Contents

AQL Parser Report

Introduction

The Archetype Query Language (AQL) is a query language specifically designed for querying Electronic Health Records (EHR). AQL provides a structured and standardized way to extract specific information from EHR systems.

Objective

The primary objective of this report is to provide a detailed overview and understanding of the AQL parser developed using ANTLR. The report aims to explain the functionalities, development process, testing methodologies, and the challenges faced during the development of the AQL parser. Additionally, this report will highlight the tools and technologies used and provide examples of AQL queries to demonstrate the capabilities of the parser.

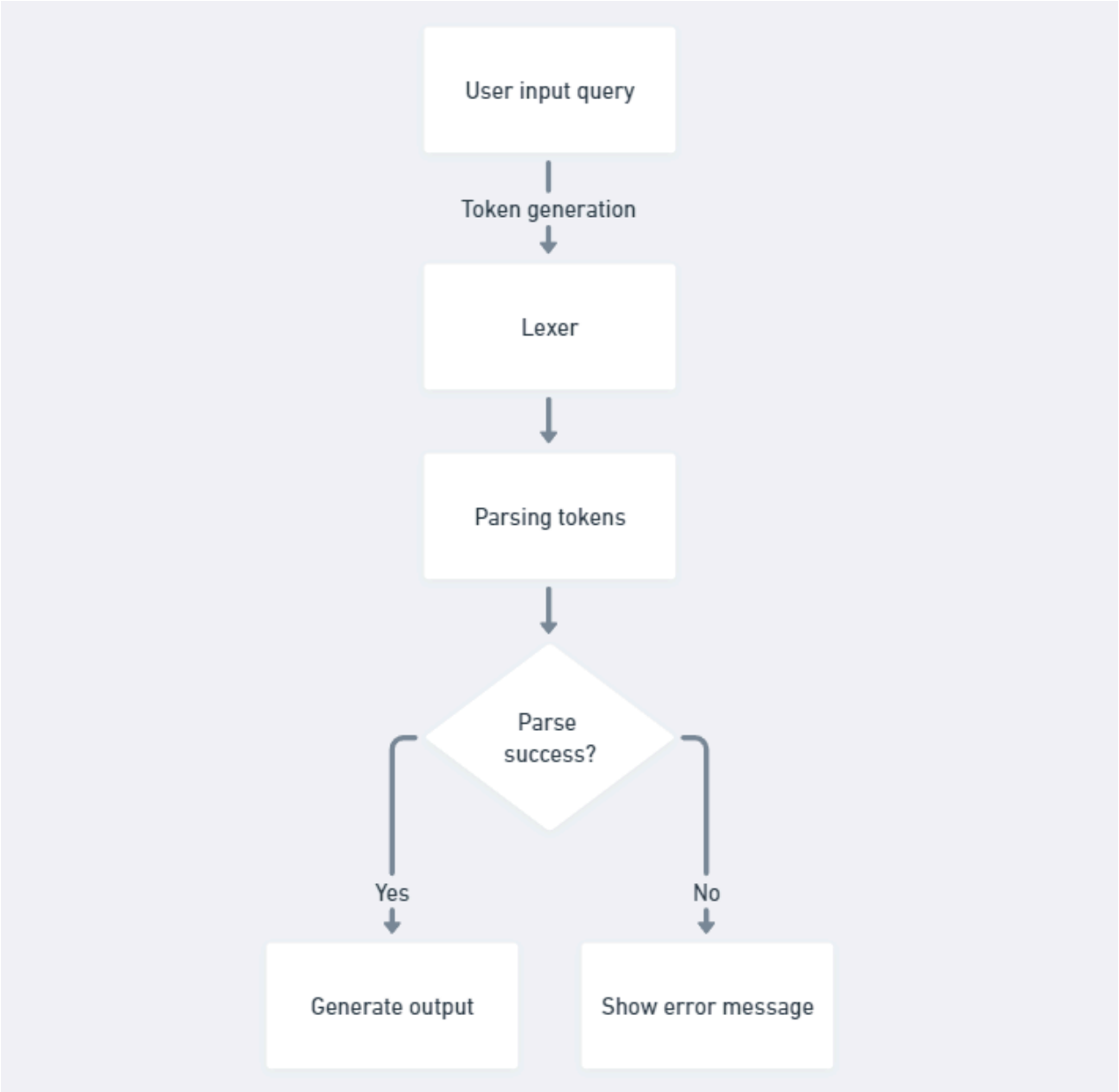
AQL Parser Overview

1. **What is AQL Parser?**

An AQL parser is a software component that interprets and processes AQL queries. It takes an AQL query as input, validates its syntax, and generates a parse tree as output. The parse tree represents the hierarchical structure of the query, enabling the parser to understand and execute the query effectively.

2. **Key Features**

- Lexer: Tokenizes the input AQL query.
- Parser: Constructs a parse tree based on the tokenized AQL query.
- Parse Tree Generation: Represents the hierarchical structure of the AQL query.


User input query
Token generation
Lexer
Parsing tokens
Parse success?
Yes
No
Generate output
Show error message

Syntax Description and Features of AQL

AQL is designed to efficiently retrieve and analyze data stored in archetype-based repositories, with a focus on semantic meaning and complex data structures.

