntax.

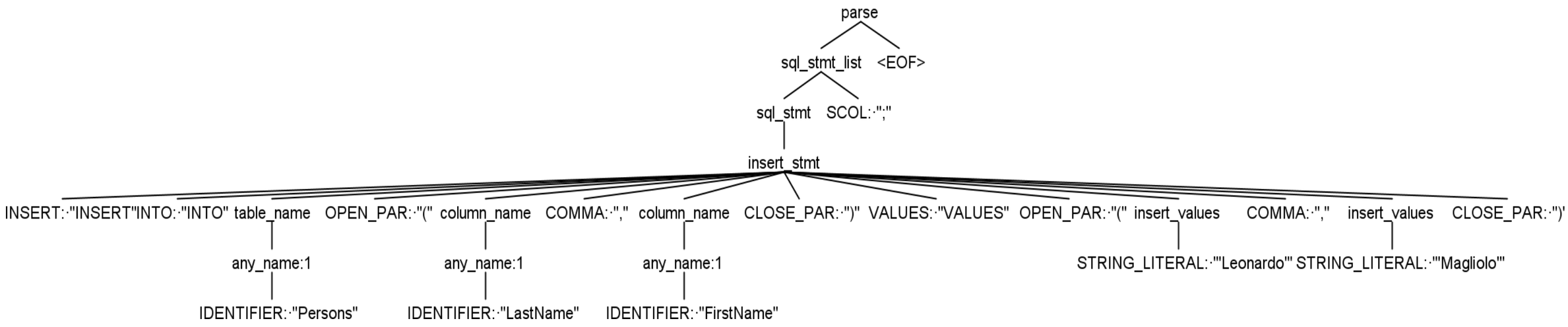
It is possible, other things, to make explicit only some attributes of a relation as long as the omitted attributes are nullable, while also varying the order in which they were defined within the DDL query creating the relation.

Given the more complex structure of the grammar considered, keeping in mind that syntactic graphs sometimes have the drawback of extending excessively to the right, it was decided to represent the grammar by means of EBNF syntax:

```
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;
```

Parsing tree built on the query:

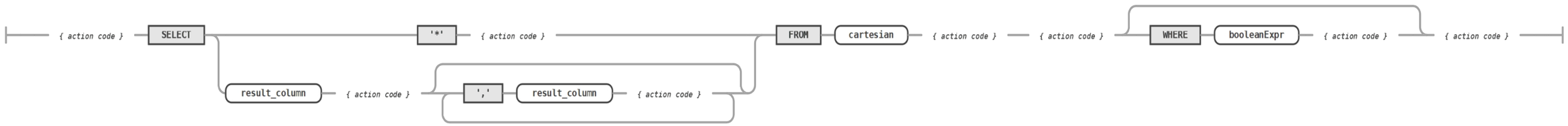
INSERT INTO Persons (LastName , FirstName)
VALUES ('Leonardo', 'Magliolo');



DQL parsing:

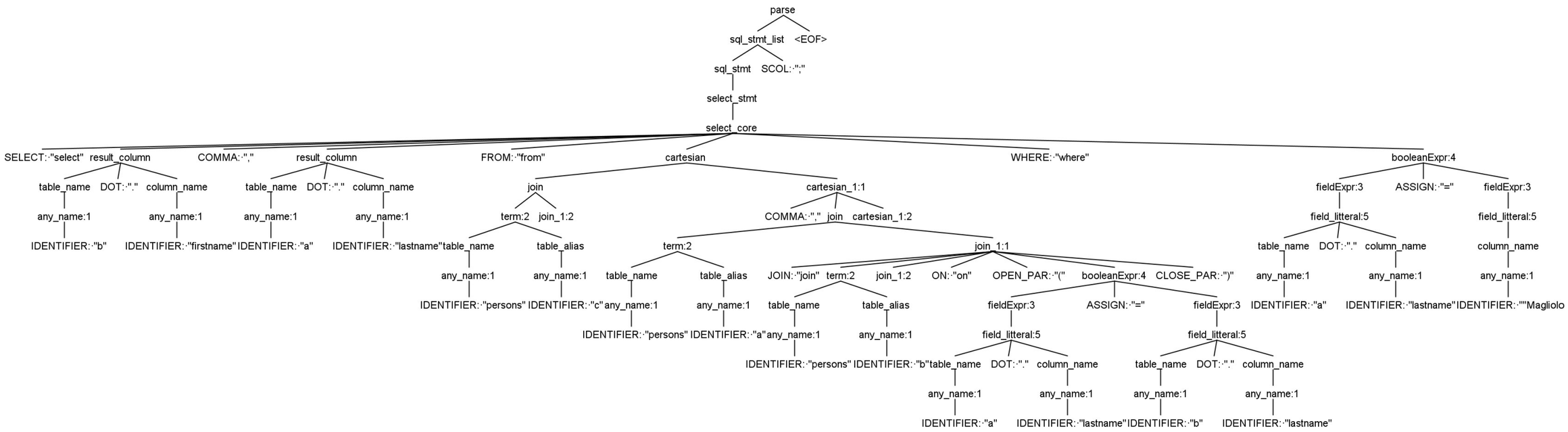
Although the fundamental constructs of DQL queries have been finalized, some marginal aspects will need further development to be implemented. At its current version, the DBMS lacks the ability to take advantage of functions, to be able to sort records, or to perform subqueries.