Data Types and Path Syntax

AQL supports various primitive data types to handle different kinds of data effectively. These include:

- **String:** Used for textual data.
- **Integer:** Represents whole numbers.
- **Real:** Represents floating-point numbers.
- **Boolean:** Represents true or false values.
- **Dates and Times:** Handles date and time values.

A key aspect of AQL is its utilization of path syntax, which allows users to refer to specific nodes or data values within archetypes. AQL paths resemble navigating folders and files on a computer, using dot notation (e.g., **a/b/c**) to pinpoint data elements within nested archetype structures.

Advanced Operators

In addition to basic comparison operators (>, <, =, etc.), AQL provides a set of advanced operators to manipulate and analyze retrieved data:

- **MATCHES:** Performs pattern matching using regular expressions.
- **EXISTS:** Checks for the existence of a specified condition.
- **NOT, AND, OR:** Combines conditions logically to form complex query expressions.

These operators enable users to construct sophisticated queries tailored to specific data retrieval requirements.

Variables and Path Syntax

AQL leverages variables to represent nodes within archetype structures, allowing for dynamic query construction. Variables are assigned to specific points in archetype paths, facilitating flexibility and reusability in query design.

OpenEHR Path Syntax

The use of OpenEHR path syntax is integral to AQL, enabling precise navigation through complex archetype structures. OpenEHR paths incorporate special codes within brackets to identify node identifiers defined in archetype ontologies, ensuring clarity and specificity in multi-archetype scenarios.

Comparison with SQL

While AQL shares similarities with SQL in terms of query structure, it is tailored for archetype-based data retrieval. Unlike SQL,AQL focuses on semantic understanding within archetypes and offers specialized features to optimize querying of healthcare data.

AQL Query Example

```
SELECT count(*) AS patient_count
FROM EHR [ehr_id/value=$patientId]
CONTAINS COMPOSITION c[openEHR-EHR-COMPOSITION.encounter.v1]
WHERE c/context/start_time > "2023-01-01"
```

AQL Query

In this AQL query:

- **SELECT count(*) AS patient_count:** Specifies the desired output as the count of patients.
- **FROM EHR [ehr_id/value=$patientId]:** Sets the starting point as electronic health records (EHRs) with a specific patient ID.
- **CONTAINS COMPOSITION c[openEHR-EHR-COMPOSITION.encounter.v1]:** Navigates within the EHRs to search for "COMPOSITION" archetypes of type "encounter.v1".
- **WHERE c/context/start_time > "2023-01-01":** Filters encounters that occurred after January 1st, 2023.

This example showcases how AQL syntax is used to formulate precise and meaningful queries for healthcare data analysis.

AQL Query Components

1. **SELECT Clause**

The SELECT clause specifies the data elements to be retrieved from the EHR.
Example: SELECT e/ehr_id/value

2. **FROM Clause**

The FROM clause specifies the EHR components or records from which the data will be retrieved.

Example: FROM EHR e

3. **WHERE Clause**

The WHERE clause specifies the conditions that the retrieved data must meet.

Example: WHERE e/ehr_id/value = '1234'

4. **ORDER BY Clause**

The ORDER BY clause specifies the sorting order of the retrieved data.

Example: ORDER BY e/ehr_id/value

5. **LIMIT Clause**

The LIMIT clause specifies the maximum number of records to be retrieved.

Example: LIMIT 10

```
SELECT                                                  -- Select clause
   o/data[at0001]/.../items[at0004]/value AS systolic,  -- Identified path with alias
   o/data[at0001]/.../items[at0005]/value AS diastolic,
   c/context/start_time AS date_time
FROM                                                    -- From clause
   EHR[ehr_id/value=$ehrUid]                            -- RM class expression
      CONTAINS                                          -- containment
         COMPOSITION c                                  -- RM class expression
            [openEHR-EHR-COMPOSITION.encounter.v1]      -- archetype predicate
         CONTAINS
            OBSERVATION o [openEHR-EHR-OBSERVATION.blood_pressure.v1]
WHERE                                                   -- Where clause
   o/data[at0001]/.../items[at0004]/value/value >= 140 OR    -- value comparison
   o/data[at0001]/.../items[at0005]/value/value >= 90
ORDER BY                                                -- order by datetime, latest first
   c/context/start_time DESC
```

Reference : https://specifications.openehr.org/releases/QUERY/latest/AQL.html

AQL Parser Development

1. Lexer

a. Tokenization

Tokenization is the process of breaking the input AQL query into tokens, which are the basic units of the query.

Example:

Input: SELECT e/ehr_id/value

Tokens: SELECT, e/ehr_id/value

b. Whitespace and Comment Handling

The lexer removes unnecessary whitespace and comments from the input query to simplify the parsing process.

Example:

Input: SELECT e/ehr_id/value --This is a comment

Output: SELECT e/ehr_id/value

c. Token Identification

The lexer identifies keywords, identifiers, literals, and symbols in the input query.

Example:

Input: SELECT e/ehr_id/value

Tokens: SELECT, IDENTIFIER, IDENTIFIER

2. Parser

a. Constructing a Parse Tree

The parser constructs a parse tree, which represents the hierarchical structure of the AQL query.

b. Validating the Query's Syntax

The parser validates the syntax of the AQL query against the defined grammar rules.

Example:

Valid Query: SELECT e/ehr_id/value FROM EHR e

Invalid Query: SELECT e/ehr_id/value EHR e

3. Parse Tree Generation

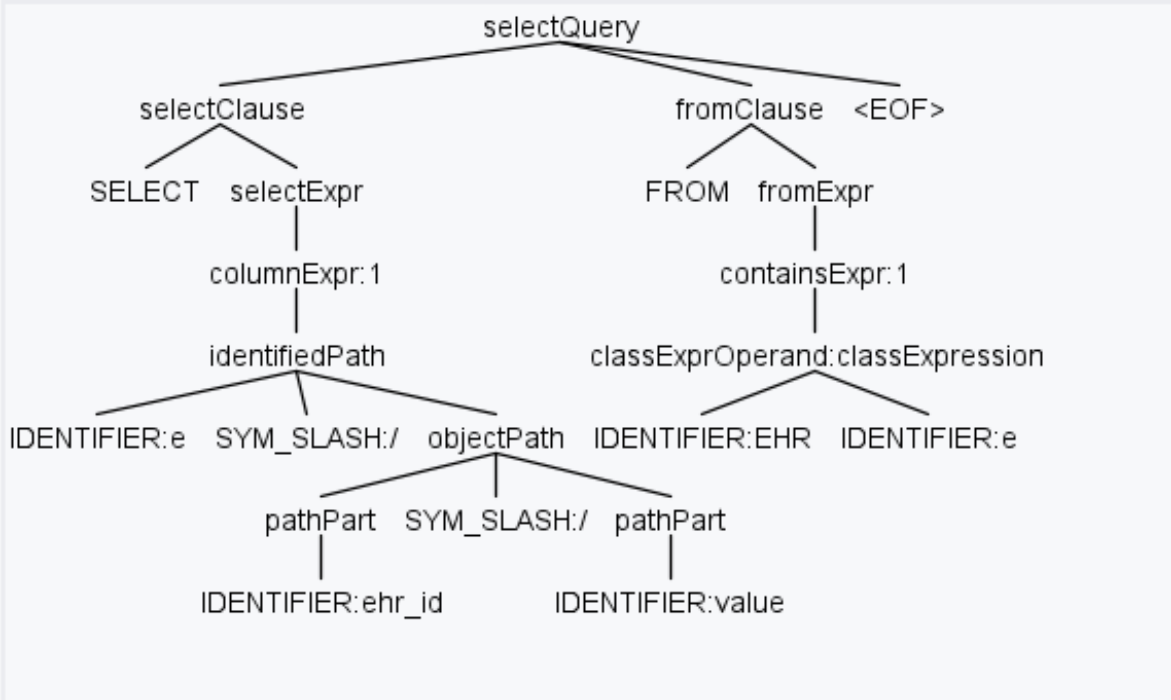
a. Generating the Parse Tree

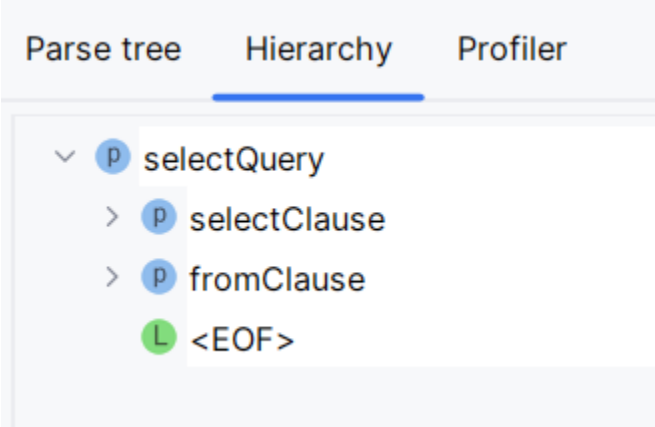
The parse tree generation process constructs a detailed hierarchical representation of the AQL query based on the validated syntax.