It remains , however, to define aliases for relationships, to select tuples by means of WHERE and JOIN ON clauses using Boolean predicates, and it remains finally possible to use the wildcard * to keep all attributes or use the contents of the attribute list after the SELECT command to select only some attributes of the resulting relationship, even in random order.



Parsing tree built on the query:

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



Optimizations were made regarding the portion of the translator that is responsible for constructing the syntactic tree of the relational expression associated with the SQL query.

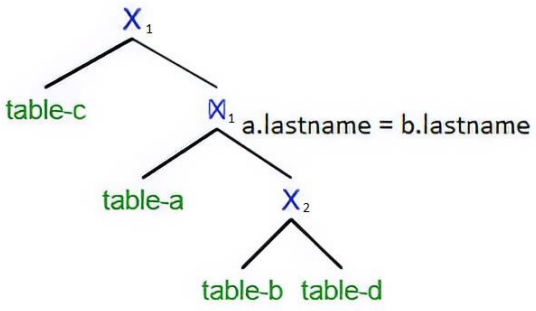
In particular, the translator grammar was reformulated in such a way to prioritize the execution theta-joins rather than Cartesian products, moving join nodes toward frontier whenever possible.

.

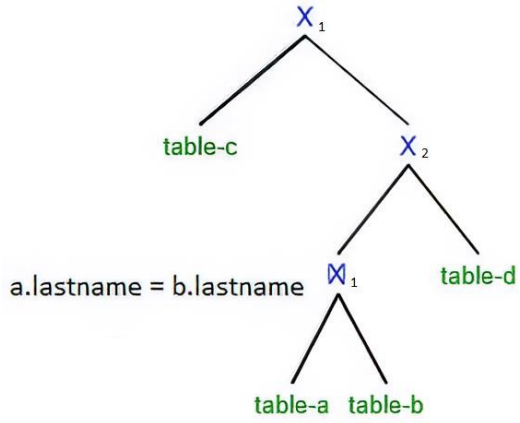
```
cartesian returns [FromClauseNode nodeToReturn]:
    n1=join  n2=cartesian_1[$n1.nodeToReturn] {$nodeToReturn = $n2.nodeToReturn;};

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);
    }
    n2 = join_1[$n2.nodeToReturn]{$nodeToReturn = $n2.nodeToReturn;}
    |{$nodeToReturn = n;};
```

Taking the example query portion into consideration:

persons c, persons a join persons b on (a.lastname = b.lastname), persons d two different possible syntactic trees returning the same relationship are reported:



(Non-optimized tree, generated considering only the order of precedence from left to right)



(Optimized tree, generated considering the join operator with precedence greater than to the Cartesian product)

Since the evaluation order is bottom-up, the nodes closest to tree frontier will be executed first. To show that the optimized tree is actually able to move fewer tuples than the unoptimized tree, we proceed to calculate the tuples moved:

For unoptimized tree:

- Cartesian product cost X_2: $|\mathbf{b}|*|\mathbf{d}|$.
- Cost of theta-join $\bowtie_{1\ (\theta:)\ (a.=b.lastname}$: $(|\mathbf{b}|*|\mathbf{d}|)*|\boldsymbol{a}|$.
- Cost of Cartesian product X_1: $\alpha*|\boldsymbol{c}|$,$con\ 0 \leq \alpha \leq (|\mathbf{b}|*|\mathbf{d}|)*|\boldsymbol{a}|$.
- Sum of costs: $\Phi = |\mathbf{b}|*|\mathbf{d}|+(|\mathbf{b}|*|\mathbf{d}|)*|\boldsymbol{a}|+\alpha*|\mathbf{c}|$.

For optimized tree:

- Cost of theta-join $\bowtie_{1)\ (\theta:)\ (a.=b.lastname}$: $|\mathbf{a}|*|\mathbf{b}|$.
- Cartesian product cost X_2: $\beta*|\mathbf{d}|$,$con\ 0 \leq \beta \leq |\mathbf{a}|*|\mathbf{b}|$.
- Cartesian product cost X_1: $(\beta*|\mathbf{d}|)*|\mathbf{c}|$.
- Sum of costs: $\varphi = |\mathbf{a}|*|\mathbf{b}|+\beta*|\mathbf{d}|+(\beta*|\mathbf{d}|)*|\boldsymbol{c}| = |\mathbf{a}|*|\mathbf{b}|+\beta*|\mathbf{d}|(1+|\boldsymbol{c}|)$.