Example:
Query

```
1  SELECT e/ehr_id/value FROM EHR e
```

Parse Tree





Parse tree | Hierarchy | Profiler

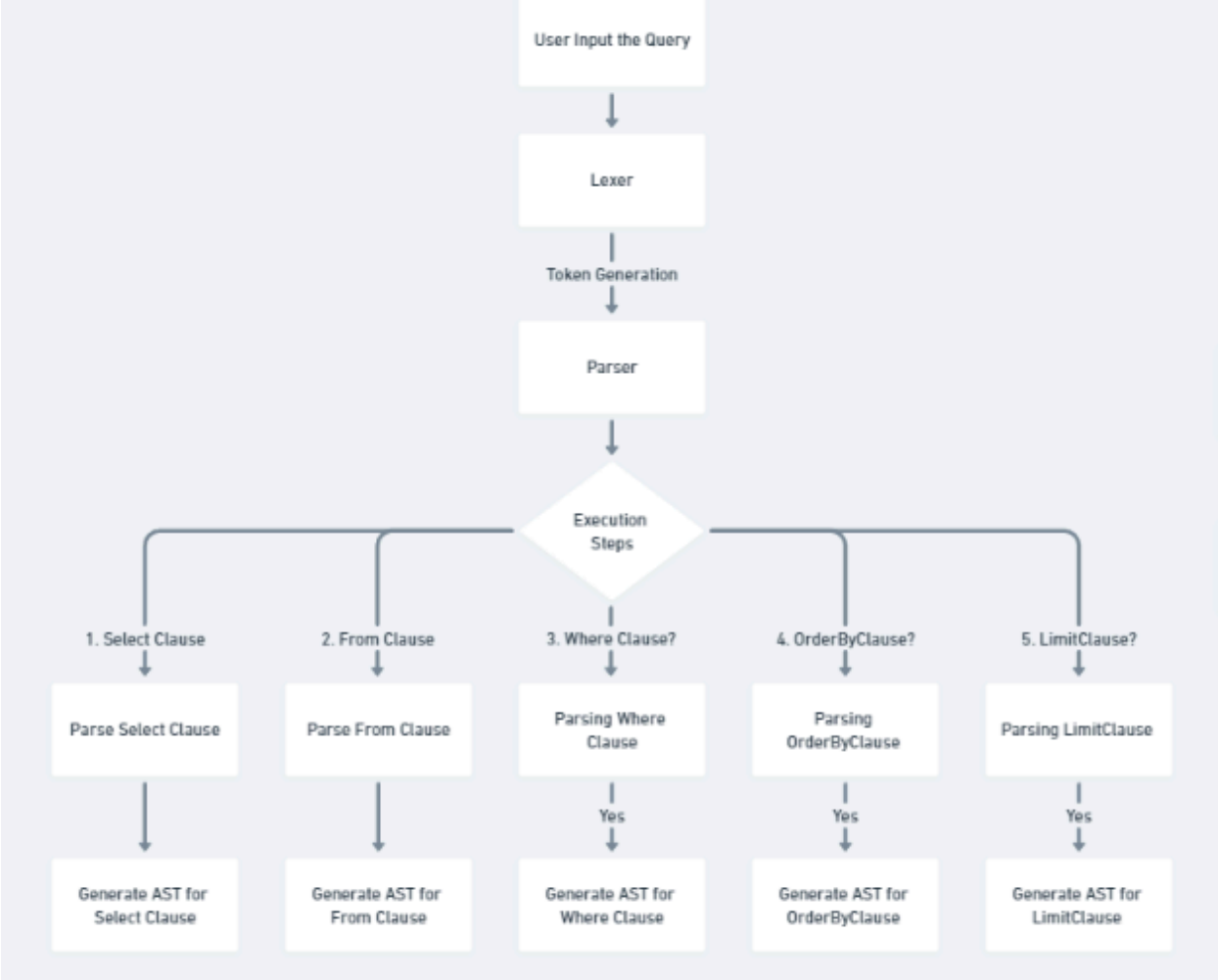
Rule	Invocations	Time (ms)	Total k	Max k	Ambiguities	DFA cach...
selectQuery...	1	0.097	1	1	0	1
selectQuery...	1	0.058	1	1	0	1
selectQuery...	1	0.041	1	1	0	1
selectQuery...	1	0.037	1	1	0	1
selectClaus...	1	0.160	1	1	0	1
selectClaus...	1	0.134	1	1	0	1
selectClaus...	1	0.032	1	1	0	1
orderByCla...	0	0.000	0	0	0	0
limitClause ...	0	0.000	0	0	0	0
selectExpr (...	1	0.046	1	1	0	1
whereExpr (...	0	0.000	0	0	0	0
whereExpr (...	0	0.000	0	0	0	0
whereExpr (...	0	0.000	0	0	0	0

Input size:	31 char, 1 lines
Number of tokens:	10
Parse time (ms):	36.029
Prediction time (ms):	6.407 = 17.78%
Lookahead burden:	22/10 = 2.20
DFA cache miss rate:	22/22 = 100.00%

☐ Show expert columns

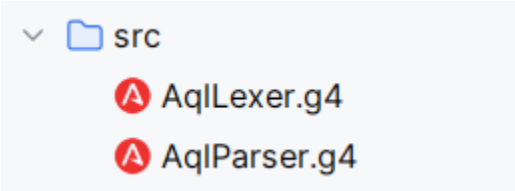
Ambiguity
Context-sensitivity
Predicate evaluation
Deepest lookahead

AQL Parser Execution Diagram



Code Implementation:

File Made:



AqlLexer Code:

Main.java | AqlParser.g4 | AqlLexer.g4

```
lexer grammar AqlLexer;
channels {
COMMENT_CHANNEL
}
WS: [ \t\r\n]+ -> skip;
UNICODE_BOM: (
'\uEFBBBF'
| '\uFEFF'
| '\u0000FEFF'
) -> skip;
COMMENT: (
SYM_DOUBLE_DASH ' ' ~[\r\n]* ('\r'? '\n' | EOF)
| SYM_DOUBLE_DASH ('\r'? '\n' | EOF)
) -> channel(COMMENT_CHANNEL);
// Keywords
// Common Keywords
SELECT: S E L E C T ;
AS: A S ;
FROM: F R O M ;
WHERE: W H E R E ;
ORDER: O R D E R ;
BY: B Y ;
DESC: D E S C ;
DESCENDING: D E S C E N D I N G ;
ASC: A S C ;
ASCENDING: A S C E N D I N G ;
LIMIT: L I M I T ;
OFFSET: O F F S E T ;
// other keywords
DISTINCT: D I S T I N C T ;
```

```
// aggregate function
COUNT: C O U N T ;
MIN: M I N ;
MAX: M A X ;
SUM: S U M ;
AVG: A V G ;
// other functions
TERMINOLOGY: T E R M I N O L O G Y ;
PARAMETER: '$' IDENTIFIER_CHAR;
ID_CODE : 'id' CODE_STR ;
AT_CODE : 'at' CODE_STR ;
fragment CODE_STR : ('0' | [1-9][0-9]*)+ ( '.' ('0' | [1-9][0-9]* ))* ;
// ---------- Delimited Regex matcher ------------
CONTAINED_REGEX: '{'WS* SLASH_REGEX WS* (';' WS* STRING)? WS* '}';
fragment SLASH_REGEX: '/' SLASH_REGEX_CHAR+ '/';
fragment SLASH_REGEX_CHAR: ~[/\n\r] | ESCAPE_SEQ | '\\/';
// ---------- ISO8601 Date/Time values ----------
fragment ISO8601_DATE
: YEAR MONTH DAY
| YEAR '-' MONTH '-' DAY
;
fragment ISO8601_TIME
: HOUR MINUTE SECOND ('.' MICROSECOND)? TIMEZONE?
| HOUR ':' MINUTE ':' SECOND ('.' MICROSECOND)? TIMEZONE?
;
fragment ISO8601_DATE_TIME
: YEAR MONTH DAY ('T' HOUR MINUTE SECOND ('.' MICROSECOND)? TIMEZONE?)?
| YEAR '-' MONTH '-' DAY ('T' HOUR ':' MINUTE ':' SECOND ('.' MICROSECOND)? TIMEZONE?)?
;
fragment MICROSECOND: [0-9][0-9][0-9] ;
```

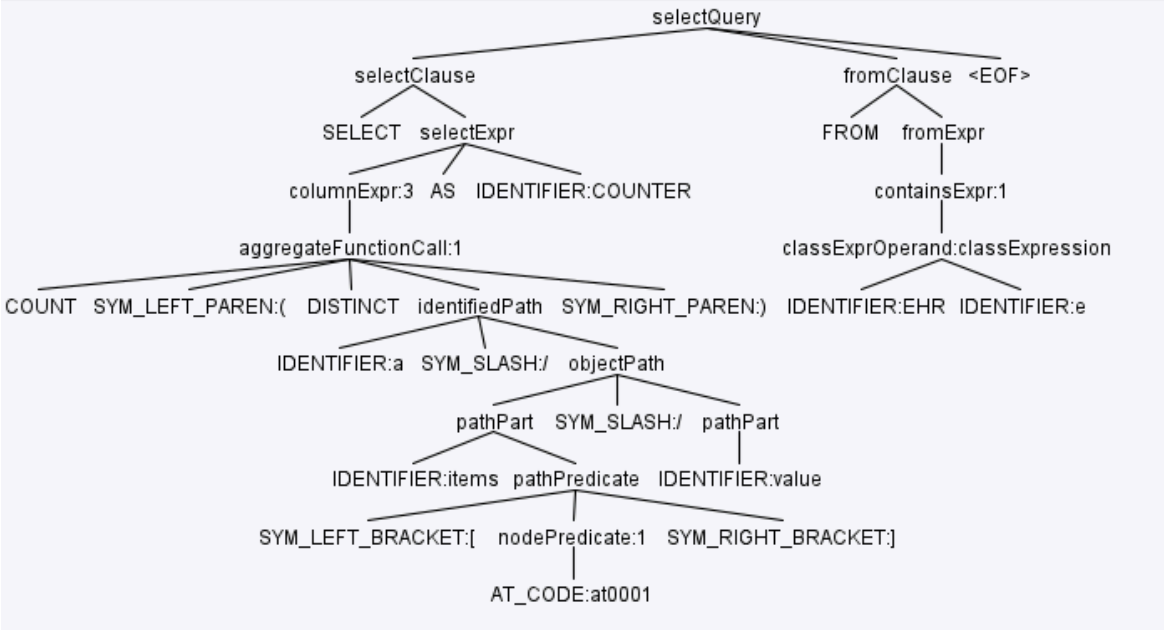
AqlParser Code:

```
parser grammar AqlParser;
options { tokenVocab=AqlLexer; }
selectQuery
: selectClause fromClause whereClause? orderByClause? limitClause? SYM_DOUBLE_DASH? EOF
;
selectClause
: SELECT DISTINCT? top? selectExpr (SYM_COMMA selectExpr)*
;
fromClause
: FROM fromExpr
;
whereClause
: WHERE whereExpr
;
orderByClause
: ORDER BY orderByExpr (SYM_COMMA orderByExpr)*
;
limitClause
: LIMIT limit=INTEGER (OFFSET offset=INTEGER) ?
;
selectExpr
: columnExpr (AS aliasName=IDENTIFIER)?
;
fromExpr
: containsExpr
;
whereExpr
: identifiedExpr
| NOT whereExpr
| whereExpr AND whereExpr
```

Query implementation:

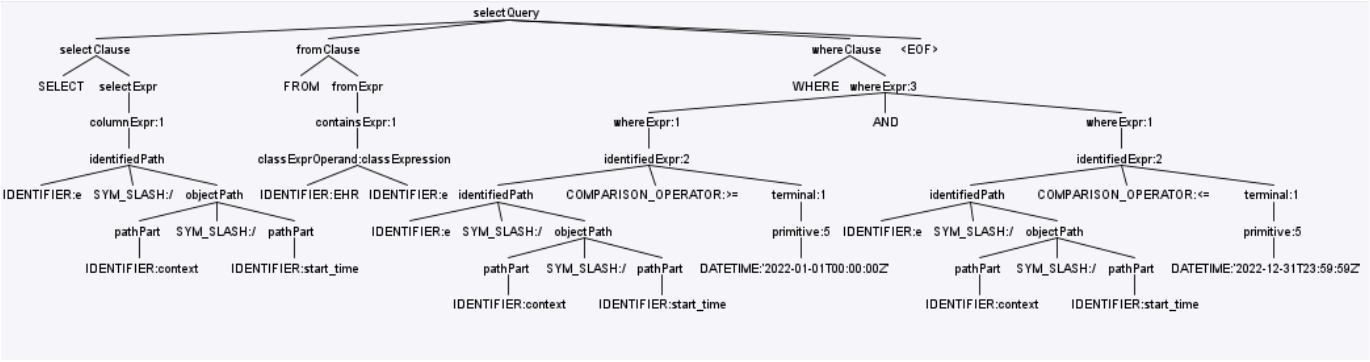
1. SELECT Query with Aggregate Function (COUNT)

```
SELECT COUNT(DISTINCT a/items[at0001]/value) AS COUNTER
FROM EHR e
```



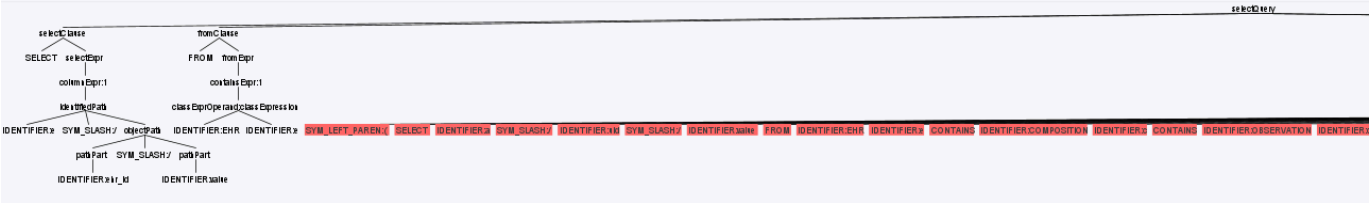
2. Query to Check Date/Time Type Matching

```
SELECT e/context/start_time
FROM EHR e
WHERE e/context/start_time >= '2022-01-01T00:00:00Z'
AND e/context/start_time <= '2022-12-31T23:59:59Z'
```



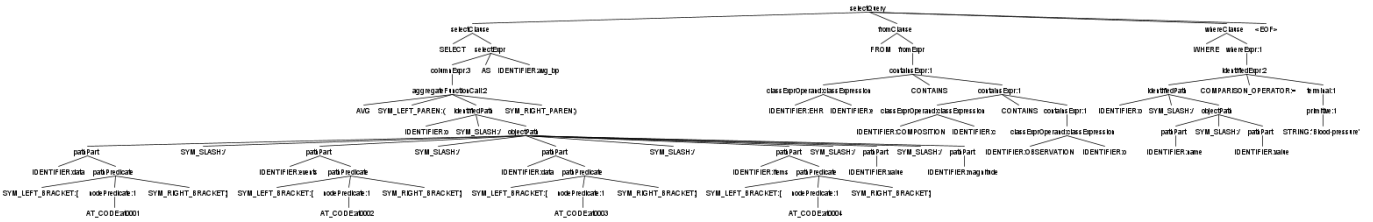
3. Nested Queries: Query to select all patients who have at least one abnormal laboratory result:

```
SELECT e/ehr_id/value
FROM EHR e
WHERE EXISTS (
    SELECT a/uid/value
    FROM EHR e CONTAINS COMPOSITION c CONTAINS OBSERVATION o CONTAINS CLUSTER cl CONTAINS ELEMENTa[openEHR-EHR-CLUSTER.lab_test.v1]
    WHERE o/data[at0001]/events[at0002]/data[at0003]/items[at0004]/value/magnitude < 0
);
```



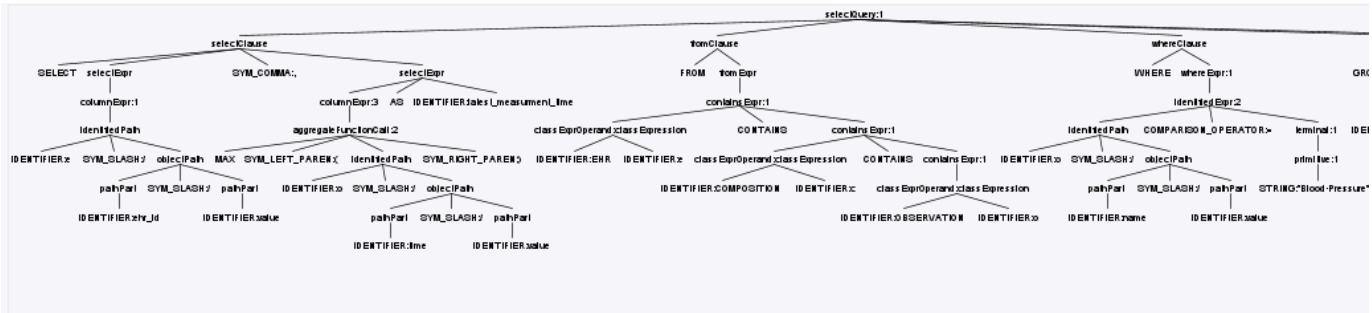
4. Subqueries with Aggregation: Query to find the average blood pressure of patients:

```
SELECT AVG(o/data[at0001]/events[at0002]/data[at0003]/items[at0004]/value/magnitude) AS avg_bp
FROM EHR e CONTAINS COMPOSITION c CONTAINS OBSERVATION o
WHERE o/name/value='Blood pressure'
```