To avoid considering further optimization, assume that |**a**|, |**b**|*e*|**c**|belong to the same order of magnitude.

In this case, since |***a***|≈|**b**|≈ |**c**| , in most casesφ < Φ since β<αe

β*|**d**|<|**b**|*|**d**|* |***a***|.

Specifically, to determine the value ofβ with some probability, it is necessary to be able to calculate the selectivity of the predicateθ by means of the statistics collected by the DBMS on the tables affected by the theta-join.

4.4 Main data structures and algorithms used:

4.4.1 Expressions solver clauses conditional:

The declarable conditional clauses in SQL exploit operators that can be classified mainly into: logical connectives, order operators, arithmetic operators, special operators exploiting null-value logic, and functions; they can also be subdivided into unary or binary.

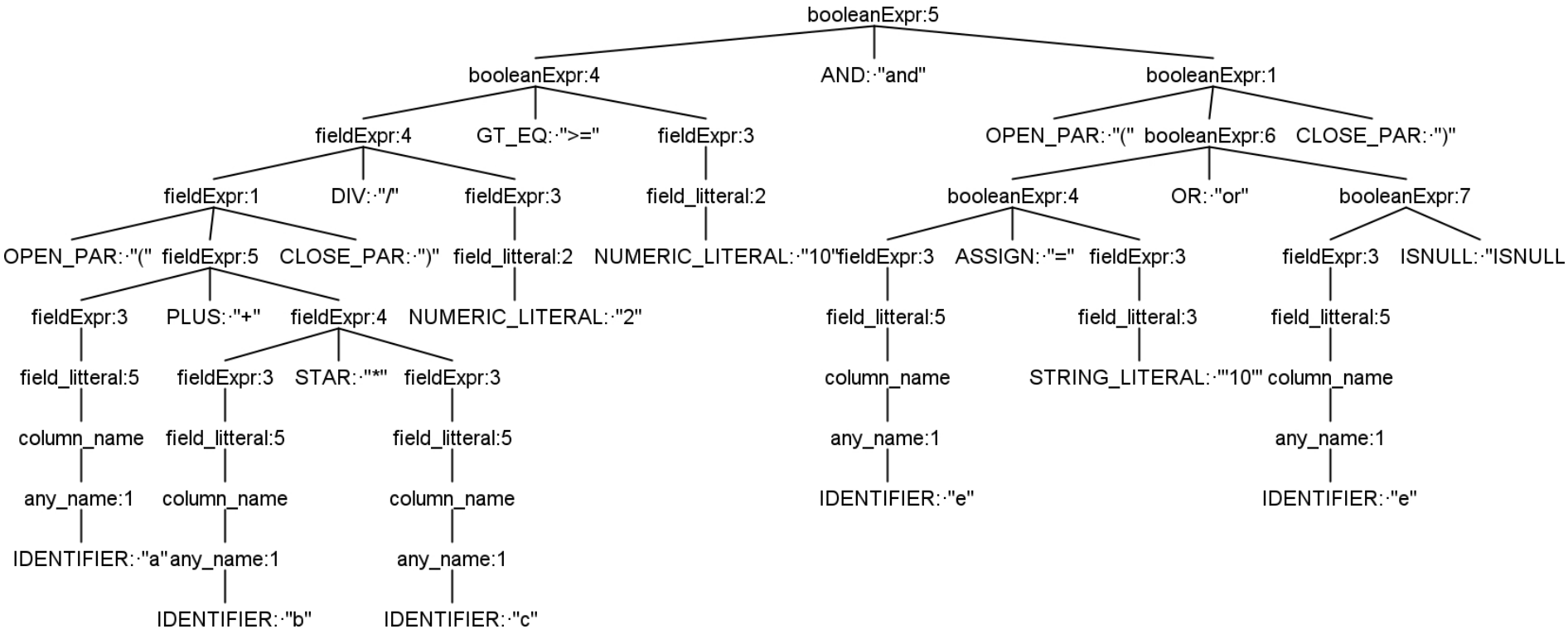
The version SQL implemented supports:

- Logical connectives: By means of the BynaryBooleanNode class it is possible to define AND, OR operators; by means of the UnaryBooleanNode class it is possible to define the NOT operator.
- Order operators: Defined as a function**Field** x **Field**→**{True, False}**
  implemented by means of the OrderBooleanNode class.
  Available order operators include: =, >,, <, ≤*e*<>.
  Note how the following definition does not allow syntaxes of the type *m*<*a*< *n* .
  However, it is possible to express an equivalent condition by means of the logical connectives: *.m < a AND a < n*
- Arithmetic operators: Defined as a function**Field** x **Field**→**Field**implemented
  By means of the ArithmeticFieldNode class.
  Available arithmetic operators include: +, *, -, /.
- Special operators based on null-value logic: By means of NullOperatorNode, the ISNULL operator can be defined.
  Combined with logical connectives, the condition to NOT NULL can be expressed by the following expression: NOT (to ISNULL).