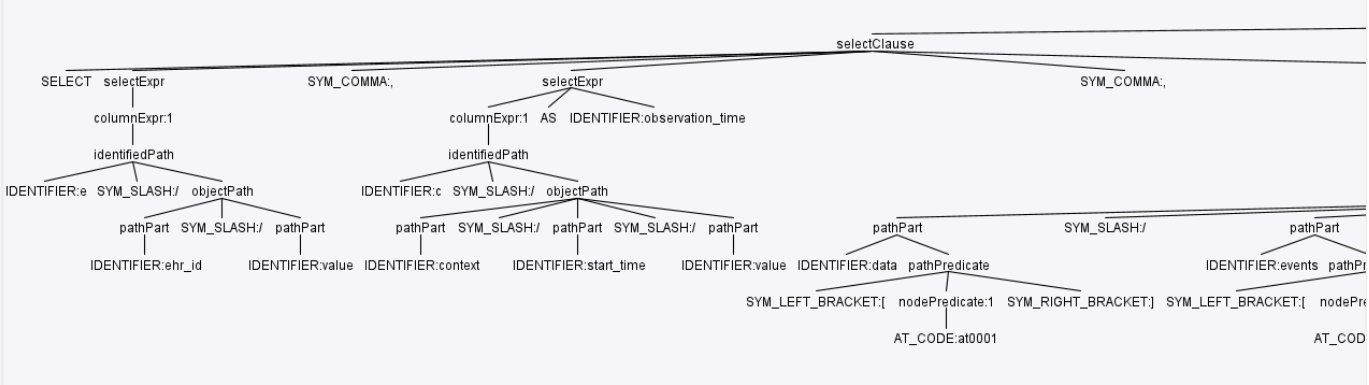
5.. Temporal Queries: Query to retrieve the latest blood pressure measurement for each patient:

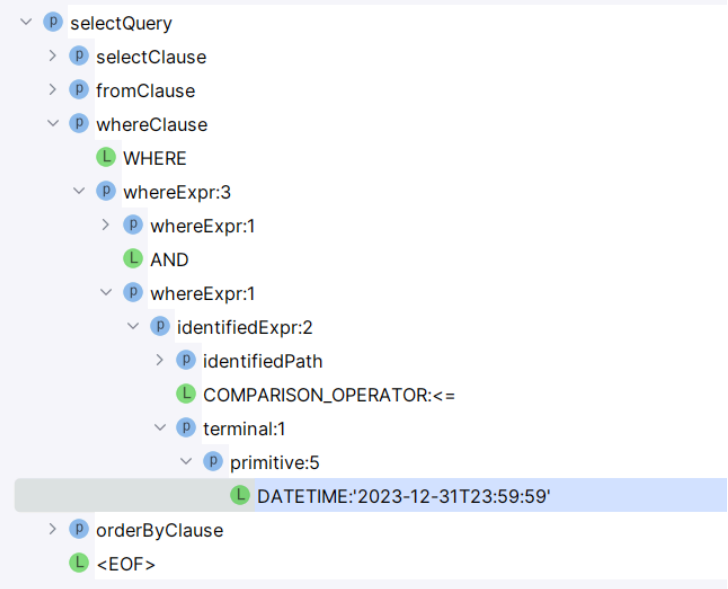
```
1  SELECT e/ehr_id/value, MAX(o/time/value) AS latest_measurment_time
2  FROM EHR e CONTAINS COMPOSITION c CONTAINS OBSERVATION o
3  WHERE o/name/value="Blood Pressure"
4  GROUP BY e/ehr_id/value
```



6. Filtering with Date Ranges: Query to retrieve all observations recorded between two specific dates

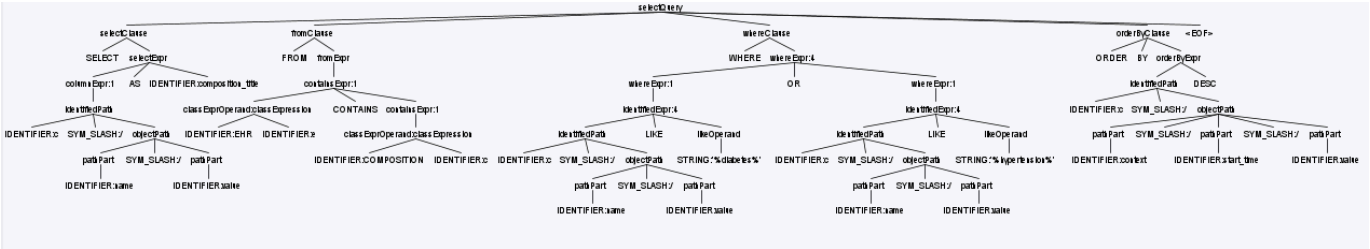
```
SELECT e/ehr_id/value, c/context/start_time/value AS observation_time, o/data[at0001]/events[at0002]/data[at0003]/items[at00
FROM EHR e CONTAINS COMPOSITION c CONTAINS OBSERVATION o
WHERE c/context/start_time/value >= '2023-01-01T00:00:00' AND c/context/start_time/value <= '2023-12-31T23:59:59'
ORDER BY c/context/start_time/value
```