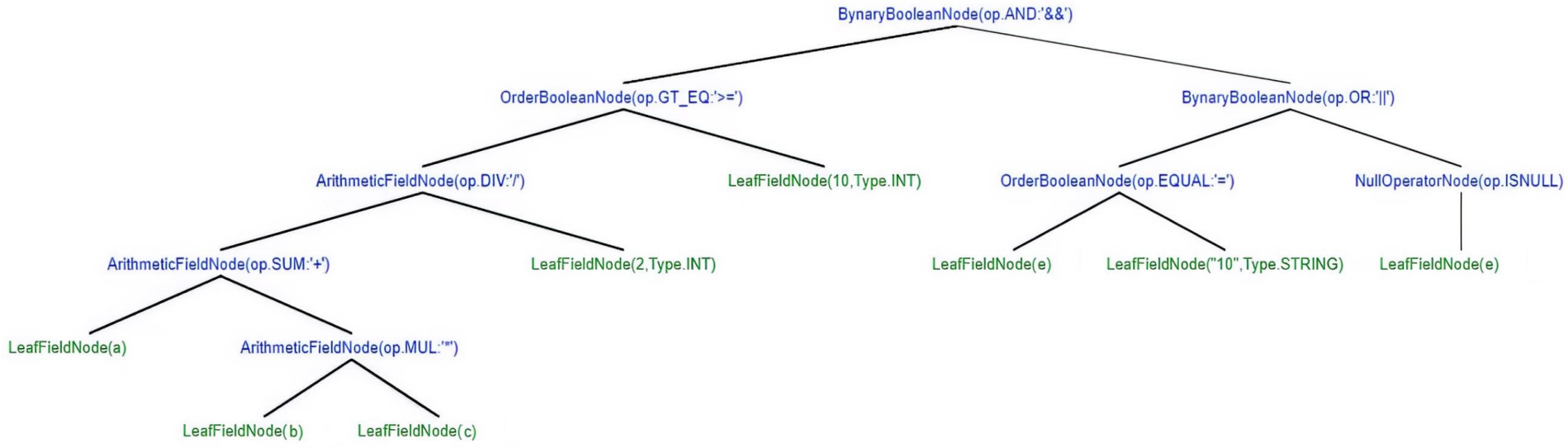
Grammar operators are rated in order of priority from left to right: arithmetic, order, and logical.
As discussed in the previous paragraphs, since the order of evaluation is bottom-up, it is necessary for the grammar to parse the elements of language having lower priority first.

The parsing tree, generated from the SQL grammar portion, built on the example condition is given:

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



Since the condition is to be applied to all tuples belonging to a certain relation, it was chosen to generate, by means of the translator, an equivalent tree whose leaves represent constants (Field) or attributes associated with the relation:



The tree will later be used by an algorithm that can replace the values within its leaves with the fields fields associated each of the tuples involved in the relationship:

```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 Implementation Block Nested Loop Join:

In order to maintain low coupling and at the same time promote the cohesion of the functions implemented within the classes, it was chosen to be able to directly access a given tuple only through the page instance that contains it.
For this reason, implementation of the Block Nested Loop Join algorithm required its specification in two parts:

1. Code related to the Table class:

```
1 public void performJoin(Table leftTable, Table rightTable, WhereBooleanTree tree)
2             throws RepeatedAttributeException,... NotFoundedAttributeException{
3         ...
4         if(leftTable.lastID.get() > 0 && rightTable.lastID.get() > 0){
5             ...
6             for(int i = 1; i<=leftTable.lastID.get(); i++){
7                 Page leftPage = leftTable.getPage(i);
8                 for(int j = 1; j<=rightTable.lastID.get(); j++){
9                     Page.performJoin(this, leftTable, leftPage, rightTable, rightTable.getPage(j), tree);
10                }
11            }
12        }
13    }
```

2. Code related to the Page class:

```
1 static void performJoin(Table table, Table leftTable, Page leftPage, Table rightTable, Page rightPage, WhereBooleanTree tree)
2         throws RepeatedAttributeException,... NotFoundedAttributeException{
3         for(Tuple tupleLeft: leftPage.tuples){
4             for(Tuple tupleRight: rightPage.tuples){
5                 substituteTupleValueBooleanTree(leftTable.getTupleDesc(), tupleLeft, rightTable.getTupleDesc(), tupleRight, tree);
6                 if(tree.evaluate()){ //perform the following actions only if the theta conditions is true
7                     Tuple toAdd = new Tuple(table.getTupleDesc());
8                     List<Field> fields = tupleLeft.getFields();
9                     for(int i = 0; i < fields.size(); i++){ //fill the left side of the new tuple
10                        toAdd.setField(i, fields.get(i));
11                    }
12                    int prevLen = fields.size();
13                    fields = tupleRight.getFields();
14                    for(int i = prevLen; i < (fields.size()+prevLen); i++){//fill the right side of the new tuple
15                        toAdd.setField(i, fields.get(i-prevLen));
16                    }
17
18                    table.addTuple(toAdd, false); // false allow to not perform integrity checks
19                }
20            }
21        }
22    }
```

Notice how in the second portion of the code, on the fifth line, the substituteTupleValueBooleanTree function is called to substitute the values of the leaves of the conditional tree so that the condition θ on the considered tuple can be evaluated.

Chapter 5: Analysis performance

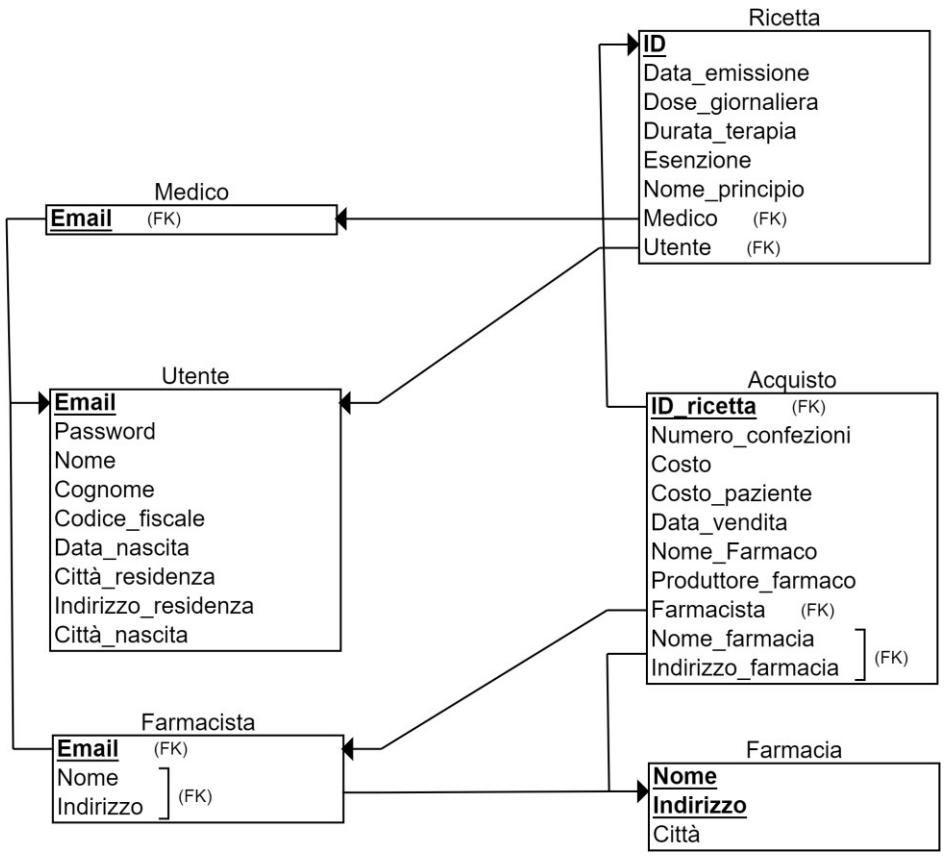
A performance analysis of the DBMS was carried out, useful for experimental verification of the observations made in the previous chapters regarding the effectiveness of the implementation choices made.
The comparative analysis compares the time taken to execute typical types of DML and DQL queries by considering data obtained using SQLite [24]: a small-scale DBMS having similar characteristics and purposes to the one implemented.

Queries were run on a single machine, having the following harware components:

- CPU: Intel Core i5-3570
- RAM: 8 GB DDR3 1333MHz
- Mass storage: SSD Crucial BX500 1 TB

The data obtained remain indicative, given use of only one computer and the limitations of the architecture employed.

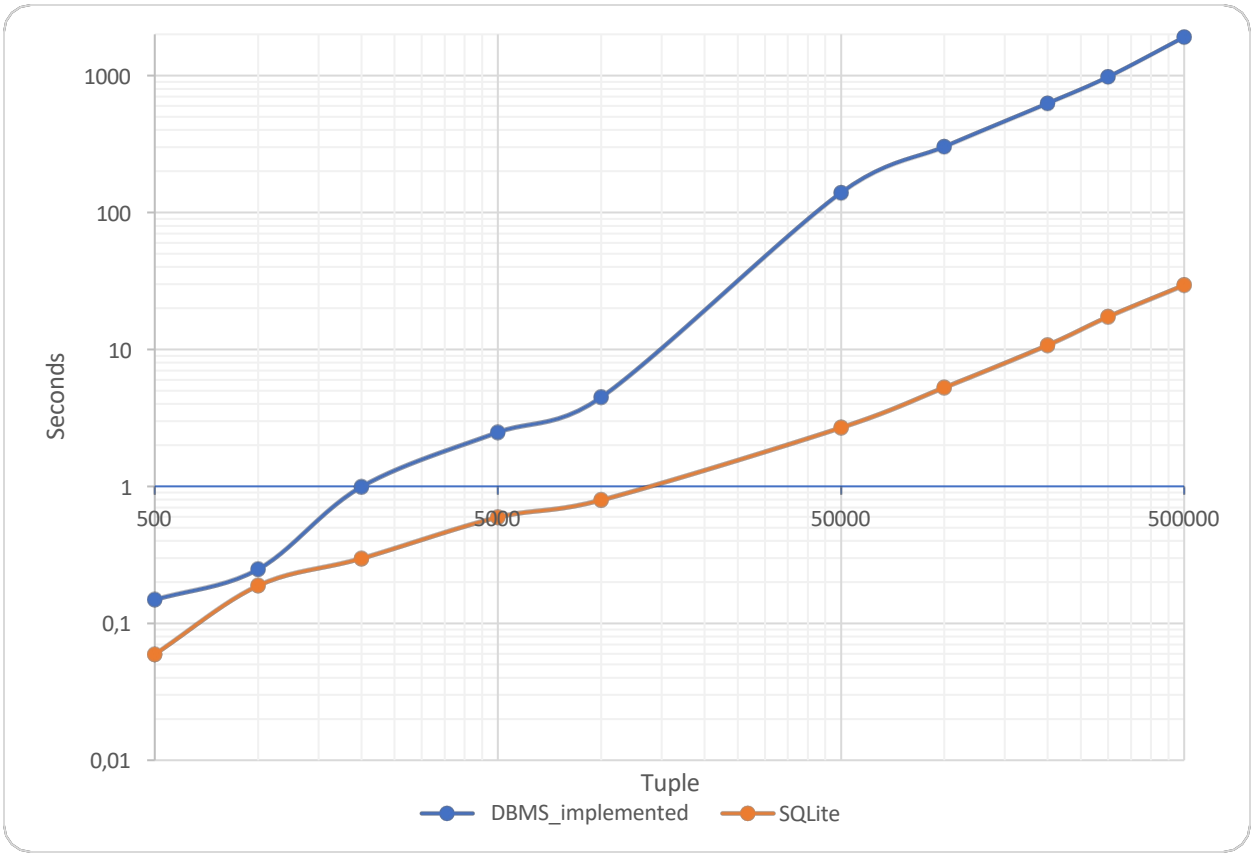
The queries used exploit the following example relational schema:



Sample DML query:

timing of insertion on the 'Pharmacist' table was evaluated:

Number tuples entered	DBMS insertion time designed	SQLite Insertion Time
500	0.15 sec	0.06 sec
1000	0.25 sec	0.19 sec
2000	1.00 sec	0.30 sec
5000	2.50 sec	0.60 sec
10000	4.50 sec	0.8 sec
50000	2 min 20 sec	2.7 sec
100000	5 min 3 sec	5.30 sec
200000	10 min 28 sec	10.8 sec
300000	16 min 19 sec	17.4 sec
500000	31 min 42 sec	29.6 sec



As can be seen from the graph depicting the data obtained (the axes of which are represented with a logarithmic scale), the performance gap as the number of tuples increases appears to have a nonlinear trend.

The progressive degradation of response time is largely attributable to the verification of integrity constraints in the absence of indexes.

The calculation of tuples moved to make n insertions is reported, guaranteeing key constraint imposed on a generic relation R of the schema, having |R| = m before making the insertions:

- Tuples moved, in the worst case, in the absence of index:

$$\sum_{i=1}^{n} m+i = m * \sum_{i=1}^{n} i = m * \frac{n*(n+1)}{2} = m * \frac{n^2+n}{2} \in \mathcal{O}(m*n^2)$$

- Tuples moved, in the worst case, in the presence of an index defined on the primary key of the relation R, having branching-factor equivalent to b:

$$\sum_{i=1}^{n} \mathcal{O}(\log_b(m+i)) \leq \sum_{i=1}^{n} \mathcal{O}(\log_b(m+n)) \in \mathcal{O}(n*\log_b(m+n))$$

Similar results are obtained by considering referential integrity constraints defined on relations external to R.

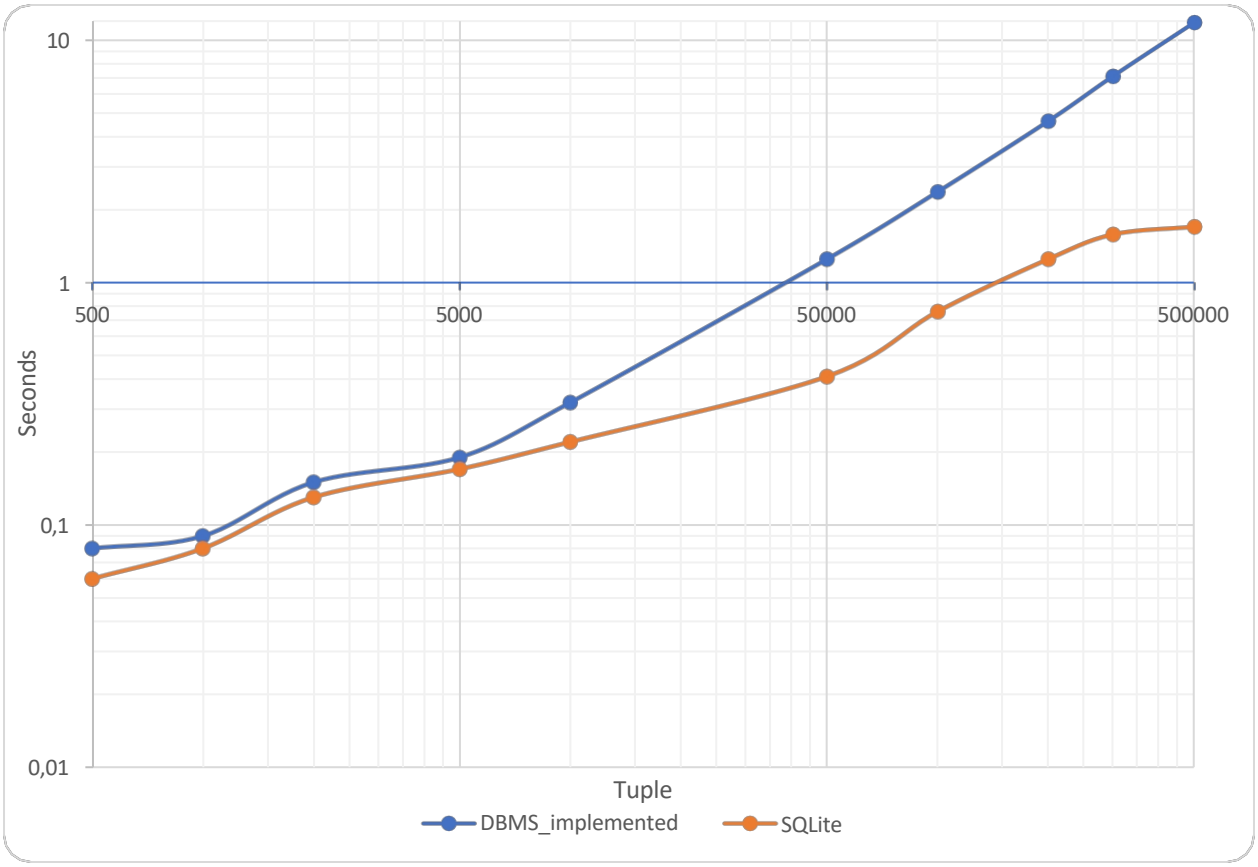
By disabling integrity constraint checks by the DBMS, it is possible to enter 500000 tuples in a time frame of less than 20 seconds, so it is reasonable to assume that with the implementation of B+Tree-based indexes, the nonlinear performance gap can be closed.

Example of a simple DQL query:

Execution times for the query were evaluated:

```
select * from pharmacy where City=  'nonExisting'
```

Cardinality of the 'pharmacy' relationship	DBMS runtime designed	SQLite execution time
500	0.08 sec	0.06 sec
1000	0.09 sec	0.08 sec
2000	0.15 sec	0.13 sec
5000	0.19 sec	0.17 sec
10000	0.32 sec	0.22 sec
50000	1.25 sec	0.41 sec
100000	2.37 sec	0.76 sec
200000	4.64 sec	1.25 sec
300000	7.11 sec	1.58 sec
500000	11.85 sec	1.70 sec

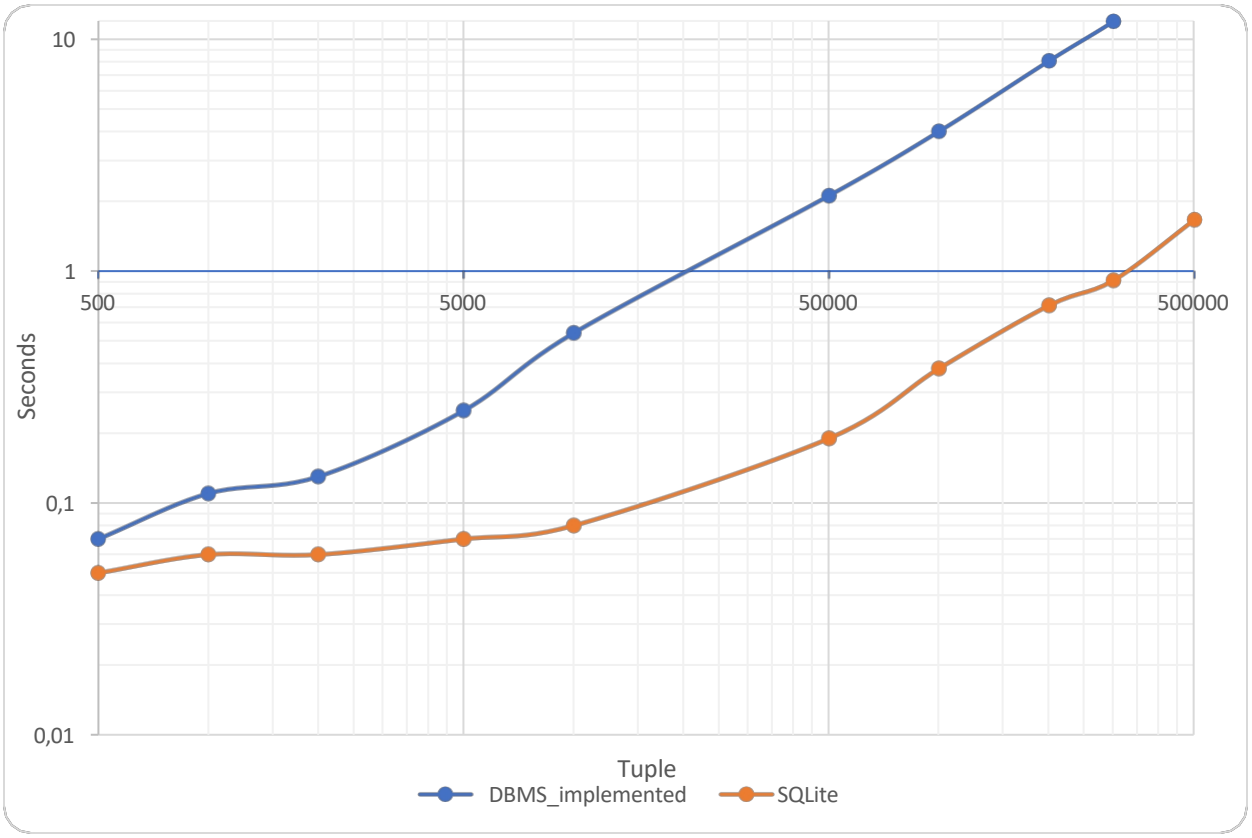


Example of DQL query with join sequences:

Execution times for the query were evaluated:

```
select a.Recipe_ID, a.Number_of_packs, a.Cost, a.Patient_cost, a.Date_sold,a.Name_pharmacy,
a.Manufacturer_pharmacy, u.First_Name, u.Last_Name, u.Email,a.First_name_pharmacy,
a.Address_pharmacy
from purchase a join user u on (a.Pharmacist= u.Email) join recipe r on (a.Recipe_ID =
r.ID) where u.Email = 'valueUnico';
```

Cardinality of relationships	DBMS runtime designed	SQLite execution time
500	0.07 sec	0.05 sec
1000	0.11 sec	0.06 sec
2000	0.13 sec	0.06 sec
5000	0.25 sec	0.07 sec
10000	0.54 sec	0.08 sec
50000	2.11 sec	0.19 sec
100000	3.99 sec	0.38 sec
200000	8.03 sec	0.71 sec
300000	11.87 sec	0.91 sec
500000	20.18 sec	1.66 sec



Again, the use of indexes guarantees lower response times, due algorithms executable by the physical optimizer to compute theta-join.

Chapter 6: Conclusions

The design and implementation of the software produced allowed for a deeper examination of the interactions between the teachings of formal languages, translators and databases, apt to the implementation and operation of today's RDBMSs.

Although the project is capable of performing the main DDL, DML and DQL functions necessary for a DBMS to be used in practice, the limited amount of time and complexity of the project handled justify improvements that can be introduced in future developments:

- Development of a dedicated driver for communication between external applications and DBMS
- Implementation of multiple types accepted by DDL syntax
- Extension of DQL syntax, applicable to the interpretation of functions and sub-queries
- Index implementations using B+Tree
- Additional logical optimizations, which can be introduced by manipulating the parse tree of interpreted DQL queries

Sources:

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

24. https://www.sqlite.org/about.html