7. Using Regular Expressions: Query to search for compositions with a specific title using regular expressions

```
SELECT c/name/value AS composition_title
FROM EHR e CONTAINS COMPOSITION c
WHERE c/name/value LIKE '%diabetes%' OR c/name/value LIKE '%hypertension%'
ORDER BY c/context/start_time/value DESC
```



Tools and Technologies Used

- ANTLR: ANTLR (ANother Tool for Language Recognition) is a powerful parser generator used for generating the lexer and parser. It provides capabilities for generating efficient and accurate lexers and parsers from grammar specifications.Installation:
    - Download ANTLR from https://www.antlr.org/download.html
    - Install ANTLR plugin in IntelliJ IDEA.
- Java: The programming language used for developing the AQL parser.
- JUnit: A unit testing framework for Java used for testing the AQL parser components.Installation:
    - Install IntelliJ IDEA from https://www.jetbrains.com/idea/download/
    - Install JUnit plugin in IntelliJ IDEA.

ANTLR4 Overview

ANTLR (ANother Tool for Language Recognition) version 4 is a powerful parser generator that facilitates the creation of parsers for structured text or binary files. It is widely used in software development for tasks such as building compilers, interpreters, transpilers, and other language processing tools. ANTLR4 operates based on grammar specifications defined by the user.

ANTLR4's Role in Parsing AQL Queries:

ANTLR4 is instrumental in parsing AQL queries by generating a parser from the specified grammar rules. The parser processes input queries, tokenizes them, constructs a parse tree, and performs syntax and semantic analysis.

Example of ANTLR4 Grammar Definition for AQL:

```
grammar AQL;

query: SELECT select_clause FROM from_clause WHERE where_clause;

select_clause: 'SELECT' (attribute | '*') (',' (attribute | '*'))*;

from_clause: 'FROM' archetype;

where_clause: 'WHERE' condition;

attribute: ID;

archetype: ID;

condition: ID comparison_operator value;

comparison_operator: '>' | '<' | '=' | '!=';

value: STRING | NUMBER | ID;
```

In this ANTLR4 grammar:

- **query** is the starting rule representing a complete AQL query.
- **select_clause** specifies the attributes to be retrieved.
- **from_clause** indicates the archetype or source of data.
- **where_clause** defines the conditions for filtering the data.
- **attribute**, **archetype**, **condition**, **comparison_operator**, and **value** are lexer rules defining the syntax elements of AQL.

ANTLR4 Role in Tokenization and Syntax Analysis:

ANTLR4 performs tokenization by breaking down input queries into tokens based on lexer rules. Each token represents a meaningful unit of the query, such as keywords, identifiers, operators, and literals. After tokenization, ANTLR4 constructs a parse tree based on the grammar rules, which represents the syntactic structure of the query.

PARSER FUNCTIONALITY:

Operation	AQL
Predicate : Determines if a condition is true or false.	Pass
Comparison Operator:Compares values to determine their relationship.	Pass
Logical Operator: Combines multiple conditions using AND, OR, or NOT.	Pass
Path Expression: Defines a path to navigate through structured data.	Pass
Existential: Checks if a specified item exists.	Pass
Relational/Boolean Operation:Performs operations like addition, subtraction, and logical operations.	Pass
Disjunction: Combines conditions using the OR operator.	Pass
Sorting: Orders query results based on specified criteria.	Pass
Update: Modifies data in the database.	Pass
Filtering: Retrieves specific data based on specified conditions.	Pass
Grouping: Groups query results based on specified attributes.	Pass
String functions: Manipulates strings, such as concatenation or substring extraction.	Pass
Numeric functions: Performs operations on numeric data.	Pass
Date and time functions: Handles date and time values, such as formatting or date arithmetic.	Pass
Aggregate functions: Performs calculations on sets of values, like COUNT, SUM, AVG, etc.	Pass
Nested: Includes one query within another query.	Fail
Join: Combines data from multiple tables based on related columns.	Pass
Union: Combines the results of two or more SELECT statements into a single result set.	Pass
TOP Operator: Retrieves a specified number of rows from the beginning of a result set.	Pass

Conclusion

The AQL parser effectively processes AQL queries by first tokenizing and identifying the input query, then constructing a parse tree based on the validated syntax. This parse tree guides the execution of the query to retrieve the specified data from the Electronic Health Records. The process ensures accurate and reliable query execution, making it a valuable tool for querying and extracting information from EHR systems.

References

- **ANTLR Documentation:** https://www.antlr.org/
- **Java Documentation:** https://docs.oracle.com/javase/
- **AQL Parser Paper:** https://specifications.openehr.org/releases/QUERY/latest/AQL.html
- **AQL Doc (Query Reference) :** https://www.ibm.com/docs/en/qsip/7.5?topic=aql-ariel-query-language
- **AQL Description :** https://openehr.atlassian.net/wiki/spaces/spec/pages/4915244/Archetype+Query+Language+Description